Leader election on two-dimensional periodic cellular automata

Nicolas Bacquey

Leader election on two-dimensional periodic cellular automata

Nicolas Bacquey[a]

[a]*GREYC - Université de Caen Basse-Normandie / ENSICAEN / CNRS*
Campus Côte de Nacre, Boulevard du Maréchal Juin
CS 14032 CAEN cedex 5, France

Abstract

This article explores the computational power of bi-dimensional cellular automata acting on periodical configurations. It extends in some sense the results of a similar paper dedicated to the one-dimensional case. More precisely, we present an algorithm that computes a "minimal pattern network", *i.e.* a minimal pattern and the two translation vectors it can use to tile the entire configuration. This problem is equivalent to the computation of a leader, which is one equivalence class of the cells of the periodical configuration.

Keywords: cellular automata, leader election, bi-periodical configuration, equivalence classes, uniform computation

Introduction

Cellular automata are a well-studied computational model. Its uniform and local properties capture a large range of natural problems, and the model is simple enough to allow definitions of algorithms, or complexity classes. However, the infinite nature of its underlying structure may raise some issues when confronted with finite objects. There are two naive ways of solving these issues: either working on a finite subset of the automaton, or requiring the whole infinite configuration to be periodical.

Periodical configuration is quite a natural concept in the context of tiling problems. As a significant example, it was first (erroneously) conjectured by Hao Wang in [13] that each finite set of tiles that tiles the plane can always do it by some periodical configuration. Periodical configurations have also been extensively studied on cellular automata, when those are considered as dynamical systems: typically, questions of undecidability of the injectivity or reversibility of the transition function have been studied [5]. On the other hand, periodical configurations are much less studied from the point of view of computation, with some notable exceptions, such as the density classification problem [7].

A natural problem would be to compute a "minimal period" of a periodical configuration. However, performing computations on periodical configurations is somehow counterintuitive, because one cannot easily define essential notions, such as the origin and termination of the computation, or the time complexity of an algorithm. These difficulties are mainly due to the fact that unlike what happens on classical models such as Turing machines, you cannot choose a single cell to start the computation or bear its result. This difficulty is overridden when

Email address: nicolas.bacquey@unicaen.fr (Nicolas Bacquey)

the input of a cellular automaton is bounded by persistent symbols, because then you can identify cells that are on the border of the computation area, and use them as starting or stopping points. However, one cannot use such tricks when the configuration is periodical; in that case, all notions related to computation should be global.

This paper presents an extension of a previous work [1] dedicated to the simpler one-dimensional case. In [1] we exhibited a one-dimensional cellular automaton that computes in polynomial time a minimal period of an infinite one-dimensional periodic configuration. We now want to deal with bi-periodical configurations of dimension 2, *i.e.* configurations that have two independent vectors of periodicity. A natural starting point would be to compute the *minimal pattern* of a given configuration, *i.e.* the smallest pattern with which we are able to rebuild the whole configuration by translation along the two orthogonal axis. This problem is unsolvable in the case of cellular automata of dimension ≥ 2, as it is briefly suggested on Fig 1. Instead, we will solve the problem of exhibiting a *minimal pattern network*, *i.e.* a minimal pattern and the two translation vectors it can use to tile the entire configuration.

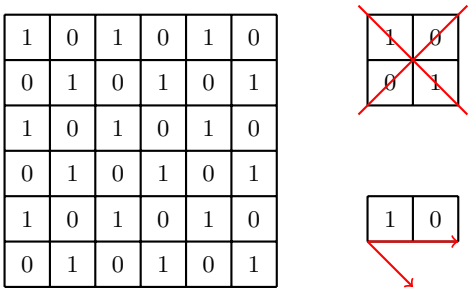


Figure 1: One cannot extract the 2×2 square pattern from the configuration, because all cells marked with 1 are *indistinguishable* from each other. Nevertheless, it is possible to extract the 2×1 pattern, with its two tiling vectors.

We can see that this minimal pattern network can be computed through the election of a leading equivalence class of cells. This leader election will be the main subject of this article. While leader election on two dimensional cellular automata has already been extensively studied [11], all the algorithms that already exist cannot be applied to our model, because they heavily rely on the existence of computational borders, that frame the finite computational area. Due to the periodical nature of the configurations we study, such borders cannot exist in our framework. In that sense, this article presents a new algorithm that performs leader election on a broader class of configurations.

After having clearly defined the problem of leader election on periodical cellular automata, we will present some algorithmic tools that fit our needs. Finally, we will present an algorithm that performs leader election in polynomial time.

