Complexity classes on spatially periodic Cellular Automata

Nicolas Bacquey

GREYC – Université de Caen Basse-Normandie / ENSICAEN / CNRS, Caen, France, `nicolas.bacquey@unicaen.fr`

Abstract

This article deals with cellular automata (CA) working over periodic configurations, as opposed to standard CA, where the initial configuration is bounded by persistent symbols. We study the capabilities of language recognition and computation of functions over such automata, as well as the complexity classes they define over languages and functions. We show that these new complexity classes coincide with the standard ones starting from polynomial time. As a by-product, we present a CA that solves a somehow relaxed version of the density classification problem.

1998 ACM Subject Classification F.1.1 Models of Computation, F.1.3 Complexity Measures and Classes

Keywords and phrases Language recognition, Cyclic languages, Computable functions, Algorithms on Cellular Automata, Linear space, Polynomial time, Density classification problem

Digital Object Identifier 10.4230/LIPIcs.STACS.2014.112

1 Introduction

Spatially periodic computation on cellular automata is a natural and well-defined notion (see e.g. [5]), though it seems it has not been studied extensively. This paper deals with algorithms and computational complexity of cellular automata (CA) acting on spatially periodic configurations, or, equivalently, on ring-cellular automata, i.e., CA whose underlying structure is a ring. The input of a ring-CA is a circular word, that is a finite word defined up to shift (around the ring). Clearly, a language recognized by a ring-CA is a cyclic language (as defined in [1], [2]), i.e., a language which is closed under shift, power and root. To our knowledge, our results are the first ones to deal with computational complexity on such automata.

A natural problem is the following, denoted MINIMAL-PERIOD: Given a spatially periodic configuration, compute its lexicographically minimal period, or, equivalently, given an input word w around a ring, compute its *canonical* root u, i.e., the lexicographically minimal word u such that $w = u^p$ up to shift, for some (maximal) integer p. We exhibit an algorithm on a ring-CA that computes the MINIMAL-PERIOD problem in polynomial time. This basic result allows us to compare computational complexity of ring-CA with complexity of standard CA whose input word is bounded by persistent symbols. Informally, we prove that the complexity classes on ring-CA and standard CA coincide down to polynomial time complexity. More precisely, we prove the following equivalences, for any cyclic language L:

- L is recognizable on a ring-CA iff L is LINSPACE;
- L is recognizable in polynomial time on a ring-CA iff L is recognized by a standard CA in linear space and polynomial time.

© Nicolas Bacquey;
licensed under Creative Commons License CC-BY
31st Symposium on Theoretical Aspects of Computer Science (STACS'14).
Editors: Ernst W. Mayr and Natacha Portier; pp. 112–124
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
SYMPOSIUM ON THEORETICAL ASPECTS OF COMPUTER SCIENCE

Moreover, we naturally extend those complexity results to functions, and use these results to show that the density classification problem (a well-studied problem defined in [6]) can be solved if we can use more states than the input alphabet $\{0, 1\}$.

2 Definitions and context

2.1 The computational model

We will use along this article the standard definition of cellular automata (CA) as a tuple $\mathcal{A} = (d, Q, N, \delta)$ (see [5]). In these lines, we will work with $d = 1$, *i.e.* cellular automata of dimension 1, so the underlying network will be $\mathbb{Z}$. Q denotes the set of *states*, and $N = \{-1, 0, 1\}$ is the *standard neighbourhood*. The local transition function of the automaton is denoted by $\delta : Q^{\{-1,0,1\}} \to Q$. As we work with cellular automata from the point of view of *language recognition*, we will identify a particular $\Sigma \subseteq Q$ as the *input alphabet*. We also introduce the *global transition function* $F_\delta : Q^{\mathbb{Z}} \to Q^{\mathbb{Z}}$ defined by the global synchronous application of δ over configurations of $\mathbb{Z}$ by $\forall C \in Q^{\mathbb{Z}}\ (F_\delta(C))(i) = \delta(C(i-1), C(i), C(i+1)))$.

We suppose that the reader is familiar with the notions of signals and computation layers on cellular automata. If it is not the case, we strongly encourage the reading of [5] or [7] for such general matters on cellular automata.

▶ **Definition 1.** We define a *Ring-Cellular Automaton (ring-CA)* as a cellular automaton whose initial configuration (and therefore any subsequent configuration) is periodic. Note that this model is equivalent to an automaton that would work on a finite, ring-like cell network. Let $u \in \Sigma^*$, we denote $C_u \in Q^{\mathbb{Z}}$ the bi-infinite repetition of u ($C_u = \dots uuu \dots$).

2.2 Recognition on ring-CA

On a standard CA where the input word is bounded by persistent states, it is easy to identify a particular cell of the configuration (e.g. the first one). We say that a word is *accepted* (or *rejected*) when this particular cell enters a persistent acceptance (or rejection) state (see [11] for all subjects related to language recognition on CA). However, if the configuration we are working on is periodic, the identification of a particular cell is intrinsically impossible. Therefore, we must define the acceptance or rejection of a word as a global phenomenon. Here are the two definitions of acceptance that we will use on ring-CAs:

▶ **Definition 2.** We say that a language $L \subset \Sigma^*$ is *ring-recognizable* if there exists a ring-CA $\mathcal{C}$ such that for all $u \in \Sigma^*$, the automaton $\mathcal{C}$ given the periodic configuration C_u as an input evolves in such a way that for some time t:

- *weak recognition:* All cells enter the same particular subset of states, either $S_a \subset Q$ (for accept) or $S_r \subset Q$ (for reject) and never leave it afterwards.
- *strong recognition:* All cells enter the same particular state (S_a and S_r are singletons), with the transition function defined so that this configuration is a fixed point of the global transition function F_δ.

