Theoretical Computer Science 559 (2014) 3–19

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Computational complexity of threshold automata networks under different updating schemes

Eric Goles [a,1], Pedro Montealegre [b,*,2]

[a] Facultad de Ciencias y Tecnología, Universidad Adolfo Ibáñez, Santiago, Chile
[b] Univ. Orléans, INSA Centre Val de Loire, LIFO EA 4022, FR-45067 Orléans, France

ARTICLE INFO

Article history:
Received 11 July 2013
Received in revised form 13 August 2014
Accepted 11 September 2014
Available online 16 September 2014

Keywords:
Automata networks
Threshold functions
Computational complexity
Updating scheme
P-completeness
NC
NP-Hard

ABSTRACT

Given a threshold automata network, as well as an updating scheme over its vertices, we study the computational complexity associated with the prediction of the future state of a vertex. More precisely, we analyze two classes of local functions: the majority and the AND–OR rule (vertices take the AND or the OR logic functions over the state of its neighborhoods). Depending on the updating scheme, we determine the complexity class (**NC**, **P**, **NP**, **PSPACE**) where the prediction problem belongs.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

An *automata network*, is a dynamical system over a connected undirected graph, where at each vertex is assigned a *state* that evolves at discrete time steps accordingly to a *vertex function*, that depends on the current state of the vertex and the states of their neighbors in the graph. The different snapshots of states in graph, obtained by the evolution of each vertex from an initial configuration, model a wide number of biological and physical structures, and social behavior [1–4]. However, those phenomena are not always precisely modeled with a synchronous execution of the local function between each vertex, and it is appropriate to consider different ways to *update* the automata network.

An *updating scheme* is a total order over the set of vertices, such that at each time step, vertices that are first in this scheme evolve before the others. Updating schemes are classified in three groups: *synchronous*, *sequential* and *block-sequential*. A synchronous updating scheme means that every vertex evolves in parallel. Sequential updating schemes are the other extreme: no two vertices are updated at the same time. The block-sequential updating schemes are an intermediate situation by considering a partition of the set of vertices such that when a partition is updated it considers the new states on partitions previously updated.

A natural problem in automata networks is *prediction*: given an initial configuration and an updating scheme, to predict the future states. One possible strategy is to simulate the evolution of each vertex step by step; since the automata network

* Corresponding author.
E-mail addresses: eric.chacc@uai.cl (E. Goles), pedro.montealegre@etu.univ-orleans.fr (P. Montealegre).

[1] This work was partially supported by FONDECYT 1140090 and BASAL-CMM.
[2] Research founded by Becas Chile scholarship.

http://dx.doi.org/10.1016/j.tcs.2014.09.010
0304-3975/© 2014 Elsevier B.V. All rights reserved.

is finite, this strategy will eventually output a solution. A straightforward question then is if this solution is *efficient*, i.e. if there exists better solutions, or if this strategy terminates in *reasonable* time.

In [5,6] is defined a monotone functional associated with threshold automata networks for synchronous and sequential updating schemes. This functional guarantees constant bounded *limit cycles* and polynomial bounded *transient lengths*. This means that for synchronous and sequential updating schemes, the strategy of simulating the evolution of the automata network finish in polynomial time. Unfortunately, no such a functional has been proposed for arbitrary block-sequential updating schemes.

In this paper we show that, in general, unless the well known conjecture of computational complexity theory, $\mathbf{P} \neq \mathbf{NC}$ is false, the simulation strategy is the best possible for synchronous and sequential updating scheme. Moreover, we show that unless $\mathbf{P} = \mathbf{NP}$, no monotone functional for block-sequential updating scheme is possible, and the simulation strategy can take super-polynomial time. We define a decision problem, called PRED, which corresponds to the prediction of a state change in a single vertex, given initial states of an automata network with a given updating scheme. For synchronous and sequential updating schemes, we show that when we restrict to threshold automata networks (in particular *majority automata networks*), the problem PREDis **P**-Complete even when each vertex has at most 3 neighbors. On the other hand, for block-sequential updating schemes, we show that PRED is **NP**-Hard, by designing an automata network that exhibits limit cycles of super-polynomial period.

2. Preliminaries

Let $G = (V, E)$ be a simple connected undirected graph, where $V = \{1, ..., n\}$ is the set of vertices, E the set of edges. An *Automata Network* is a triple $\mathcal{A} = (G, \{0, 1\}, (f_i : i \in V))$, where, $\{0, 1\}$ is the set of *states* and $f_i : \{0, 1\}^{|V|} \to \{0, 1\}$ is the *vertex function* associated with the vertex i. Vertices in state 1 are called *active* while vertices in state 0 are *passive*. We say that the vertex functions are the *rule* of $\mathcal{A}$. The set $\{0, 1\}^{|V|}$ is called the set of *configurations*.

In this paper, every local rule is a particular case of a *threshold function*, formally, let $A = (a_{ij})$ be a real $n \times n$ matrix, and b be the real threshold vector, then:

$$f_i(x) = \begin{cases} 1 & \text{if } \sum_j a_{ij} x_j > b_i, \\ 0 & \text{otherwise.} \end{cases}$$

Let $N(i) = \{j \in V : ij \in E\}$ be the set of neighbors of vertex i. The *majority rule* is obtained taking A as the *adjacency matrix* of the automata network, and $b_i = |N(i)|/2$. Formally, given $i \in \{1, \ldots, n\}$ the vertex function is:

$$f_i(x) = \begin{cases} 1 & \text{if} \sum_{j \in N(i)} x_j > \frac{|N(i)|}{2}, \\ 0 & \text{if} \sum_{j \in N(i)} x_j \leq \frac{|N(i)|}{2}, \end{cases}$$

which means that a vertex takes the state of the majority of its neighbors. When the thresholds are 0 for every vertex, the resulting function is the OR *rule*, and when a vertex becomes active if and only if every neighbor is active (for example the threshold of vertex i is $|N(i)| - 0.5$), the resulting function is called the AND *rule*. When some vertices have a function OR and others AND, the resulting global function is called an AND–OR *rule*.

When the sum is taken over the closed neighborhood $N[i] = N(i) \cup \{i\}$ we say that the function is a *closed threshold function*, and in such a case we have the *closed majority rule*. Another class of rules we will study is that of *frozen threshold functions*, i.e. a vertex at state 1 remains fixed at such state. Formally, this can be encoded by taking A to be the adjacency matrix of the automata network, plus a diagonal of value $a_{ii} = |N(i)|/2 + 1$.

An *updating scheme* of an automata network $\mathcal{A}$ is a function $\mathcal{S} : V \to \{1, \ldots, |V|\}$ such that if u and v are vertices and $\mathcal{S}(u) < \mathcal{S}(v)$ then the state of u is updated before v, and if $\mathcal{S}(u) = \mathcal{S}(v)$ then nodes u and v are updated at the same time. When $\mathcal{S}(v) = 1$ for every vertex v, i.e. all vertices are updated at the same time, we have the *synchronous* updating scheme. When $\mathcal{S}(v) = \sigma_v$, where σ is a permutation of the set of vertices, we have a *sequential* updating scheme. A *block sequential* updating scheme is one where the vertex set is partitioned into several subsets, such that the sets are updated one after the other, in a prescribed order, and elements inside the same set are updated synchronously.

An automata network and updating scheme $\mathcal{S}$ define a *global transition function* $F_{\mathcal{S}} : \{0, 1\}^{|V|} \to \{0, 1\}^{|V|}$ as follows: Let F_k be the function that updates the vertices in turn k according to $\mathcal{S}$:

$$F_k(x)_i = \begin{cases} f_i(x) & \text{if } \mathcal{S}(x) = k, \\ x_i & \text{otherwise;} \end{cases} \qquad \text{then } F_{\mathcal{S}} = F_n \circ F_{n-1} \circ \ldots \circ F_1.$$

The iterated application of the global transition function on a given initial configuration generate a dynamic in the hypercube. For instance, Fig. 1 shows three updating schemes for a graph of four vertices and the majority rule. Notice that for the same initial configuration we can obtain different dynamics.

Given an automata network and an updating scheme $\mathcal{S}$, the *trajectory* of a configuration x is the set $T_{\mathcal{S}}(x) = \{x_{\mathcal{S}}(t) : t \geq 0\}$ where $x_{\mathcal{S}}(0) = x$ and $x_{\mathcal{S}}(t + 1) = F_{\mathcal{S}}(x_{\mathcal{S}}(t))$. When there is no ambiguity, we omit the subindex $\mathcal{S}$. We say that the trajectory of x *enters in a limit cycle of period* p if for some t the set $\{x(t), x(t + 1), \ldots, x(t + p - 1)\}$ has size p, and $x(t + p) = x(t)$; in other words if $|T(x(t))| = p$. A cycle of period 1 is a *fixed point*. Naturally, since the graph has a finite

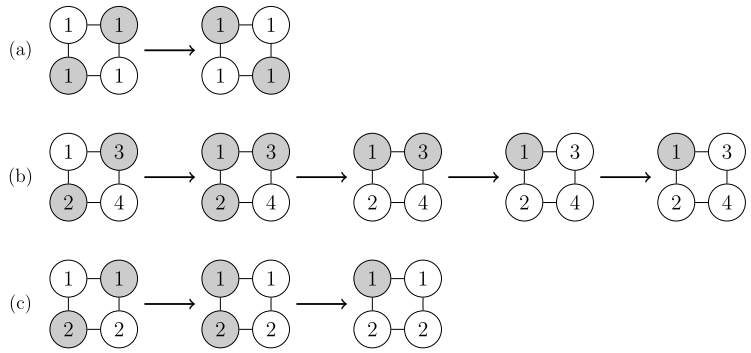


Fig. 1. One step of the a majority automata network in a same initial configuration under different updating schemes: synchronous (a), sequential (b) and block-sequential (c). Grey vertices are active while the white ones are passive. The number represent the position of the vertex in the total order defined by the updating scheme.

number of vertices, there are $2^{|V|}$ different configurations and then, for every initial configuration x and any updating scheme, every trajectory eventually reaches a limit cycle. The set of configurations in the limit cycle is called an *attractor*.