1. Context and basic definitions

1.1. The computational model

We will use along this article the standard definition of cellular automata (CA) as a tuple $\mathcal{A} = (d, \boldsymbol{Q}, V, \delta)$ (see [6]). In these lines, we will work with $d = 2$, *i.e.* with cellular automata whose underlying network is $\mathbb{Z}^2$. $\boldsymbol{Q}$ denotes the set of *states*, and V is the standard *Moore neighbourhood* [1]. The local transition function of the automaton is denoted by $\delta : \boldsymbol{Q}^V \to \boldsymbol{Q}$. As we work with cellular automata from the point of view of *language recognition*, we will identify a particular subset $\Sigma \subseteq \boldsymbol{Q}$ as the *input alphabet*. A *configuration* is an application $C : \mathbb{Z}^2 \to \boldsymbol{Q}$. We also introduce the *global transition function* $F_\delta : \boldsymbol{Q}^{\mathbb{Z}^2} \to \boldsymbol{Q}^{\mathbb{Z}^2}$ defined by the global synchronous application of δ over configurations of $\mathbb{Z}^2$.

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 [6] or [9] for such general matters on cellular automata.

Definition 1. *We define a* Toric-Cellular Automaton (toric-CA) *as a cellular automaton whose initial configuration (and therefore any subsequent configuration) is bi-periodic (*i.e. *periodic in two independent directions).*

Note that this model is equivalent to an automaton that would work on a finite, torus-like cell network.

Definition 2. *Let w be a rectangular word over Σ, we denote $C_w \in \boldsymbol{Q}^{\mathbb{Z}}$ the configuration formed by the uniform repetition of w over $\mathbb{Z}^2$.*

Definition 3. *We call a* cell *an element of the underlying network $\mathbb{Z}^2$ and the state of $\boldsymbol{Q}$ associated with it.*

Note that the state of a given cell may change through time.

2. Leader election on toric-CA

We now consider our bi-periodic configurations, and will try to perform leader election upon them. We will see that it is not possible to elect a single cell on the configuration, and will introduce the definition of equivalence classes to deal with that issue.

2.1. Equivalence classes of cells

Let C be a bi-dimensional, bi-periodic configuration over $\boldsymbol{Q}$.

Definition 4. *We say that two cells c_1 and c_2 of C are* equivalent *if the translation that moves c_1 on c_2 leaves the configuration unchanged.*

It is immediate to see that the previous definition induces an *equivalence relation*. Let us now consider the equivalence classes of that relation. As all the cells of an equivalence class have the same state and the same neighbourhood, the application of the transition function on these cells will give the same resulting state. By recurrence, it appears that at any given time during the

[1] $V = \{(-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (0,1), (1,-1), (1,0), (1,1)\}$

computation, each cell of an equivalence class will be in the same state. From a computational point of view, the cells of a given equivalence class are *indistinguishable from each other* (See Fig 2).

Because the configuration C we are considering is bi-periodic, it also appears that each cell belongs to an infinitely large equivalence class, and that there only exists a finite number of equivalence classes.

Definition 5. *We define the* size *of a bi-periodic configuration as the number of its equivalence classes. The size of the initial configuration will be denoted as* N.

It is clear that N is the only pertinent parameter when one wants to discuss time complexity of algorithms running on toric-CA.

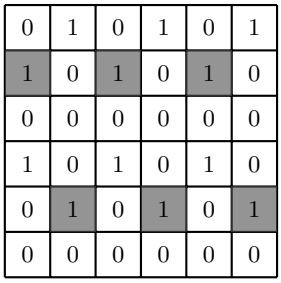


Figure 2: All darkened cells will have the same behaviour during the computation, as they belong to the same equivalence class.

2.2. Proper definition of leader election

As a corollary of previous statements, it appears that it is impossible to elect a single cell of the configuration as a leader (or any finite subset of cells for that matter).

The leader election problem on bi-periodic configurations actually sums up to the *election of a single equivalence class*.

2.3. Main result

In the following sections, we will describe an algorithm that performs leader election on toric-CA.

More precisely, the algorithm will process bi-periodic configurations such that after a certain time:

- The states of all the cells of the elected equivalence class will be in a certain final subset of states $F \in \boldsymbol{Q}$ and will never leave it.
- The states of every other cell will never be in F any more.

Theorem 1. *The algorithm presented in this paper solves the leader election problem in a time polynomial in the size N of the initial configuration.*

At the beginning, all cells will be candidates to this election, then our algorithm will perform a percolation amongst those cells, only sparing those which belong to a single equivalence class. Note that due to the nature of our computational model, it is impossible for a single cell to assert that the election is over and no more cells are to be percolated. This can only be done by an observer *outside* of the model.