Note that our definition of strong recognition is the best recognition we can hope for on a ring-CA, due to its intrinsically spatially periodic nature. Also note that due to that very nature, for any language that is weakly or strongly recognized, there exists a time exponentially bounded in the length of the period after which all the cells are in their definitive subset of states (S_a or S_r).

While it is obvious that any strongly recognizable language is weakly recognizable (actually by the same automaton), we will show that those two definitions are actually equivalent in section 4.

▶ **Definition 3.** We introduce the *shift function* denoted as $\sigma : \Sigma^* \to \Sigma^*$ and defined for all $u = u_1u_2...u_n \in \Sigma^*$ by $\sigma(u) = u_2...u_nu_1$.

Since one cannot discriminate between configurations produced by a word, and those produced by shifts, powers or roots of this word, every language recognized by a ring-CA must be closed under shift, power and root operations. More formally, we can only recognize *cyclic* languages defined as follows:

▶ **Definition 4.** A language $L \subset \Sigma^*$ is said to be *cyclic* (see [2]) if $\forall w \in \Sigma^*, \forall k \geq 1$:
- $w \in L$ iff $\sigma^k(w) \in L$,
- $w \in L$ iff $w^k \in L$.

We say that a cyclic language L is *strongly (resp. weakly) ring-recognizable* if there exists a ring-CA that strongly (resp. weakly) recognizes it.

2.3 Computability of functions on ring-CA

We can also design our ring-CAs to compute functions, as an extension of language recognition, which is a particular function with $\{0, 1\}$ as its output set. Intuitively, we want to give the input of the function as a periodic configuration of the CA, wait some time, and read the result of the function when the CA has reached a fixed point. We now give the following formal definition of a computable function:

▶ **Definition 5.** Let $f : \Sigma^* \to \Gamma^*$ be a function over words of Σ^*.
f is said to be *ring-computable* if there exists a *ring-CA* $\mathcal{A}$ such that $\forall u \in \Sigma^*$:
- there exists an integer t such that $F_\delta^t(C_u) = C_{f(u)}$,
- $F_\delta(C_{f(u)}) = C_{f(u)}$ (i.e. $C_{f(u)}$ is a fixed point of F_δ).

We define the time complexity of the computation of such a function f on a ring-CA $\mathcal{C}$ over a word u as the smallest t such that $C_{f(u)} = F_\delta^t(C_u)$.

We define an analogous to cyclic languages in the case of functions. As we read the output of the function on the automaton where the word was input, we need an additional condition on the length of the output of those functions ($\|w\|$ is the length of the word w):

▶ **Definition 6.** A function $f : \Sigma^* \to \Gamma^*$ is said to be *cyclic* if $\forall u \in \Sigma^*, \forall k \geq 1$:
- $f(\sigma^k(u)) = \sigma^k(f(u))$,
- $f(u^k) = f(u)^k$,
- $\|f(u)\| = \|u\|$.

2.4 Complexity classes and results

▶ **Definition 7.** Due to the circular nature of the model, we define the *size* n of an input as the length of its minimal period.

We note that, as a consequence of the fact that our work space can be seen as a finite ring, we can't use more than a linear space for our computations on ring-CAs, with respect to the input size. Therefore, every language or function we study must be in $LINSPACE$, which is a robust complexity class. Then we give the following definitions:

▶ **Definition 8.** We denote the complexity class of languages that are strongly recognizable in polynomial time on a ring-CA as $TIME_{ring}(POLY(n))$. We also denote the complexity class of languages that are recognizable in linear space and polynomial time on a bounded input CA (or equivalently, classical models such as Turing machines, RAM machines...) as $SPACETIME(n, POLY(n))$. We will abusively use this notation to talk about complexity classes of functions.

We will prove our results along these lines:

▶ **Theorem 9 (Recognition of languages).** *Let L be a cyclic language, then:*
- *L is strongly ring-recognizable $\iff$ $L \in LINSPACE$,*
- *$L \in TIME_{ring}(POLY(n)) \iff L \in SPACETIME(n, POLY(n))$.*

▶ **Theorem 10 (Computability of functions).** *Let f be a cyclic function, then:*
- *f is ring-computable $\iff$ $f \in LINSPACE$,*
- *$f \in TIME_{ring}(POLY(n)) \iff f \in SPACETIME(n, POLY(n))$.*

3 From the cyclic model to the standard model

Throughout the rest of the paper, $L \subset \Sigma^*$ will denote a cyclic language. In this section, we will prove that L is ring-recognizable implies $L \in LINSPACE$.