We define the transient length of a configuration x under the updating scheme $\mathcal{S}$ as the number of steps required to reach, for the first time, a configuration in an attractor, and denote this by $\tau_{\mathcal{S}}(x)$. We define the transient length of the automata network $\mathcal{A}$ under the updating scheme $\mathcal{S}$ as the greatest transient length between all possible configurations, that is,

$$\tau_{\mathcal{S}}(\mathcal{A}) = \max\left\{\tau_{\mathcal{S}}(x) : x \in \{0,1\}^{|V|}\right\}.$$

Suppose that we would like to make predictions about the attractor associated with a configuration x, of an automata network $\mathcal{A}$ with updating scheme $\mathcal{S}$. Clearly a solution is to simulate the evolution of each vertex until we reach a limit cycle. A straightforward question is for the existence of better solutions, this is, some algebraic or algorithmic properties that allow us to make predictions, e.g. the size of the attractors, or which vertices change their initial states.

In this context, the following decision problem is defined, which consists in *one vertex change prediction*:

> PRED: Let $\mathcal{A} = (G, \{0,1\}, (f_i : i \in V))$ be an automata network, $\mathcal{S}$ be an updating scheme, $x \in \{0,1\}^{|V|}$ be a configuration of $\mathcal{A}$, and $v \in V(G)$ a vertex initially passive ($x_v = 0$). Does there exists $y \in T_{\mathcal{S}}(x)$ such that $y_v = 1$?

We let PRED-(RULE NAME) be the restriction of PRED to some particular rule, for example PRED-MAJORITY is the restriction of PRED to majority automata networks.

The computational complexity of a decision problem is defined as the amount of resources (time or space) required to give an answer. The classical complexity theory considers the following fundamental classes: **P**, the class of problems solvable by a Turing machine in polynomial time; **NC**, the class of problems solvable in poly-logarithmic time with a polynomial amount of processors in a parallel architecture, for instance a PRAM machine [7]; **NP**, the class of problems solvable by a nondeterministic Turing machine in polynomial time, **PSPACE** the class of problems solvable by a Turing machine that uses polynomial space. It is easy to prove that $\mathbf{NC} \subset \mathbf{P} \subset \mathbf{NP}$ [7,8]. Informally **NC** is known as the class of problems which have a fast parallel algorithm [9], **P** is the class of problems with a *feasible* solution, and **NP** the class of problems where it is feasible to verify a given solution.

It is a well known conjecture that $\mathbf{P} \neq \mathbf{NP}$, where the most likely to not belong to **P** are the **NP**-Complete problems, for example SAT is **NP**-Complete [8]. Similarly is conjectured that $\mathbf{NC} \neq \mathbf{P}$, and if so, there exist "inherently sequential" problems, that belong to **P** and do not belong to **NC**. The most likely to be inherently sequential are **P**-Complete problems, to which any other problem in **P** can be reduced by an **NC** reduction.[3] If any of these problems have a fast parallel algorithm, then $\mathbf{P} = \mathbf{NC}$ [9,10].

One **P**-Complete problem is the *Circuit Value Problem* (CVP), which consists in predicting the truth value of the output of a Boolean circuit. This problem is **P**-Complete since any deterministic Turing machine computation of length k can be converted into a Boolean circuit of depth k; a complete analysis of this reduction can be found in [9].

Given a circuit, we define the *layer* of a gate g, denoted *layer*(g), as follows: it is zero for the input gates, and for the rest is the length of a longest path from an input to g. A circuit is *synchronous* if all inputs to a gate g come from gates at precedent layer. Furthermore, we require that all output vertices be on the same layer, namely the highest [9]. A circuit is

[3] Often logarithmic space reductions are used to prove **P**-Completeness. Since $\mathbf{L} \subset \mathbf{NC}$, any **P**-Complete problem for logarithmic space reductions is **P**-Complete for **NC** reductions.

monotone if there are no NOT gates (only AND and OR gates); it is *alternating* when the gates alternate between OR and AND gates layer by layer, and the inputs are connected only to OR gates, and the outputs are OR gates.

The CVP remains **P**-Complete when the circuit is restricted to be synchronous, monotone, alternating and all vertices have in degree (fan in) and out degree (fan out) exactly two, with the obvious exceptions of the input with in degree zero, and the outputs with out degree zero [9]. We call AS2MCVP this restriction of the CVP.

2.1. Previous results

Threshold automata networks, also called *neural networks*, have been widely studied [5,6,11,12]. In [6] Goles et al. give a characterization of the attractors and transient of such networks through the use of an operator, analogous to the *spin glass energy* [11]. It is shown that for the synchronous updating scheme, when the interconnection matrix A is symmetric then the attractors are only fixed points or cycles of period two. Further, if the diagonal elements of the matrix are non-negative ($diag(A) \geq \vec{0}$) then the sequential update admits only fixed points. If A is the adjacency matrix of the automata network (i.e. $a_{ij} = 1$ if $j \in N(i)$; 0 otherwise) the transient lengths are bounded by $\mathcal{O}(n^2)$, where n is the size of the graph. On the other hand, consider a symmetric matrix A and a block-sequential updating scheme $\mathcal{B}$. Let $\mathcal{B}_k = \{i : \mathcal{B}(i) = k\}$, $n_k = |\mathcal{B}_k|$, and $A_k = (a_{ij})_{i,j \in \mathcal{B}_k}$. If A_k is non-negative definite over the set of vectors in $\{-1, 0, 1\}^{n_k}$, for every k, then the limit sets are only fixed points, and the transient length is bounded by n^2.

Also in the context of block-sequential updating schemes, in [13] closed threshold automata networks are studied. Recall that in the matrix that define those automata networks every diagonal entry is 1. Using different arguments than the energy approach and non-negative matrices above, they prove that when the largest block size is at most 3, closed threshold automata networks admit only fixed points as attractors (clearly for partitions of size one and two, the fixed point behavior is direct from the non-negativity of such matrices with $diag(A) = \vec{1}$). Further, they exhibit two-cycles when at least a partition with four vertices is considered and they conjecture that two is the maximum period for any block sequential updating on closed threshold automata networks. We will disprove this conjecture as a corollary of Theorem 4 in Section 4.

In terms of complexity, in [10] Moore studies the majority cellular automata in the d dimensional lattice. He proves that PRED-MAJORITY restricted to synchronous updating schemes is **P**-Complete for $d \geq 3$, while the complexity remains open for $d = 2$. In [14] is studied the *frozen version* (vertices in state 1 remain fixed) of the majority automata network, also known as *Bootstrap percolation model*, and it is shown that for synchronous updating schemes, PRED-FROZEN is **P**-Complete on the family of graphs with maximum degree equals 5 and it is in NC for maximum degree ≤ 4.

2.2. Contributions of the paper

We study first, in Section 3, the *frozen threshold automata networks*. Clearly, for those automata, the only possible attractors are fixed points. Moreover, we show that for any initial configuration, and for any pair of different updating schemes, the automata network reaches the same attractor. We conclude, by the results in [14] that PRED-FROZEN MAJORITY is **P**-Complete.

In Section 4, we analyze the complexity of the majority automata networks as a particular case of the threshold automata. Recall that the majority rule means that the vertices are active if and only if the strict majority of their neighbors is active. We first use the results of [6], and appropriate gadgets to show that PRED-MAJORITY is **P**-Complete for synchronous and sequential updating schemes. Moreover, this remains true when we restrict the underlying network to the family of graphs with maximum degree 3.

On the other hand, for block-sequential updating schemes, and still considering the majority rule, we show that we may construct majority automata networks with limit cycles of arbitrary periods. Using these structures in an appropriate way, we build a majority automata network with a block-sequential updating scheme (blocks of cardinality 2) admitting limit cycles with super-polynomial period in the size of the network.

The possibility of large periods suggests, unlike the synchronous and sequential cases, that it is not possible to have a monotone functional associated with a majority rule for block-sequential updating schemes. We justify this claim showing that PRED-MAJORITY is **NP**-Hard. As said in the previous sub-section, the results for majority automata networks can be translated to closed majority automata networks.

In Section 5 we study other particular threshold automata networks, the one with the OR and the AND rules. First we show that PRED–OR and PRED–AND are in **NC**. A more interesting case occurs when we have the AND–OR rule. Using again the results of [6] and appropriate gadgets, we show that PRED–AND–OR is **P**-Complete for synchronous and sequential updating schemes. In the case of block-sequential updating schemes for the AND–OR rule, we show that there are AND–OR automata networks that admit limit cycles with super-polynomial period. Finally, we show that in a restricted family of the initial configurations, and for synchronous update scheme PRED–AND–OR is in **NC**.

3. Frozen threshold automata networks

As we defined above, a *frozen threshold automata network*, is a threshold automata network such that active vertices remain always active. Formally, a frozen threshold automata network is an automata network $\mathcal{A} = (G, \{0, 1\}, (f_i : i \in V))$, where the vertex function is defined as follows:

$$f_i(x) = \begin{cases} 1 & \text{if } x_i = 1, \\ 1 & \text{if } \sum_{j\in N(i)} x_j > \theta_i \wedge x_i = 0, \\ 0 & \text{if } \sum_{j\in N(i)} x_j \leq \theta_i \wedge x_i = 0. \end{cases}$$

Notice that every vertex state may change at most one time, and then in at most $|V(G)|$ steps the trajectory reaches a fixed point for any initial configuration and any updating scheme. We will prove that the complexity of Pred-frozen is independent of the updating scheme, by showing that for a given configuration, the fixed point is the same for any updating scheme.

Lemma 1. *Let $\mathcal{A} = (G, \{0,1\}, (f_i : i \in V))$ be a frozen threshold automata network, $\mathcal{U}$ be an arbitrary updating scheme and $\mathcal{S}$ be the synchronous one. Let x be any configuration of $\mathcal{A}$. Then, the trajectory of x reaches the same fixed point for both updating schemes $\mathcal{U}$ and $\mathcal{S}$.*

Proof. Let $\{x_{\mathcal{U}}(t), t \geq 0\}$ and $\{x_{\mathcal{S}}(t), t \geq 0\}$ be the trajectories of x for the updating schemes $\mathcal{U}$ and $\mathcal{S}$ respectively. Notice that since the active state is frozen, $x_{\mathcal{U}}(t) \leq x_{\mathcal{U}}(t+1)$ and $x(t)_{\mathcal{S}} \leq x_{\mathcal{S}}(t+1)$, for any $t \geq 0$. Let M be the number of blocks in $\mathcal{U}$. The proof follows from observing that for all $t \geq 0$,

$$x_{\mathcal{S}}(t) \leq x_{\mathcal{U}}(t) \leq x_{\mathcal{S}}(Mt). \tag{1}$$

Indeed, the first inequality follows from the fact that the vertices that are updated later in $\mathcal{U}$ have at least the same number of active neighbors than in the synchronous updating scheme. On other hand, updating synchronously M times the whole network will give at least the same number of active vertices as when updating with $\mathcal{U}$.

Let $\bar{x}_{\mathcal{U}}$ and $\bar{x}_{\mathcal{S}}$ be the fixed points for the updating schemes $\mathcal{U}$ and $\mathcal{S}$ respectively. Let $T \geq 0$ be the minimum value such that $x_{\mathcal{S}}(T) = \bar{x}_{\mathcal{S}}$ and $x_{\mathcal{U}}(T) = \bar{x}_{\mathcal{U}}$. If follows from (1) that $x_{\mathcal{S}}(T) \leq x_{\mathcal{U}}(T) \leq x_{\mathcal{S}}(MT)$ then $\bar{x}_{\mathcal{S}} = \bar{x}_{\mathcal{U}}$. □

The following theorem follows from Lemma 1 and the results of [14]:

Theorem 2.

- Pred-frozen-majority *is **P**-Complete when the underlying graph of the automata network is restricted to the family of graphs with maximum degree* 5.
- Pred-frozen-majority *is in **NC** when the underlying graph of the automata network is restricted to the family of graphs with maximum degree* 4.

4. Majority automata networks

The *majority automata network* $\mathcal{A} = (G, \{0,1\}, (f_i : i \in V))$ is the one where a passive vertex is active if the strict majority of their neighbors is active, formally a majority automata network is defined by the following local functions:

$$f_i(x) = \begin{cases} 1 & \text{if } \sum_{j\in N(i)} x_j > \frac{|N(i)|}{2}, \\ 0 & \text{if } \sum_{j\in N(i)} x_j \leq \frac{|N(i)|}{2}. \end{cases}$$

We will first study the synchronous and sequential updating schemes. Later, we dedicate a sub-section to block-sequential updating schemes. Finally, we will show how the results for majority automata networks can be extended to other thresholds.

4.1. Synchronous and sequential updating schemes

From [6], we know that restricted to synchronous and sequential updating schemes Pred-majority is in **P**. Indeed, since the transient length is bounded by $\mathcal{O}(n^2)$ and the limit cycles are always of constant length (at most cycles of length 2 for the synchronous updating scheme, and only fixed points for any sequential updating scheme), then to predict state changes of a single vertex can be done by simulating the evolution of the automata network under the updating scheme. Since each step can be simulated in $\mathcal{O}(n^2)$, in at most $\mathcal{O}(n^4)$ time the simulation stops and the algorithm terminates.

Unlike the *frozen* case, Pred-majority is **P**-Complete not only when we restrict to graphs with maximum degree 4, but also for graphs with vertices of degree 3 (*cubic graphs*):

Theorem 3. Pred-majority *is **P**-Complete for synchronous and sequential updating schemes and when the underlying graph of the automata network is restricted to the family of cubic graphs.*

Proof. We will first prove the **P**-Completeness for the synchronous case. Let (C, I, o) be an instance of the As2mcvp, this is, C is an alternating, synchronous, fan-in 2, fan-out 2 and monotone Boolean circuit; I is a truth value assignment of its

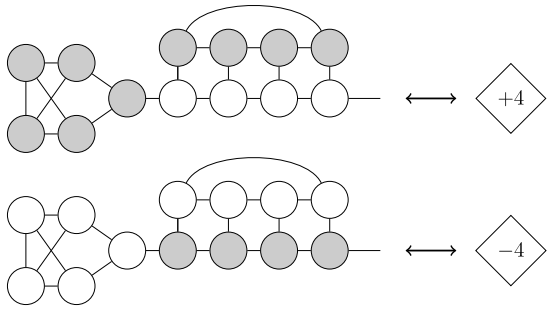


Fig. 2. Positive and negative clocks of length 4. Clocks are represented with a diamond shaped node, where the number in the diamond represents the length of the clock and the sign represents if the clock is positive or negative.

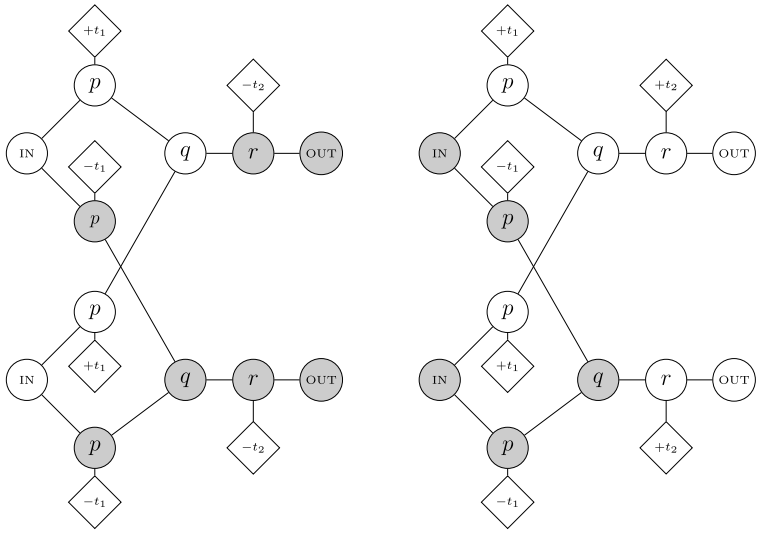


Fig. 3. An OR (left) and an AND (right) gate of layer k. In the figure $t_1 = 4k - 4$ and $t_2 = 4k - 2$. The letter in the vertices are identifiers, the updating scheme is synchronous.

inputs; and o an output. We will simulate (C, I, o) with a majority automata network $\mathcal{A}$ over a cubic graph G, this is, every vertex have three neighbors. The goal is to do this using $\mathcal{O}(\log(n))$ space, where n is the size (number of gates) of the circuit.

The first gadget that we will use are the ones shown in Fig. 2, which we call *positive* or *negative clocks*. A positive clock of length k is a path of k initially passive vertices, with one end connected to a cycle of active vertices, such that before k steps all become active. The idea is to attach a positive clock of length k to an initially passive vertex, and then this vertex will have a passive neighbor for the first k steps, and then an active one. A negative clock is like a positive clock but swapping between active and passive vertices.

Using positive and negative clocks, we can build gadgets that can simulate the gates of the circuit. Recall that C is a synchronous alternating circuit, which means that the gates are ordered by layers, which alternate between OR and AND gates. Input gates are simply initially active or passive vertices according to the truth value of the input gate, this is, input gates *true* in I will be active, and the rest will be passive.

In Fig. 3 are represented the OR and AND gates as if they were in layer k. We overlap the OUT gates of the gadgets to the corresponding IN gates of the following layer, according to the adjacency given by C.

For any gadget of layer k, both active and passive vertices remain in their state for the first $4k - 4$ steps. At step $4k - 4$, the IN vertices change their states, and the clocks of length $4k - 4$ will stop, making their adjacent vertices p take the state of the corresponding IN vertex in the following step $(4k - 3)$. At step $4k - 2$, the following vertices, labeled q, will simulate the logic functions OR or AND, depending on the gadget, and the clocks of length $4k - 2$ will stop, making the vertices labeled r take the state of q in step $4k - 1$. Finally, at step $4k$ the OUT vertices take the state of the r vertices (notice that the OR vertices are IN vertices in the next layer, and then have both an active and a passive neighbor). Vertices that represent input gates are connected with clocks of length 0, positive or negative depending on I. This is done to maintain the degree 3.

Let L be the maximum layer. Recall that we named o the output gate of the circuit on which we make the decision for AS2MCVP, and let g_o be the gadget that simulates o according to Fig. 3. We overlap both OUT vertices of g_o with the IN

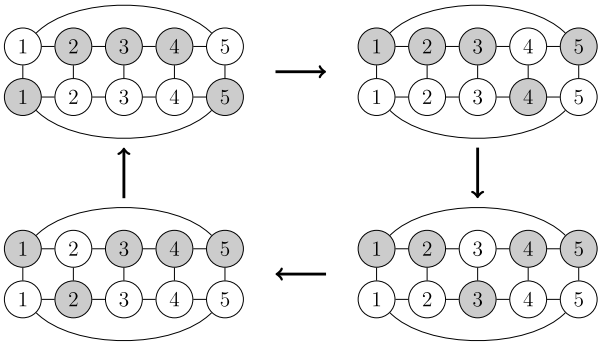


Fig. 4. A *ladder* of length 4. Vertex numbers represent its value in the updating scheme. A limit cycle of period 4 is obtained.

vertices of a new gadget g_v that simulates an AND Êgate in layer $L+1$, and name v one of the OUT vertices of g_v. Finally, to maintain the 3-regularity, we connect to each OUT vertex of v and of each gadget simulating gates in layer L (except o), to two clocks, one positive and one negative, of length 0. In other words, we connect the last OUT vertices to a fixed active and a fixed passive vertices.