This part of the proof is quite straightforward: Indeed, if we consider a ring-automaton $\mathcal{C}$ that weakly recognizes L, it is easy to design a standard cellular automaton $\mathcal{A}$ that recognizes it: it suffices to add an additional layer to $\mathcal{C}$, in which $\mathcal{A}$ will write the mirrored input word in a preprocessing phase. The automaton will now simulate two computations of $\mathcal{C}$ in parallel, connecting the beginnings and ends of the two simulated configurations (see Fig 1). After this linear time preprocessing, $\mathcal{A}$ will simulate $\mathcal{C}$ step-by-step.

Though, one must be careful when defining the halting conditions of $\mathcal{A}$. Indeed, there can be a state where all simulated cells of $\mathcal{C}$ are in the "accept" or "reject" subset of states, but the computation has not ended yet. Therefore, $\mathcal{A}$ also has to construct the exponential time bound before which we are sure that the computation of $\mathcal{C}$ is over, then decide if the word is accepted or rejected by checking the state of an arbitrary cell of its array (by construction, all cells are in the same "accept" or "reject" subset of states at that moment).

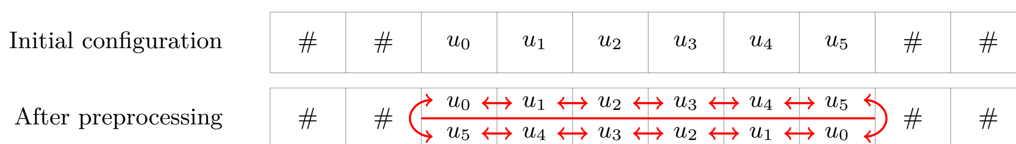


Figure 1 Simulating a ring-CA on a standard CA.

For the proof that $L \in TIME_{ring}(POLY(n))$ implies $L \in SPACETIME(n, POLY(n))$, it suffices to construct the polynomial time bound before which the computation of $\mathcal{C}$ is over instead of the exponential one, and the construction still holds.

4 From the standard model to the cyclic model

In this section, we will prove that $L \in LINSPACE$ implies L is ring-recognizable.

Let $\mathcal{A}$ be a standard CA recognizing $L \subset \Sigma^*$. We will design a ring-CA $\mathcal{C}$ that will mimic the computation of $\mathcal{A}$. If we denote as Q the work alphabet of $\mathcal{A}$, our automaton $\mathcal{C}$ will use

a new set of states $Q' = Q \times \Sigma \times \omega$, where ω will handle all the specific constructions of $\mathcal{C}$. We also introduce a bijection $val : \Sigma \cup \{\vdash\} \to [0, |\Sigma|]$ such that $val(\vdash) = 0$. We want each cell of $\mathcal{C}$ to retain its input letter, which means that the Σ part of Q' will never change. All our computation will be done through two computational layers, namely $Q \times \omega$.

4.1 Global vision

We will now present a mechanism that is able to find a minimal period of every periodic configuration it is started on. Note that this minimal period is defined up to a shift. We will combine this mechanism to a simulation of $\mathcal{A}$ over the word contained in this minimal period.

The periodic configuration of $\mathcal{C}$ will be divided into *intervals*, in which we will simulate the computation of $\mathcal{A}$ over the word contained in the interval. We design those intervals so that two different adjacent intervals will merge over time.

For a better understanding of the algorithm, we can imagine that each interval can be in three states, namely "*merge-to-the-right*", "*merge-to-the-left*" and "*waiting*". Those states can evolve over time, according to the following rule:

▶ Rule 11. If an interval is in the *merge-to-the-right* state and its right neighbour is in the *merge-to-the-left* state, then they must merge together, and it is the only way for intervals to merge.

When an interval is created, we use the Q layer to simulate the computation that $\mathcal{A}$ would have done over the word contained in the interval. When the computation is over, and if the interval has not merged yet, a special success/failure state of Q is propagated over all the cells of the interval.

At the beginning, each cell will define its own interval, and those intervals will start to merge as the computation goes. We state that intervals will merge until every two adjacent intervals are equal (see fig 2 for an intuition of the fusion process).

Once this configuration where every interval contains the same word is reached, the simulation of $\mathcal{A}$ in this interval and the propagation of the success/failure state will properly match our first recognition condition (i.e. all cells enter the same particular subset of states and never leave it).

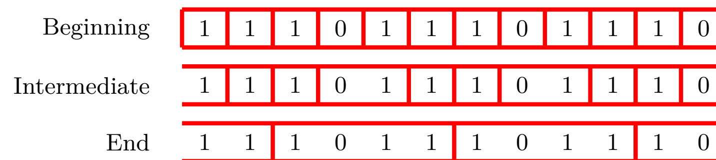


Figure 2 Outline of the merging process on a cyclic configuration.

4.2 Basic tools

Intervals: Let us identify a specific sub-layer of our work alphabet ω, which is of the form $\omega = \omega' \times \{\#, \varnothing\}$; We define the *intervals* of a configuration as the maximum sets of adjacent cells beginning with a # and containing exactly one #. The size of an interval I is the number of cells it contains, denoted by $size(I)$ (see Fig 3).

We define the *content* of an interval as the word formed by the concatenation of the canonical projection of the states of its cells over Σ, preceded by the special symbol "$\vdash$". We say that two intervals are *different* if their contents are different words.