Clearly by the clocks connected to the vertices labeled r, it is impossible for a vertex OUT of a gadget of a gate in layer k to change states in any step before $4k$. Notice also that if the OUT vertices of g_o are not active at step $4L$, then v will never become active. Indeed, suppose that the IN vertices of g_v are passive at step $4L$. Since the clocks connected to the p vertices of g_v (see Fig. 3) will stop at step $4L$, then the four vertices labeled p will become passive at step $4L+1$, and then the two vertices labeled q will become passive at step $4L+2$, making impossible to v to become active in any step grater that $4(L+1)$. We obtain that v becomes active if and only if the OUT of g_o are active at step $4L$. We conclude that v becomes active if and only if o is true for the truth assignment I in C.

Since every layer has the same number of gates, the maximum layer is $n/L = \mathcal{O}(n)$, where n is the number of gates in the circuit. From all this we see that the gadgets, in particular the clocks, can be constructed in $\mathcal{O}(\log(n))$ space, since one just have to remember the layer of the simulated gadget ($\mathcal{O}(\log(n))$ space), and the numbers of the input and outputs of the gate ($\mathcal{O}(\log(n))$ space each).

In a similar way we can build gadgets for the asynchronous case: First, we replace the clocks in Fig. 3 by clocks of length 0, and keeping the sign of the clock. Notice that a positive (resp. negative) clock of length 0 is a 3-cycle of active (resp. passive) vertices. Second, we take the following sequential updating scheme: vertices in gadgets that simulate gates in layer k are updated before the ones that simulate gates in layer $k+1$. Vertices in the same gadget are updated from left to right, according Fig. 3, where two parallel vertices can be updated in arbitrary order. Two vertices in gadgets simulating different gates of the same layer can be updated in any order. The construction of the gadgets require the same space, and the construction of the updating scheme can be done in $\mathcal{O}(\log(n))$ since we just need to remember the corresponding layer. □

4.2. Block sequential updating schemes

The fact that PRED-MAJORITY belongs to **P** for the synchronous and asynchronous updates seems to be no longer true for arbitrary block-sequential updating schemes. Actually, we prove in the next theorem that blocks of size 2 in a block-sequential updating scheme allow limit cycles of super-polynomial period. This suggest that there is no possible monotone *energy* functional for block-sequential updating schemes, analogous to the one exhibited in [6].

Theorem 4. *There is a block sequential update scheme in a majority automata network, such that each block has cardinality* 2 *and the limit cycle has super polynomial period.*

Proof. A *ladder* of length k (Fig. 4) is a graph with $2(k+1)$ vertices, which can be updated in a particular updating scheme in order to obtain a limit cycle of period k.

One can connect two ladders as shown in Fig. 5, and then obtain that the limit cycle is the least common multiple (lcm) of the lengths of the two ladders.

To obtain a lower bound of the periods, we follow the arguments of [15]. Let m be a positive integer, and let $\pi(m)$ be the number of primes not exceeding m, and let $\{p_1, p_2, \ldots, p_{\pi(m)}\}$ be the first $\pi(m)$ primes. Let G be the graph obtained from $\pi(m)$ ladders of sizes $p_1, p_2, \ldots, p_{\pi(m)}$, connected one after the other following the structure shown in Fig. 5. We obtain:

$$V(G) \leq \sum_{i=1}^{\pi(m)} 2(p_i+1) \leq 2\pi(m)(m+1) \tag{2}$$

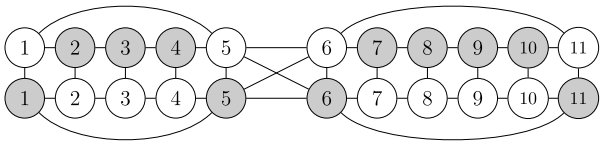


Fig. 5. A *ladder* of length 4 connected with a ladder of length 5. The limit cycle of this graph is $\text{lcm}(4, 5) = 20$.

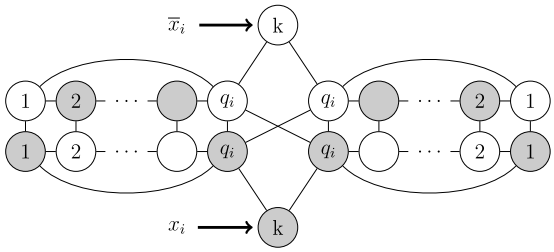


Fig. 6. Variable x_i. Gray vertices represent active ones, and vertex numbers represent its value for the updating scheme. In the figure $q_i = p_i + 1$, and then the ladder make a cycle of length p_i. Also in the figure $k = p_n + 2$, and then the vertices that simulate x_i and $\bar{x}_i$ are updated after every ladder.

and

$$\text{lcm}(p_1, \ldots, p_{\pi(m)}) = \prod_{i=1}^{\pi(m)} p_i = e^{\theta(m)} \tag{3}$$

where $\theta(m) = \sum_{i=1}^{\pi(m)} \log(p_i)$. From the Prime Number Theorem [16] we know that $\pi(m) = \Theta(m/\log(m))$, furthermore in [16] is shown that $\theta(m) = \Theta(\pi(m)\log(m))$, which together with (2) and (3) imply that

$$\text{lcm}(p_1, \ldots, p_{\pi(m)}) \geq e^{\Omega(\sqrt{|V(G)|\log(|V(G)|)})}$$

and then the length of the limit cycle of G is not bounded by any polynomial in $|V(G)|$. □

Theorem 5. PRED-MAJORITY *is* ***NP****-Hard.*

Proof. We will show that PRED is **NP**-Hard by reducing 3-SAT to it. Remember that 3-SAT is the Boolean satisfiability problem restricted to instances with exactly three variables per clause [8]. This problem is **NP**-Complete, and we will show that it can be reduced to PRED with the majority automata and a block sequential updating scheme.

Let ϕ be an instance of 3-SAT with n variables and m clauses. Let $x_1, \ldots, x_n$ be the n variables of ϕ, and $p_1, \ldots, p_n$ be the first n primes. For each literal of a variable (either the variable or the negation of a variable) of each clause, we will build a gadget for that variable (see Fig. 6).

The gadget for variable x_i consists of two ladders of size p_i, connected to two vertices, that will represent the *positive* and *negative* literals, say x_k and $\bar{x}_k$. The ladders will produce cycles of length p_i, and then vertex representing x_i will be active in steps multiple of p_i, and passive *otherwise*, in the same way the vertex representing $\bar{x}_i$ will be passive in steps multiple of p_i, and active otherwise. We interpret the state of this vertex as the truth value of variable x_i.

With this construction, we can go over every combination of input values of ϕ. For example, if $n = 5$ (i.e. ϕ have 5 variables) then the input combination $(x_1, x_2, x_3, x_4, x_5) = (1, 0, 1, 0, 1)$ will occur after $p_1 \times p_3 \times p_5 = 2 \times 5 \times 11 = 110$ steps. From the Prime Number Theorem [16], we have that $p_n = \mathcal{O}(n\log(n))$. Since the input is of size $\mathcal{O}(n)$, then finding the first n primes can be done in polynomial time, and then in polynomial time we can build the gadgets for each variable of ϕ.

The reduction then consists of 3 layers: The first layer will have the gadget for simulating the variables (one different for each appearance of the variable in a clause). In the second layer we simulate every clause by joining the literals with a node that will simulate the OR function (see Fig. 7).

Finally, we connect every clause gadget to a vertex v that simulates the AND function, that can be built using an initially passive vertex joined with $m - 1$ initially passive vertices (m is the number of clauses). In Appendix A there is an example of the reduction from a formula with three variables and two clauses. We conclude that v becomes active at some step if every vertex that simulates an OR function becomes active at the same step, and this happens if and only if there exists a truth assignment of the variables that satisfy ϕ. We conclude that PRED-MAJORITY is **NP**-Hard. □

4.3. Closed majority rule and other threshold automata networks

An important remark is that all the results of last section are valid for closed majority automata networks. Recall that the closed majority rule is like the majority, but also considers the state of the vertex. The ladders in the proof of Theorem 4 may

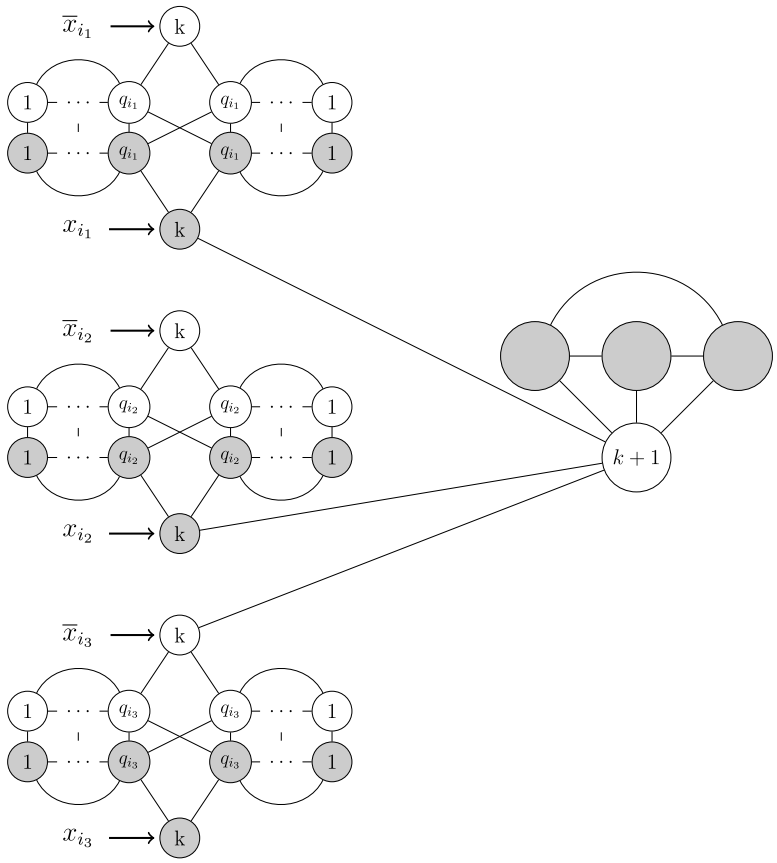


Fig. 7. Clause $C_i = (\bar{x}_{i_1} \vee \bar{x}_{i_2} \vee x_{i_3})$. In the figure $q_i = p_i + 1$ and $k = p_n + 2$, where p_i is the i-th prime. Gray vertices represent active ones, and the numbers in the vertices are the value of the vertex in the updating scheme. Vertices without number can be updated in an arbitrary order.

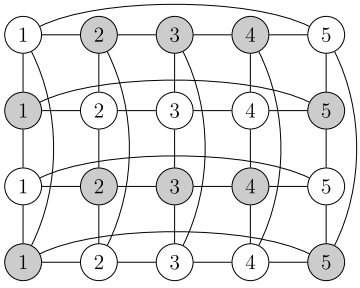


Fig. 8. A *ladder* of length 4 for the closed majority rule.

be easily modified as seen in Fig. 8. The remainder arguments follows as the non-closed case, which disproves the conjecture proposed in [13], since there are closed threshold automata networks that exhibit limit cycles with periods greater that 2. Actually, following the arguments of the non-closed case, we conclude that closed threshold automata networks can exhibit super-polynomial limit cycles.

Notice that in this case, blocks of size 4 are required. This complements the result of [13], where it is shown that closed threshold automata networks and block sequential updating schemes with maximum blocks of size 3 only exhibit fixed points.

5. AND–OR automata networks

Let us consider the automata network $\mathcal{A} = (G, \{0, 1\}, (f_i : i \in V))$ such that the local function is the AND rule:

$$\text{AND}_i(x) = \begin{cases} 1 & \text{if } \forall j \in N(i) x_j = 1 \\ 0 & \text{if } \exists j \in N(i) x_j = 0 \end{cases}$$

We call this automata the AND automata network. Similarly, we define the OR automata network as the one where the vertex function is the logic OR:

$$\text{OR}_i(x) = \begin{cases} 1 & \text{if } \exists j \in N(i) x_j = 1, \\ 0 & \text{if } \forall j \in N(i) x_j = 0. \end{cases}$$

Clearly both cases correspond to threshold automata networks. In the case of the AND rule, the threshold of each vertex is $|N(i)| - 1$ while for the OR rule, the threshold is 0 for any vertex. Moreover, we define the AND–OR automata network as a threshold automata where each local rule is either an AND or an OR function. For an AND–OR automata network $\mathcal{A}$, we define the sets $S_{\textbf{And}} = \{i \in V : f_i = \text{AND}_i\}$, $S_{\textbf{Or}} = \{i \in V : f_i = \text{OR}_i\}$. In this context, we interpret the AND (resp. OR) rule as the AND–OR rule when $S_{\textbf{And}} = \emptyset$ (resp. $S_{\textbf{Or}} = \emptyset$).

Notice that the problem PRED–AND belongs to **NC**. Indeed, let $(\mathcal{A}, \mathcal{S}, x, v)$ be an instance of PRED–AND. By definition of the problem, $x_v = 0$, and then the only possibility for v to become 1 is to have every neighbor initially active and updated after v. This can verified in $\mathcal{O}(\log(n))$ time assigning one processor to each vertex i neighbor of v, and answer 1 if $\mathcal{S}(i) \geq \mathcal{S}(u)$ and $x_i = 1$. After that another $\mathcal{O}(n)$ processors can check if every neighbor of v answered 1 in $\mathcal{O}(\log(n))$ using a prefix sum algorithm [7]. This is also true for OR automata networks.

Theorem 6. PRED–OR *is in* ***NC***.

Proof. Let $(\mathcal{A} = (G, \{0, 1\}, \text{OR}_i), \mathcal{S}, x, v)$ be an instance of PRED–OR. Evidently, if every vertex is initially passive, the answer of PRED–OR will be *false*. Moreover, if two adjacent vertices are initially active, in at most $n = |V(G)|$ steps every vertex will become active in a fixed point. Suppose then that there is at least one initially active vertex, but no adjacent vertices are active.

Let A be the adjacency matrix of G, and suppose that $\mathcal{S}$ is the synchronous updating scheme. Clearly in this case $x_i(t) = 1$ if $(Ax(t-1))_i > 0$. We say that a graph G is *primitive* if its adjacency matrix satisfies $A^k > 0$ for some $k < n^3$. In [17] is shown that any undirected connected non-bipartite graph G is primitive. Remember that an automata network is always defined over a connected graph. If G is non-bipartite, since $x \neq 0$, in at most n^3 steps every vertex becomes active. Suppose then that G is bipartite, with partitions V_1 and V_2. If there are initially active vertices in both partitions, in at most n steps two adjacent vertices will become active, and then eventually the trajectory will reaches a fixed point with each vertex active. Suppose then that every initially active vertex belong to the same partition, say V_1. In this case, in at most n the trajectory will reach a limit cycle of period 2, where in even steps every vertex in V_1 is active, while vertices in V_2 are passive, and the inverse for odd steps.

Suppose now that $\mathcal{S}$ is not the synchronous updating scheme. Remember that we are in the case where no pair of adjacent vertices are initially active. If every initially active vertex is updated before their neighbors, then in the first step every vertex in the graph will be passive, and then the trajectory will enter into a fixed point where everyone is passive. On the other hand, if there is an initially active vertex with a neighbor that updates before it, then after the first step both vertices will be active, and in at most n steps every vertex will be active in a fixed point. Finally, in the case where, for every initially active vertex u, there exists $w \in N(u)$ such that $\mathcal{S}(w) = \mathcal{S}(u)$. Let C_u be the connected component of G that contains u, where $\forall w_1 \in C_u, \mathcal{S}(w_1) = \mathcal{S}(u)$, and pick $w_2 \in N(C_u) \cap (V \backslash C_u)$. We know that after at most $|C_u|$ steps every vertex in C_u will become active at least once, and then in at most $|C_u + 1|$ steps w_2 will become active. Since $\mathcal{S}(w_2) \neq \mathcal{S}(u)$, there will be a step where w_2 and some vertex in C_u will be active, and then in at most n mores steps, the trajectory will reach a fixed point where each vertex is active. In brief:

- If every vertex is initially passive, or there is no pair of adjacent initially active vertices and every initially active vertex is updated before their neighbors, then the dynamic enters into a fixed point where every vertex is passive.
- If $\mathcal{S}$ is the synchronous updating scheme, the graph is bipartite and every initially active vertex belong to the same partition of G, then the attractor is a cycle of period 2.
- Otherwise, the attractor is a fixed point where each vertex is active.

Notice that in the previous analysis, if a vertex is not initially active and at some step becomes active, then it is not possible that in the steady state (fixed point or cycle of length 2) this vertex is always passive. We define then Algorithm 1 (see Appendix A), which will take as input an adjacency matrix associated with $G = (V, E)$, a configuration x of G as a vector of length $|V(G)|$ where the i-th component is the initial state of vertex i, and an updating scheme $\mathcal{S}$ as a vector of length $|V(G)|$ where the i-th component is k if $\mathcal{S}(i) = k$. The output of the algorithm is a vector S where S_i is 1 if i is active in the attractor, 0 if i is passive, *even* if i is active only in even steps, and *odd* if i is active only in odd steps.

Every step of this algorithm can be done in $\mathcal{O}(\log^2(n))$ time and $\mathcal{O}(n^2)$ processors. Indeed, steps (1), (4), (10), (13), (16), (19), (23) and (28) can be done in $\mathcal{O}(\log(n))$ time with $\mathcal{O}(n)$ processors. Steps (2), (3), (6), (9), (12), (15) and (22) can be done in $\mathcal{O}(\log(n))$ time and $\mathcal{O}(n^2)$ processors just doing a prefix sum algorithm [7]. Finally, steps (7) and (8) can be done in $\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n^2)$ processors. We start calculating a spanning tree of G, taking an arbitrary vertex as a root. Then we calculate the level (distance to the root) of every vertex of the tree. Finally, the vertex set is partitioned into two sets depending on the parity of the level. We check if every edge connects vertices of different sets. Finding the spanning tree

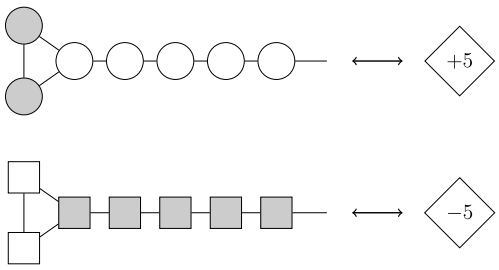


Fig. 9. Positive and negative clocks of length 5, squared nodes represent vertices in $S_{\mathbf{And}}$ while circular nodes represent vertices in $S_{\mathbf{Or}}$. Clocks are represented with a diamond shaped node, where the number in the diamond represents the length of the clock and the sign represents if the clock is positive or negative.

of G can be done in $\mathcal{O}(\log^2 n)$ time using $\mathcal{O}(n^2)$ processors [7]. Calculating the level of the vertices of a tree takes $\mathcal{O}(\log n)$ time and require $\mathcal{O}(n)$ processors [18]. Everything else can be done in $\mathcal{O}(\log(n))$ time with $\mathcal{O}(n^2)$ processors. □

A much more complex case is the combination of the Or and the And rules, this is, And–Or rule. Indeed, unless $\mathbf{P} = \mathbf{NC}$, Pred–And–Or is *inherently sequential* for synchronous updating scheme.