Figure 3 Intervals of size 2, 5 and 3.

We will construct a method ensuring the following property:

▶ **Lemma 12.** *There exists an integer C such that, if two adjacent intervals have different contents at time t, then at least one of them will merge before time $t + C \times n^3$, where n is the size of the larger of both intervals.*

Signals and pointers: Every interval will have a signal (as defined in, e.g. [8]) going back and forth in it, starting from its left border, and will also keep a pointer over the letters of its content (see Fig 4). Each time the signal will enter a cell containing the pointer from its right, it will move the pointer one letter to the right (for this purpose, we suppose that the first "$\vdash$" is stored in the first cell of the interval). When the pointer is at the rightmost letter of the content, its next move brings it back to "$\vdash$" (in n time steps).

At the first time step, a # is written on each cell, so that every cell will form an interval of size 1, a signal starts in each interval, and all the pointers are set to the first letter of the two-letter content of each interval, namely "$\vdash$". For an interval of size n, let a_i denote the symbol currently pointed, with $i \in [0, n]$ and $a_0 =$ "$\vdash$".

Let $k = |\Sigma| + 1$. The signal will wait $t_i = kn^2 + 2n \times val(a_i)$ time steps on each border, then move at speed ± 1 to the other border. These times (t_i) will encode the content of the interval, thus allowing an interval to compare itself with its neighbours, and merging if necessary. Note that by standard techniques (see [8]), the function $n \mapsto kn^2 + 2n \times val(a_i)$ is easily time-constructible using properly defined signals carrying $val(a_i)$ in their state, thus allowing use to wait such times on the borders.

We say that an interval is in the *merge-to-the-left* (*resp. merge-to-the-right*) state of our global vision if its signal is on the left border (*resp.* right border), and in the *waiting* state otherwise.

4.3 Merging process

Let us consider what happens when two adjacent intervals I_1 and I_2 have their signals meet on their common border. It ensues from our previous definitions that I_1 is in the *merge-to-the-right* state, and I_2 is in the *merge-to-the-left* state. Therefore according to rule 11, a merging of I_1 and I_2 must occur. We do so by erasing the # symbol defining their common border, and destroying the signals. Then new signals are sent to the beginning and the end of the new interval. Those two signals will reset the pointer to the beginning of the new word, and reset the simulation of the computation of $\mathcal{A}$. Finally, the whole process starts again (see fig 5).

Proof of lemma 12. Let I_1 and I_2 be two adjacent intervals with different contents a and b, and S_1 and S_2 be their respective signals. Let I_1 be the interval on the left and I_2 be the one on the right. Let $size(I_1) = m$ and $size(I_2) = n$. It suffices to consider the case where neither I_1 nor I_2 merge with other intervals during the time spans we consider. We will prove our lemma by considering two different cases, whether I_1 and I_2 have different sizes or not.

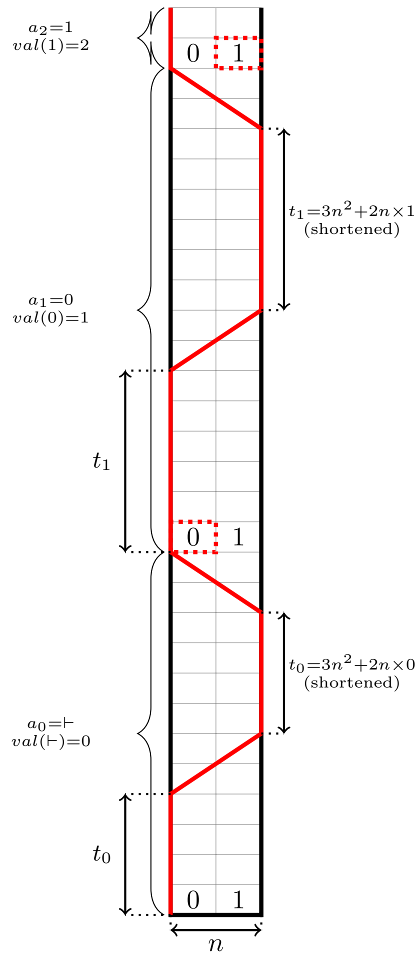


Figure 4 Basic move of a signal in an interval (with $|\Sigma| = 2$). The dashed square figures the pointer, which is set to "⊢" at the beginning and until further notice.

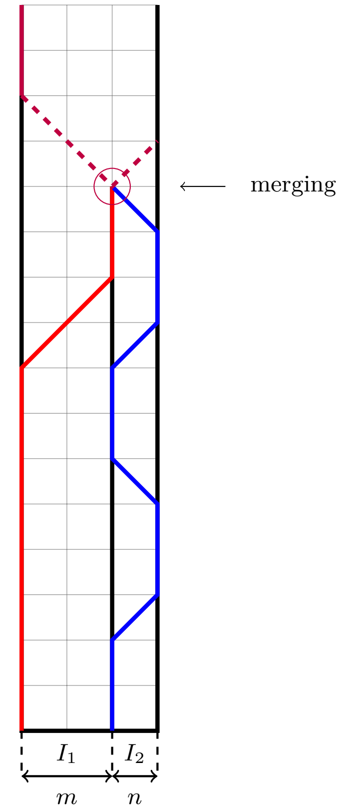


Figure 5 The merging process – different sizes

Merging intervals of different sizes

We suppose without loss of generality that $m > n$, and we will prove that the time interval when S_1 is on the common border cannot be contained in the time interval when S_2 is away from that border. The signal S_1 will wait at least $T_1 = km^2$ on each border, in particular on the border between I_1 and I_2. Now S_2 will be away from a border during at most $T_2 = kn^2 + 2(k-1)n + 2n = kn^2 + 2kn$ consecutive time steps (since the journey back and forth from a border lasts $2n$ time steps, and $val(a_i) \leq k-1$).