Theorem 7. Pred–and–or *is* ***P****-Complete for the synchronous updating scheme.*

Proof. Remember that a problem is **P**-Complete if it belongs to **P** and is **P**-Hard. From the results of [6] and the same arguments used for the majority rule, we have the membership in **P**. We will follow the same approach that we did for Theorem 3. Let (C, I, o) be an instance of As2mcvp, that is, C is an alternating, synchronous, fan-in 2, fan-out 2, monotone circuit, I a truth value of the inputs of C, and o an output of C.

We will simulate an instance of As2mcvp by an instance of Pred–and–or, where some vertices will represent the gates of the circuit, and the states of this vertices will represent the truth value of the simulated gates. Even when the functions of the vertices are similar to logical functions, the simulation of the gates is not trivial, since we have to simulate a directed graph with an undirected graph. For example a vertex u with function $\mathbf{and}_u$ need every neighbor (say, the inputs and the outputs) to be active to become active.

To solve this issue, consider the gadgets shown in Fig. 9, which are called *positive and negative clocks*. A positive clock of length k is a path of k passive vertices in $S_{\mathbf{Or}}$ where one of its ends have two connected active vertices and the other is connected to a passive vertex, and then after k steps every vertex of the path will become active. In a similar fashion, we define a negative clock of length k as a path of k active vertices in $S_{\mathbf{And}}$, where one of its ends have two connected passive vertices and the other end is connected to a active vertex. Then after k steps every vertex in the path will become passive. We will use positive clocks of length k to keep vertices in $S_{\mathbf{And}}$ passive for at least k steps, and negative clocks of length k to keep vertices in $S_{\mathbf{Or}}$ active for at least k steps.

Using these clocks, we can simulate the gates of C by a graph G with configuration x and rule And–Or defined as follows (remember that the circuit is alternating, in even layers every gate is of type Or, and in the odd layers there are only gates of type And):

- The input gates of C are simulated by a vertex in $S_{\mathbf{Or}}$. Those who simulate *true* gates according to I are active in x;
- A gate of on even layer k is simulated by a vertex in $S_{\mathbf{Or}}$, initially active and connected to a negative clock of length $k-1$;
- A gate of an odd layer k is simulated by a vertex in $S_{\mathbf{And}}$, initially passive and connected to a positive clock of length $k-1$;
- Finally, we connect the vertex that simulates the output gate o to a vertex v with function $S_{\mathbf{And}}$ and a positive clock of length L (remember that we denote by L the maximum layer). Define this vertex as the one that we pick to decide Pred–And–Or.

By constructions of the clocks and definitions of functions And_i and Or_i, a vertex in layer k will be in its initial state for the first $k-1$ steps. After that, the future state of a vertex in layer k will depend on the states of the vertices in layer $k-1$ (and not in the vertices of layer $k+1$). For example, a vertex which simulates a gate And in layer k will be passive for the first $k-1$ steps, since the neighbor corresponding to the clock gadget will be passive, even when all the other neighbors (Or gadgets) are active. In step $k-1$ simultaneously the clock stops (the neighbor of the gate becomes active), the vertices in layer $k-1$ change their states according to the circuit simulation, while the vertices of layer $k+1$ remain active due to their clocks.

Inductively, we can repeat this argument for every step and every layer, obtaining that the vertex that simulates the output of the circuit o will be active at step L (where L is the value of the maximum layer) if and only if o is true for the truth assignment I in C. Finally, the vertex that simulates o is connected to a vertex v in $S_{\mathbf{And}}$, active at step $L+1$ if and

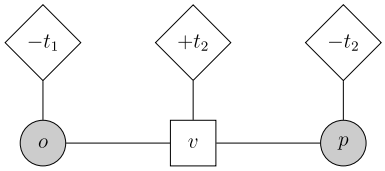


Fig. 10. Output of the gadget. In the figure $t_1 = L - 1$ and $t_2 = L$. The decision of Pred–And–Or is going to be taken over vertex v, while vertex o is the output of the As2mcvp instance.

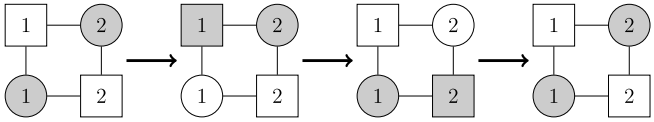


Fig. 11. Cycle gadget for the And–Or rule. Squared nodes represent vertices in $S_{\mathbf{And}}$ while circular nodes represent vertices in $S_{\mathbf{Or}}$. Grey vertices are active, and the number represent the value of the vertex in the updating scheme. A cycle gadget reaches a limit cycle of period 3.

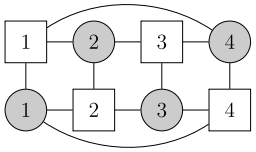


Fig. 12. Two joint cycle gadgets. This one reaches a limit cycle of length 7.

only if the vertex that simulates o is active at step L. If it does not become active at step $L + 1$, the vertex p in Fig. 10 will become passive, forcing v to stay passive.

This reduction use only logarithmic space since we only have to remember the pointers to the outputs and inputs (size $\mathcal{O}(\log(n))$), the type of the gate (And, Or) (size $\mathcal{O}(1)$), and the layer (size $\mathcal{O}(\log(n))$). Then the And–Or rule with a synchronous updating scheme is **P**-Complete. □

Notice that if we delete the clocks and define a sequential updating scheme where the vertices in layer k are updated after the vertices in layer $k - 1$ and before the ones in layer $k + 1$, starting from layer 1, we obtain the same result for the And–Or rule and the set of sequential updating schemes.

Corollary 1. Pred–and–or *is* ***P****-Complete for sequential updating schemes.*

Moreover, we have shown that not fixing the updating scheme the problem is **P**-Hard, but in this case the membership in **P** is unknown.

Corollary 2. Pred–and–or *is* ***P****-Hard.*

Like what happens with the majority automata networks, the And–Or automata networks can have large (super-polynomial) limit cycles when we accept block sequential update schemes. To establish that, let us consider the following *cycle gadget* given in Fig. 11, where a cycle of period 3 is obtained as attractor.

If we join two cycle gadgets, as shown in Fig. 12 we obtain a cycle of length 7. It is easy to prove that if we follow this construction joining k groups of cycle gadgets, which sums $4k$ vertices, we can obtain a cycle of length $4(k - 1) + 3$ (see Appendix A for a proof of this fact). That gives us a way to construct large cycles by using prime numbers as elementary cycles connected each to a pair of central nodes, as we establish in next theorem.

Theorem 8. *There is a block sequential update scheme in a* And–Or *automata network, such that each block has cardinality* 2 *and the limit cycle has super-polynomial period.*

Proof. We follow again the arguments developed in [15], as we have done for Theorem 4. In this case we cannot make cycles of prime length for every prime number, since the cycles for this case have length of the form $4k + 3$. From the well known Dirichlet's Theorem, we have that there are infinitely many primes in the arithmetic progression $\{4k + 3\}_{k\in\mathbb{N}}$. Let $\pi(m)$ be the number of primes in the progression $\{4k + 3\}_{k\in\mathbb{N}}$ that are lower than m, and $\theta(m) = \sum_{i=1}^{\pi(m)} \log(p_i)$. By the results in [19] we have that $\pi(m) = \Theta(m/\log(m))$ and $\theta(m) = \Theta(\pi(m)\log(m))$. The proof follows from the use of these bounds and similar constructions as those given in Theorem 4. □

Let $\mathcal{A}$ be an AND–OR automata network, and let X be the set of configurations of $\mathcal{A}$ such that $\forall x \in X$, $\{v \in V \mid x_v = 1\} \subset S_{\mathbf{Or}}$, that is, for any configuration $x \in X$, all active vertex in x are in $S_{\mathbf{Or}}$, and there are no initially active vertices in $S_{\mathbf{And}}$. Notice that even with this restriction we can build the cycle gadgets, and the **P**-Completeness reduction for the sequential updating scheme. However, when we restrict the input configuration to X, we are unable to build negative clocks, in the **P**-Completeness proof for the synchronous case. Moreover, unless $\mathbf{P} = \mathbf{NC}$, no possible reduction to AS2MCVP is possible.

Theorem 9. PRED–AND–OR *is in* ***NC*** *for synchronous updating schemes and when the input configuration belongs to* X.

Proof. Let $(\mathcal{A}, \mathcal{S}, x, v)$ be an instance of PRED–AND–OR, where $\mathcal{A}$ is the AND–OR automata network, $\mathcal{S}$ is the synchronous updating scheme and x is a configuration in the set X, this is, every vertex in $S_{\mathbf{And}}$ is initially passive.

Notice first that since the input configuration belongs to X, two adjacent vertices in $S_{\mathbf{And}}$ will stay passive. Then we will be only interested in $S_{\mathbf{And}}$ vertices with only $S_{\mathbf{Or}}$ neighbors. Let $G[S_{\mathbf{Or}}]$ be the subgraph of G induced by the set $S_{\mathbf{Or}}$, and for $u \in S_{\mathbf{Or}}$, let C_u be the connected component of u in $G[S_{\mathbf{Or}}]$. We will apply Algorithm 1 (of the proof of Theorem 6) to every connected component of $G[S_{\mathbf{Or}}]$, obtaining an array S, where S_u is $0, 1, even, odd$ depending on the state of u in the attractor of x restricted to $G[S_{\mathbf{Or}}]$. In the following we will analyze how the states in the attractor of x restricted of $G[S_{\mathbf{Or}}]$ can be transferred to G.

- Evidently, vertices $u \in S_{\mathbf{Or}}$ such that $S_u = 1$ will be always active in the attractor of x.
- If $S_u = 0$, then there are two options: (1) every vertex in C_u is initially passive or (2) u is initially active but isolated in $G[S_{\mathbf{Or}}]$. In the first case, every vertex in C_u will be also passive in the attractor of x. However, in case (2), there exists another possibility. Let u be a vertex in $S_{\mathbf{Or}}$ such that $S_u = 0$ and u is isolated in $G[S_{\mathbf{Or}}]$. Since G is connected, the set $N(u)$ is a subset of $S_{\mathbf{And}}$, and then necessarily in step 1 u becomes passive ($x_u(1) = 1$). If any neighbor w of u satisfies $x_p = 1, \forall p \in N(w)$, that is, every neighbor of w is initially active, then in that case $x_w(1) = 1$, and then $x_u(2) = 1$, and u begin to blink between passive and active states in even-odd steps. Otherwise, if any neighbor of u becomes active at the first step, then u will be passive in the attractor.
- If u is a vertex such that $S_u = odd$, then u must have a neighbor $w \in C_u$ with $S_w = even$ (it is analogous if $S_u = even$). Clearly any neighbor of u that does not belong to C_u can, in the better case, become active only in even steps, and then the state of u in the attractor will be the same as in $G[S_{\mathbf{Or}}]$.

Then, to decide PRED–AND–OR, we first calculate the connected components of $G[S_{\mathbf{Or}}]$, and execute Algorithm 1 on every connected component. After that, we check if any initially active and isolated vertex in $G[S_{\mathbf{Or}}]$ has any neighbor $w \in S_{\mathbf{And}}$ such that $x_p = 1, \forall p \in N(w)$. We then obtain a vector S such that $S_u = 1, 0, odd, even$ depending on the attractor of $u \in S_{\mathbf{Or}}$. If the vertex v (the one for which we decide PRED–AND–OR) belongs to $S_{\mathbf{Or}}$ we just check if $S_u \neq 0$. If the vertex v belongs to $S_{\mathbf{And}}$, we check if every vertex of v is in $S_{\mathbf{Or}}$, if it is the case, we check that $S_u \neq 0$ and also that there is no pair of neighbors $u_1, u_2 \in N(v)$ such that $S_{u_1} = odd$ and $S_{u_1} = even$.

We obtain then Algorithm 2, which can be seen in Appendix B. This algorithm can be executed in $\mathcal{O}(\log^2(n))$ time using $\mathcal{O}(n^3)$ processors: Steps (1) can be done with time $\mathcal{O}(\log^2(n))$ using $\mathcal{O}(n^2)$ processors [7], step (2) require, by the analysis of Algorithm 1 $\mathcal{O}(\log^2(n))$ time using $\mathcal{O}(n^2)$ processors, steps (3)–(10) require $\mathcal{O}(\log(n))$ time using $\mathcal{O}(n^3)$ processors, n processors for each **do in parallel** and n more for step 6, and using a prefix sum algorithm [7]. Steps (15) and (16) can be done each in time $\mathcal{O}(\log(n))$ with $\mathcal{O}(n^2)$ processors using a prefix sum algorithm, and the rest can be done in time $\mathcal{O}(1)$ with $\mathcal{O}(1)$ processors.

Using this algorithm we can decide PRED–AND–OR in time $\mathcal{O}(\log^2(n))$ using $\mathcal{O}(n^3)$ processors for the case where the initial configuration x is in the set X, and we conclude the membership in **NC**. □

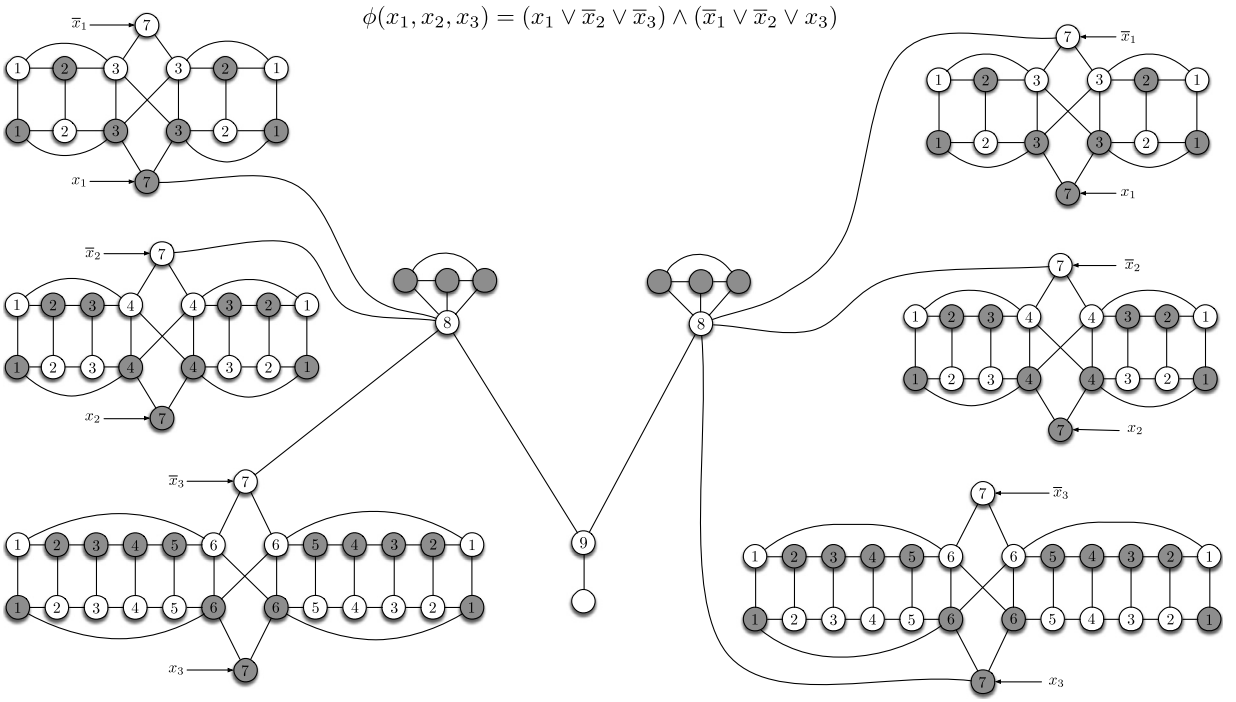
6. Conclusion

We have discussed the computational complexity of threshold automata networks, and how it is affected by changes in the updating scheme. We have proved first that the complexity of frozen threshold automata networks is invariant over this kind of change. Then, we have shown that this is not the case for the complexity of threshold automata network. While for the synchronous and sequential updating scheme PRED is **P**-Complete, for the block sequential updating schemes we have shown that the problem is **NP**-Hard.

We conjecture that PRED-MAJORITY is **PSPACE**-Complete. Moreover, we conjecture that the same is true for the AND–OR rule. If this is true, the restriction of inputs configurations to the set X (only vertices with function OR can be initially active) separate the complexities of the three classes of updating schemes. Indeed, restricting the input configurations to belong to the set X, PRED–AND–OR is in **NC** for synchronous updating scheme, **P**-Complete for sequential updating schemes, and (we conjecture) **PSPACE**-Complete for block sequential updating schemes.

Finally, a notion of complexity can be defined for automata networks and updating schemes. If we say that the complexity of an automata is the complexity of the problem PRED restricted to this automata, then an automata have a *portable* complexity if the complexity of PRED does not depend on the updating scheme. For example, frozen threshold automata networks are *portable* ***P****-Complete*.

Appendix A. Example of NP-Hardness proof and Algorithms 1, 2



Algorithm 1 ORS.

Require: A matrix A of dimensions $n \times n$, which corresponds to the adjacency matrix of an undirected graph $G = (V, E)$. An array $\mathcal{S}$ of length n which represent the updating scheme, an array x of dimension n which represent the initial configuration, and an integer v which corresponds to the index of a vertex of G given by A.

Ensure: An array S of length n where $S_u = 0, 1, even, odd$ depending on the state of u in the attractor the evolution of the OR automata network defined by G, with updating scheme $\mathcal{S}$ and initial configuration x.

Define $S_i = 0\ \forall i \in \{1, \ldots, n\}$
if there is an initially active vertex **then**
  if there is a pair of adjacent initially active vertex **then**
    $S_i = 1\ \forall i \in \{1, \ldots, n\}$, Return S.
  else
    if $\mathcal{S}$ is the synchronous updating scheme **then**
      if G is bipartite **then**
        Define V_1, V_2 the partitions of G.
        if every initially active vertex belongs to V_1 **then**
          $S_i = even\ \forall i \in V_1$ and $S_i = odd\ \forall i \in V_2$, Return S.
        end if
        if every initially active vertex belongs to V_2 **then**
          $S_i = odd\ \forall i \in V_1$ and $S_i = even\ \forall i \in V_2$, Return S.
        end if
        if there is an initially active vertex in V_1 and another in V_2 **then**
          $S_i = 1\ \forall i \in \{1, \ldots, n\}$, Return S.
        end if
      else
        $S_i = 1\ \forall i \in \{1, \ldots, n\}$, Return S.
      end if
    else
      if there exists $i, j \in \{1, \ldots, n\}$ with $a_{ij} = 1$ (this means that $j \in N(i)$), $x_i = 1$ and $\mathcal{S}(j) < \mathcal{S}(i)$ **then**
        $S_i = 1\ \forall i \in \{1, \ldots, n\}$, Return S.
      end if
    end if
  end if
else
  Return S.
end if

Algorithm 2 ANDS-ORS.

Require: A matrix A of dimensions $n \times n$, which corresponds to the adjacency matrix of an undirected graph $G = (V, E)$. An array T of dimension n such that $T_u = 0$ if vertex $u \in S_{\mathbf{And}}$ and $T_u = 1$ if u is in $S_{\mathbf{Or}}$. An array x of dimension n which represent the initial configuration, and an integer v which corresponds to the index of a vertex of G given by A.

Ensure: Accept if $(\mathcal{A} = (G, \{0, 1\}, F), \mathcal{S}, x, v) \in$ Pred–And–Or, where F is the And–Or rule defined by vector T and $\mathcal{S}$ is the synchronous updating scheme. Reject otherwise.