Since $m \geq n+1$, we have $T_1 \geq k \times (n+1)^2 = kn^2 + 2kn + k = T_2 + k > T_2$. $T_1 > T_2$ means that S_2 cannot be away from the common border long enough not to meet S_1, which means that I_1 and I_2 will merge eventually.

Complexity analysis: It is easily seen that I_1 and I_2 will merge together in time $O(max(n,m)^2)$, because the signal of the larger of both intervals cannot achieve a complete trip back and forth without encountering the signal from the smaller one, and such a trip takes a time $O(max(n,m)^2)$. *A fortiori*, there exists an integer c such that I_1 and I_2 will merge before time $t + c \times max(n,m)^3$, which proves Lemma 12 for the case $n \neq m$.

Merging intervals of equal sizes and different contents

We now consider the case where two adjacent intervals have the same size n, but different contents. We will show that they will merge when their pointer will be set on different symbols. Let us start by defining some useful notions.

Asynchronicity: During the lifetime of I_1, we consider the case where its signal S_1 does one or more round trips between its borders (the contrary would mean that I_1 has merged before, see Fig 6). Let t_1 be any time when S_1 reaches the left border. S_1 comes from the right border, where it has waited during a certain time interval T_0. Note that by construction, we must have $T_0 \geq 2n$, because $kn^2 \geq 2n$.

I_1 has not merged during the time interval T_0. That means that S_2 must have been away from the left border of I_2 during that time. There are two cases: either S_2 was on the right border, or it was travelling through I_2. Note that S_2 can't have been travelling more than $2n-2$ time steps, because that would mean it would have encountered the left border. As $T_0 > 2n-2$, this means that there must be a time t_1' during T_0 when S_2 was on the right border of I_2. Let us now define t_2 as the first time after t_1' when S_2 will reach the left border of I_2.

▶ **Definition 13.** As we have defined a specific time for each interval, we can now define the *asynchronicity* between them, as $\delta = t_1 - t_2$.

We note that this asynchronicity can be defined each time S_1 reaches the left border of I_1. Those times t_1 and t_2 are special in our construction, as they are the times when the intervals move their pointer one cell to the right (or move it back to the first symbol of their content). Thus, we can associate a pair of symbols $(a_i, b_j) \in \Sigma \cup \{\vdash\}$ to each pair of times (t_1, t_2) that define an asynchronicity (these symbols are the ones which are newly pointed by I_1 and I_2 respectively). Now let us consider the boundaries of the asynchronicity δ: If $a_{i-1} = b_{j-1}$, we must have $-n < \delta < n$. Indeed, the contrary would mean that S_1 and S_2 would have met before t_1, and a merging would have occurred (see Fig 7 for the absurd cases where $a_{i-1} = b_{j-1}$ and $\delta \leq -n$ (7a) or $\delta \geq n$ (7b)).

Let us prove that the sequences (a_j) and (b_j) cannot be the same if $a \neq b$. Indeed, the contrary would mean that (a_j) and (b_j) would coincide on every symbol, including the only "$\vdash$" a and b contains, and therefore any subsequent symbol until the next "$\vdash$". This would mean that $a = b$, hence a contradiction (let us recall that "$\vdash$" marks the beginning of the content).

We consider the first pair of times (t_1, t_2) when the associated symbols (a_i, b_j) are different (it exists, since we have $a \neq b$). If we note δ the associated asynchronicity, we have $|\delta| < n$, because $a_{i-1} = b_{j-1}$. We state that I_1 and I_2 will merge before their signals can go back and forth. We will identify two cases, whether $a_i < b_j$ or not (See Fig 8).

We will first see what happens if $a_i < b_j$ (See Fig 8a). S_1 will wait a time $t_0 = kn^2 + 2n \times val(a_i)$ on each border, and S_2 will wait $t_0 + \Delta_t$, with $\Delta_t = 2n \times (val(b_j) - val(a_i))$. Since $a_i < b_j$, we must have $val(b_j) > val(a_i)$, and therefore $\Delta_t \geq 2n$. Now S_1 will arrive on the right border of I_1 at time $t_1 + t_0 + n$, and S_2 will stay on this border from time $t_1 + \delta$ to time $t_1 + \delta + t_0 + \Delta_t$. Let us prove that $t_0 + n \in [\delta, \delta + t_0 + \Delta_t]$, which means that the two signals will meet on the common border.

Since $t_0 > 0$ and $|\delta| < n$, we have $t_0 + n > \delta$. $\Delta_t \geq 2n$, so $\delta + \Delta_t > n$, and therefore $t_0 + n < \delta + t_0 + \Delta_t$.