Calculate the connected components of $G[S_{\mathbf{Or}}]$ using the connected components algorithm of [7].
Execute in every component of $G[ORS]$ Algorithm 1 and save the output in an array S.
for each $i \in \{1, \ldots, n\}$, **do in parallel**
  if $T_i = 1$ $(i \in S_{\mathbf{Or}})$, $S_i = 0$ and $x_i = 1$, **then**
    for each $j \in \{1, \ldots, n\}$ such that $a_{ij} = 1$ **do in parallel**
      if $x_k = 1$ for all $k \in \{1, \ldots, n\}$ such that $x_{jk} = 1$ **then**
        $S_i = 1$.
      end if
    end for
  end if
end for
if $v \in S_{\mathbf{Or}}$ **then**
  Accept if $S_v \neq 0$, and reject if $S_v = 0$.
else
  if $N(v) \subset S_{\mathbf{Or}}$ **then**
    if $\forall i \in N(v)$, $S_i \in \{1, even\}$ or $\forall i \in N(v)$, $S_i \in \{1, odd\}$ **then**
      Accept.
    else
      Reject.
    end if
  else
    Reject.
  end if
end if

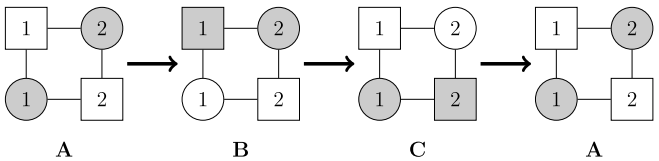


Fig. B.13. Cycle gadget for the And–Or rule. Squared nodes represent vertices in $S_{\mathbf{And}}$ while circular nodes represent vertices in $S_{\mathbf{Or}}$. Grey vertices are active, and the number represent the value of the vertex in the updating scheme. A cycle gadget reaches a limit cycle of period 3, and we represent steps with letters **A, B** and **C**.

Appendix B. Proof of the limit cycles of period $4k - 1$ of k joined cycle gadgets of the And–Or-Rule

A cycle gadget is a cycle of 4 vertices, 2 initially active with function Or and two initially passive with function And. The dynamics are shown in Fig. B.13, where each step of the trajectory of this gadget is represented with letters **A, B** and **C**.

We join k cycle gadgets as shown in Fig. B.14, by connecting consecutive And and Or vertices, and then complete a cycle by connecting the first pair of And and Or vertices with the last pair.

We can represent the configurations of k joined cycle gadgets using the representation using letters **A, B** and **C**. Consider the trajectory of two cycle gadgets, shown in Fig. B.15.

In letter notation, the trajectory is represented as:

Step	0	1	2	3	4	5	6	7
Configuration	**AA**	**BB**	**AC**	**CA**	**AB**	**BA**	**CC**	**AA**

Let $\mathcal{C}$ be any one-dimensional cellular automata with three states **A, B** and **C**, and with transition function that satisfies:

AAA	**ABA**	**ABB**	**ACA**	**ACC**
B	**C**	**B**	**A**	**A**
	BAA	**BAB**	**BBB**	
	B	**B**	**A**	
	CAA	**CAC**	**CCC**	
	C	**C**	**A**	

Then the trajectory of k joined cycle gadgets can be simulated by $\mathcal{C}$ on input **A...A** (k times), with updating scheme from left to right and periodic boundary conditions. For example, simulating the dynamics of five joined cycle gadgets, we obtain:

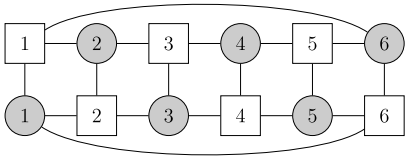


Fig. B.14. Three joined cycle gadgets – we represent this configuration as **AAA**.

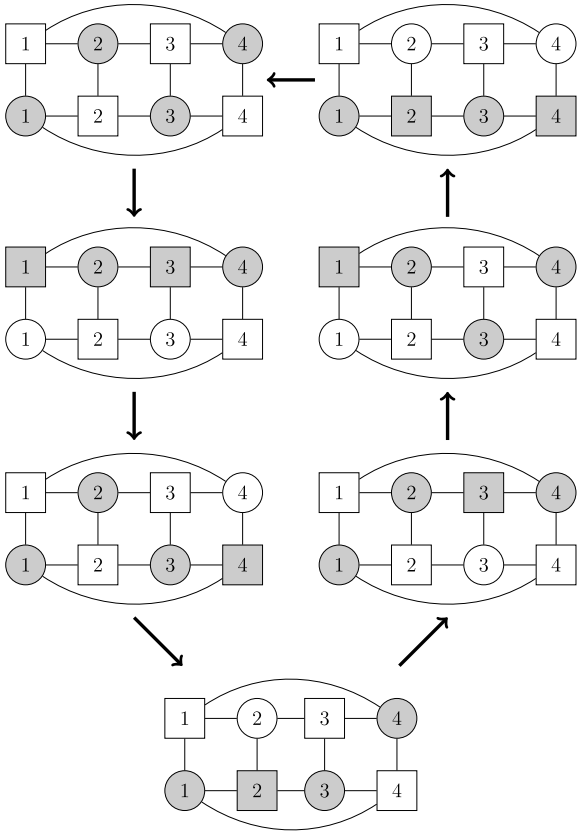


Fig. B.15. Dynamics of two cycle gadgets. Here we obtain a cycle of length 7.

AAAAA
BBBBB
AAAAC
CCCCA
AAAAB
BBBBA
AAACC
CCCAA
AAABB
BBBAA
AACCC
CCAAA
AABBB
BBAAA
ACCCC
CAAAA
ABBBB
BAAAA
CCCCC
AAAAA

This is the same result as if we apply $\mathcal{C}$ to the input **AAAAA**.

The trajectory of k joined cycle gadgets, follows the following pattern, when it is represented in letters: the following block of 4 steps is repeated k times, where the repetition p is:

(1) $\mathbf{A}^{k-(p-1)}\mathbf{B}^{p-1}$,
(2) $\mathbf{B}^{k-(p-1)}\mathbf{A}^{p-1}$,
(3) $\mathbf{A}^{k-p}\mathbf{C}^{p}$,
(4) $\mathbf{C}^{k-p}\mathbf{A}^{p}$.

Is easy to check, using the rules of $\mathcal{C}$, that for every $p \in \{1, \ldots, k\}$, the update of (1) leads to (2), (2) to (3) and (3) to (4). Moreover, by induction it is easy to prove that the configuration (4) of repetition p leads to configuration (1) of repetition $p+1$. Since the first and the last configuration are equal ($\mathbf{A}^{k-(p-1)}\mathbf{B}^{p-1}$ when $p=1$ is equal to $\mathbf{C}^{k-p}\mathbf{A}^{p}$ when $p=k$, both equal to $\mathbf{A}^{k}$), and then the limit cycle reached by k joined cycle gadgets has period $4k-1$.

References

[1] C. Castellano, S. Fortunato, V. Loreto, Statistical physics of social dynamics, Rev. Modern Phys. 81 (2009) 591–646.
[2] N. Goles Domic, E. Goles, S. Rica, Dynamics and complexity of the Schelling segregation model, Phys. Rev. E 83 (2011) 056111.
[3] S. Bornholdt, Boolean network models of cellular regulation: prospects and limitations, J. R. Soc. Interface 5 (Suppl. 1) (2008) S85–S94.
[4] M.I. Davidich, S. Bornholdt, Boolean network model predicts cell cycle sequence of fission yeast, PLoS ONE 3 (2) (2008) e1672.
[5] E. Goles-Chacc, Comportement oscillatoire d'une famille d'automates cellulaires non uniformes, Université scientifique et médicale de Grenoble, Institut national polytechnique de Grenoble, 1980.
[6] E. Goles-Chacc, F. Fogelman-Soulie, D. Pellegrin, Decreasing energy functions as a tool for studying threshold networks, Discrete Appl. Math. 12 (3) (1985) 261–277.
[7] J. Jájá, An Introduction to Parallel Algorithms, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.
[8] S. Arora, B. Barak, Computational Complexity: A Modern Approach, 1st edition, Cambridge University Press, New York, NY, USA, 2009.
[9] R. Greenlaw, H. Hoover, W. Ruzzo, Limits to Parallel Computation: P-Completeness Theory, Oxford University Press, 1995.
[10] C. Moore, Majority-vote cellular automata, Ising dynamics, and P-completeness, J. Stat. Phys. 88 (1997) 795–805.
[11] J.J. Hopfield, Neural networks and physical systems with emergent collective computational abilities, Proc. Natl. Acad. Sci. USA 79 (8) (1982) 2554–2558.
[12] H.S. Mortveit, C.M. Reidys, An Introduction to Sequential Dynamical Systems, Universitext, Springer, 2008.
[13] H.S. Mortveit, Limit cycle structure for block-sequential threshold systems, in: ACRI, 2012, pp. 672–678.
[14] E. Goles, P. Montealegre-Barba, I. Todinca, The complexity of the bootstraping percolation and other problems, Theoret. Comput. Sci. 504 (2013) 73–82.
[15] M.A. Kiwi, R. Ndoundam, M. Tchuente, E. Goles, No polynomial bound for the period of the parallel chip firing game on graphs, Theoret. Comput. Sci. 136 (2) (1994) 527–532.
[16] G.H. Hardy, E.M. Wright, D.R. Heath-Brown, J. Silverman, An Introduction to the Theory of Numbers, Oxford Mathematics, OUP, Oxford, 2008.
[17] R.A. Brualdi, H.J. Ryser, Combinatorial matrix theory, in: Encyclopedia of Mathematics and Its Applications, Cambridge University Press, 1991.
[18] U. Vishkin, On efficient parallel strong orientation, Inform. Process. Lett. 20, 235–240.
[19] I. Soprounov, A short proof of the prime number theorem for arithmetic progressions, available online at http://www.math.umass.edu/isoprou/pdf/primes.pdf.