If $a_i > b_j$ (See Fig 8b), the merging occurs later, but for similar arguments. t_0 is now defined as the waiting time of S_2, and Δ_t as $2n \times (val(a_i) - val(b_j))$. We will have to prove

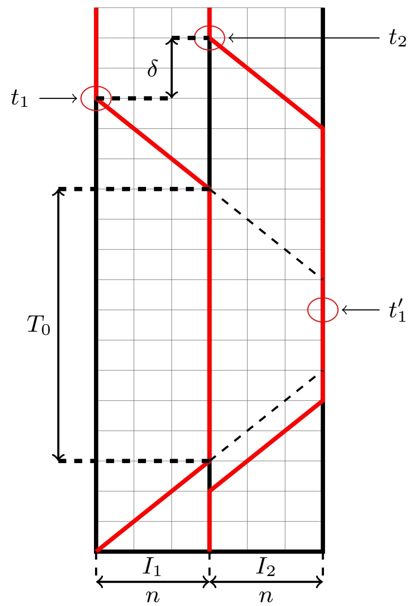
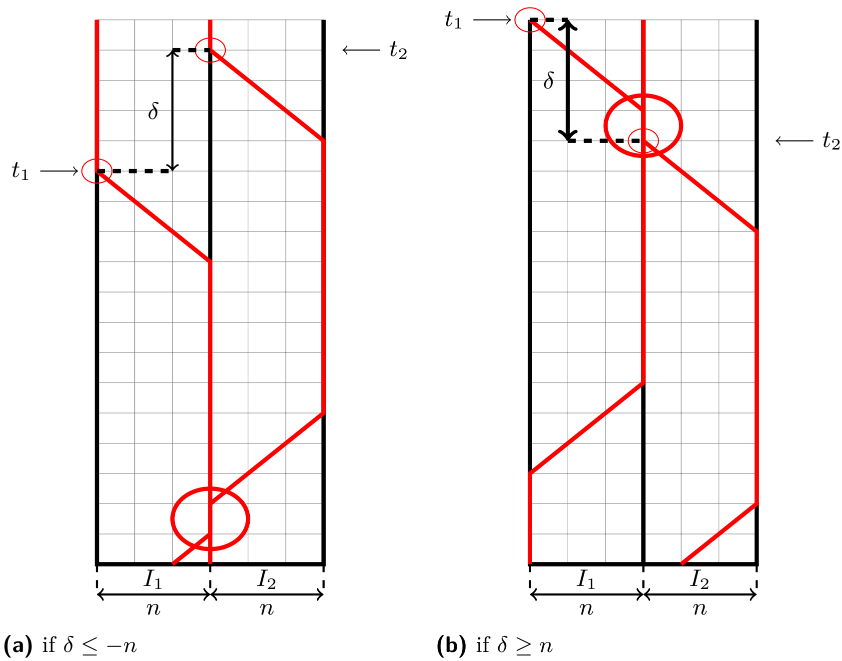


Figure 6 Definition of asynchronicity between I_1 and I_2 at time (t_1, t_2).

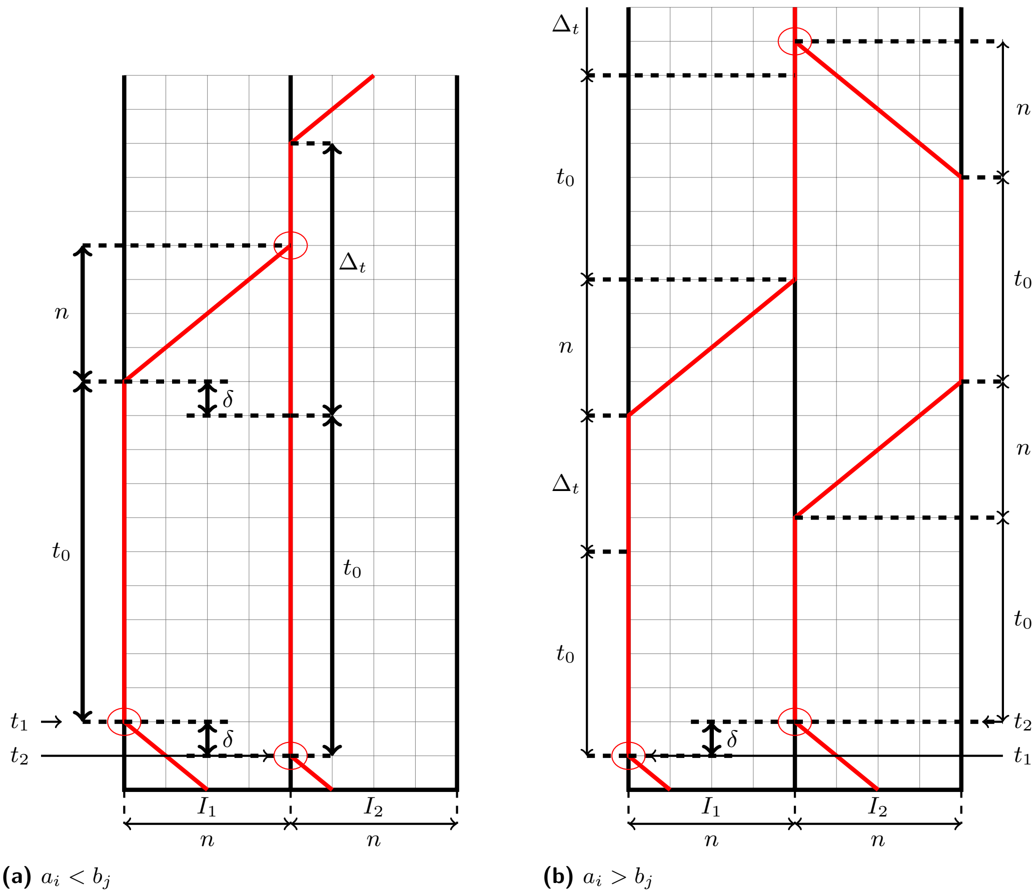


(a) if $\delta \leq -n$ **(b)** if $\delta \geq n$

Figure 7 Boundaries of asynchronicity.

that $2t_0 + 2n + \delta \in [t_0 + \Delta_t + n, 2t_0 + 2\Delta_t + n]$, using all former remarks plus the fact that $\Delta_t \leq 2n^2$. The complete proof is left to the reader (see Fig 8b for an intuition).

Complexity analysis: Once again, let (t_1, t_2) be the first pair of times when their associated symbols (a_i, b_j) are different. We claim that there exists a c such that I_1 and I_2 will merge



(a) $a_i < b_j$

(b) $a_i > b_j$

Figure 8 The merging process – intervals of equal sizes.

before time $t_1 + c \times n^2$ (or $t_2 + c \times n^2$ equivalently, since $|t_1 - t_2| < n$). Indeed, it is true because both signals cannot complete a trip back and forth from t_1 (resp. t_2). Now let us see how much time happens before such a pair of times (t_1, t_2) is encountered. The pointers over the contents of I_1 and I_2 change with delay $O(n^2)$ by construction. Therefore, the signals S_1 and S_2 have a period of $O(n^3)$, which means that they must encounter in time $O(n^3)$, thus proving the last remaining case of Lemma 12. ◀

Now let us recall that at the beginning, each period of our workspace is divided into n intervals of size 1. Lemma 12 states that while there exists two adjacent different intervals at least one of them must merge before time $O(n^3)$. Therefore, the number of intervals in a single period of the workspace must decrease every $O(n^3)$ time steps until all intervals are equal, which means there exists only one interval per period. At this point, which happens after $O(n^4)$ time steps, it suffices to wait for the end of the simulation of $\mathcal{A}$ in each interval for the ring-CA $\mathcal{C}$ to enter in a loop in which no merging can occur. The acceptance or rejection of the input can be then decided by projection of the states of $\mathcal{C}$ over Q. This behaviour matches our definition of weak recognition, therefore L is weakly ring-recognizable.

The proof remains the very same if $L \in \mathit{SPACETIME}(n, \mathit{POLY}(n))$ and we want to prove $L \in \mathit{TIME}_{ring}(\mathit{POLY}(n))$. Indeed, our construction only adds an $O(n^4)$ time before the simulation of $\mathcal{A}$ decides in polynomial time whether or not the input belongs to L.

4.4 Strengthening the construction to achieve strong recognition

We will now add a few enhancements to our construction, so that the ring-CA $\mathcal{C}$ will strongly recognize a language if the underlying standard automaton $\mathcal{A}$ recognizes it. The first thing we have to do is to add two states encoding the acceptance or rejection of a word to our work alphabet Q' (we need this to cope with our definition of *strong recognition*). Our new work alphabet is now: $Q' = (Q \times \Sigma \times \omega) \cup \{accept, reject\}$ The naive way to achieve strong recognition is to put the cells of an interval in the *accept* or *reject* state when a local simulation of $\mathcal{A}$ is over. This leads to a crucial issue, which will be explained in the following paragraphs.

False positives: The main issue that arises when we enforce the *accept* or *reject* states into the cells instead of letting the signals work is the risk of false acceptance (or rejection): one can imagine a case where an interval decides that its word is accepted, then overwrites itself with the *accept* state, whereas there are other intervals in the configuration, whose contents are different from itself and thus need to merge with it, leading to an error. Even if we somehow manage to reconstruct the interval afterwards, its initial content (a word from Σ^*) is lost. The following mechanism solves this problem.

Saving private content: Regarding the previous paragraphs, an interval should not overwrite its content with *accept* or *reject* until we are sure that its initial input can be found elsewhere, and then restored if necessary. We will design a mechanism that will ensure that an interval overwrites itself only if the content of its left neighbour is the same as its proper content. Our Lemma 12 ensures that if any pair of adjacent intervals I_1 and I_2 do not merge during a certain time $c \times n^3$, then they have the same content. We want the intervals to detect those cases that will eventually happen, and only to overwrite themselves when they are assured that their left neighbour has the same content as themselves. As a consequence of Lemma 12, there exists a constructible time t_{eq} such that if two adjacent intervals evolve together during a time t_{eq}, then their content is equal. We will exploit that property by adding a "timer" layer to the intervals that will wait for time t_{eq} and check if the left border of the interval is periodically visited by the signal of its left neighbour (this ensures that its left neighbour has not merged yet; One can check this by leaving a "token" when a signal reaches the right border of its interval). We redesign the overwriting process so that it can only happen when the interval has waited for at least t_{eq} alongside its left neighbour. This ensures that an interval only overwrites itself when it is sure that its content can be restored later.

Restoring lost content: As the right neighbour of an interval may have overwritten itself, we must ensure that each time the content of an interval changes (i.e. when it merges), its former content is sent to its right to replace the *accept* or *reject* states. We can copy the content of an interval by the method detailed on Fig 9. When an interval gets its old content back by this method, it immediately creates a new signal, begins its computation as a newly created interval, and checks if its right neighbour should also be restored.

5 Extensions

Towards function computing: We suppose that we are given a standard CA $\mathcal{A}$ that computes a cyclic function f. A few minor tweaks to the mechanism that gave us strong recognition are enough to obtain a ring-CA that computes the function f. Indeed, instead

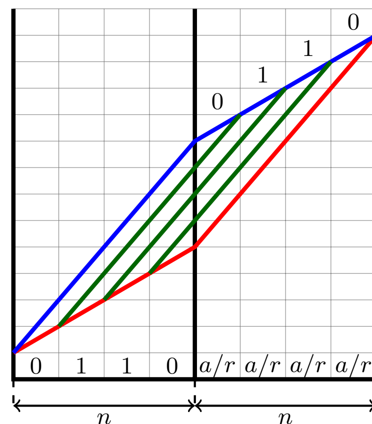


Figure 9 An intuition of the copying mechanism (a/r means accept/reject).

of adding the $\{accept, reject\}$ set of states to the automaton, we will add Γ, the output alphabet of the function f. When an interval was supposed to overwrite itself with the *accept* or *reject* state, it writes the output of f over its content instead (it has enough room to do so, as $\|f(u)\| = \|u\|$). *Ceteris paribus*, our new automaton exactly computes f, therefore f is ring-computable.

Density classification problem: We consider the density classification problem, as defined in [6] and studied in [3], [4]. The goal of this problem is to design a CA that works on periodic configurations on the alphabet $\{0, 1\}$ and converges towards the bi-infinite configuration composed only of 1 (denoted as $1^{\mathbb{Z}}$) if there are more 1s than 0s in the initial configuration and $0^{\mathbb{Z}}$ otherwise. This can be seen as the computation of a specific function, which happens to be cyclic. Therefore, our construction provides a solution to the open problem of the density classification in deterministic case, by using more states than the sole input alphabet.

6 Conclusion and open problems

We note that our method computes an optimal leader election on a spatially periodic configuration: the leaders are the cells where the # finally remain. We can shift those symbols in such a way that they delimit the lexicographically minimal word, thus solving in polynomial time the MINIMAL-PERIOD problem defined in the introduction.

Whereas spatially periodic CA have been studied in the framework of dynamic systems, e.g. for proving that a given automaton is injective or surjective (see [5], [9]), very little work has been done to our knowledge in the framework of language recognition or computability. Several cyclic languages can be suggested from this article: the majority languages, which consist of the languages where a specific symbol appears more than the other ones, or the cyclic closure of regular languages. However, it is difficult to describe how comprehensive and interesting cyclic languages can be. It is also natural to extend spatially periodic computation problems to 2-dimensional cellular automata. As usual, it raises several issues, including the very definition of a somehow "minimal pattern" of a bi-periodic infinite picture, and its computability on a cellular automaton. Thus, in the 2-dimensional case the definition of cyclic languages and cyclic functions fitting our framework is unclear, and is a fine food for thought for future works.

References

1 Marie-Pierre Béal, Olivier Carton, and Christophe Reutenauer. Cyclic languages and strongly cyclic languages. In *STACS*, volume 1046 of *LCNS*, pages 49–59, Berlin, 1996. Springer.
2 Olivier Carton. A hierarchy of cyclic languages. *ITA*, 31(4):355–369, 1997.
3 Nazim Fatès. Stochastic cellular automata solve the density classification problem with an arbitrary precision. In *STACS*, volume 9 of *LIPIcs*, pages 284–295, 2011.
4 Henryk Fúks. Solution of the density classification problem with two cellular automata rules. *Phys. Rev. E*, 55:R2081–R2084, 1997.
5 Jarkko Kari. Basic concepts of cellular automata. In Rozenberg et al. [10], pages 3–24.
6 Mark WS Land and Richard K Belew. No two-state CA for density classification exists. *Physical Review Letters*, 74(25):5148–5150, 1995.
7 Jacques Mazoyer. Computations on one-dimensional cellular automata. *Annals of Mathematics and Artificial Intelligence*, 16(1):285–309, 1996.
8 Jacques Mazoyer and Véronique Terrier. Signals in one-dimensional cellular automata. *TCS*, 217(1):53–80, 1999.
9 John Myhill. The converse of Moore's garden-of-eden theorem. In *Proceedings of the American Mathematical Society*, volume 14, pages 658–686, 1963.
10 Grzegorz Rozenberg, Thomas Bäck, and Joost N. Kok, editors. *Handbook of Natural Computing*. Springer, 2012.
11 Véronique Terrier. Language recognition by cellular automata. In Rozenberg et al. [10], pages 124–158.