3. Basic objects and tools

We will consider *patches* as the basic objects in this article. A patch denotes a finite set of neighbouring cells, upon which computation will be performed. We precisely intend to subdivide the whole configuration into patches, and we want the behaviour of a patch to depend only on its content and the content of its neighbours. Later in the article, we will introduce signals that will travel along the borders of the patches. We want a single signal to be able to travel through the entire border of a patch, which forbids patches to have holes. However, we present in the next section a method to process patches that may have holes.

3.1. Patches, borders and contents

In order to properly define what a patch of cells is, we have to subdivide the cell network. More precisely, each cell of the network will be divided into four sub-cells (See Fig 3). Please note that this rather unusual definition is due to the dynamics of the main algorithm. It is convenient to first define a patch as a simple closed curve, and then as a set of cells.

Definition 6 (patch as a curve)**.** *We define a* patch *as a finite, simple, closed curve in the subdivided cell network that obeys some restrictions, which will be detailed thereafter.*

The restrictions focus on the angles of the curve, and consist of a set of allowed and forbidden angles, such as shown on Fig 4. They can be informally summarized as follows :

- Each outer angle (*i.e.* 90°angle when measured from the inside) must externally correspond to a full line corner.
- Each inner angle (*i.e.* 270°angle when measured from the inside) must internally correspond to a dashed line corner.

Fig 5 gives examples of simple curves of the subdivided network. We can note that the only curves which properly define a patch are those on Fig 5a and Fig 5b, while Fig 5c presents angles that are forbidden.

Definition 7 (border)**.** *The* border *of a patch is canonically defined as the projection of its curve over the corresponding outer cell edges of the original network.*

Definition 8 (patch as a set of cells)**.** *We define the* content *of a patch as the set of cells within its border, or indifferently the initial state of those cells.*

As an abuse of notation, we may use the term *patch* to denote its content.

Examples of patches with their contents and borders are shown on Fig 6. Note that this definition allows patches whose content may contain holes, but whose border can be travelled through by a single signal (see Fig 6b).

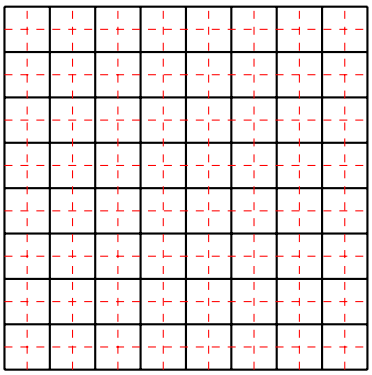


Figure 3: Dividing the cell network

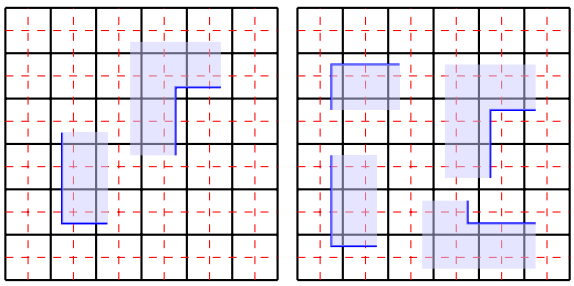
Proper patches. We can see that some parts of the border of a patch might be useless (see Fig 7). The following definition will give us a canonical representative for those patches.

Definition 9 (Proper patch)**.** *We say that a patch is a* proper patch *if the curve defining it does not contain the pattern presented in Fig 7a. The proper patch associated to a given raw patch is the patch where all the occurrences of the forbidden pattern have been locally deleted (see Fig 7). Note that this operation does not change the content of a patch.*

We would like to be able to distinguish a particular cell in a patch; this is the goal of the next definition.

Definition 10 (Patch leading cell)**.** *We define the* leading cell *of a patch as the uppermost cell amongst its leftmost.*

Note that this notion is *a priori* unrelated with the leader election problem we exposed earlier.

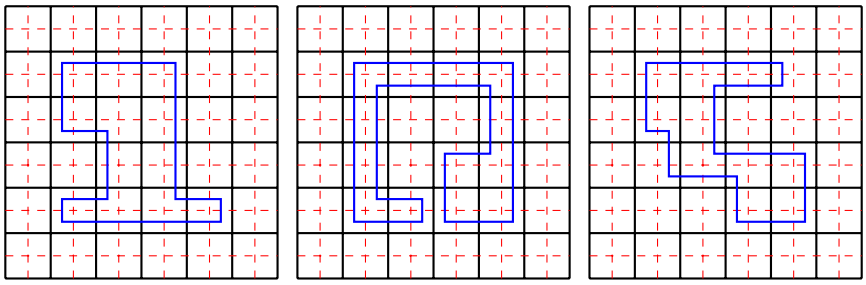


(a) Allowed angles (b) Forbidden angles

Figure 4: Local recognition of acceptable curves (the coloured area denotes the inside of the curve)

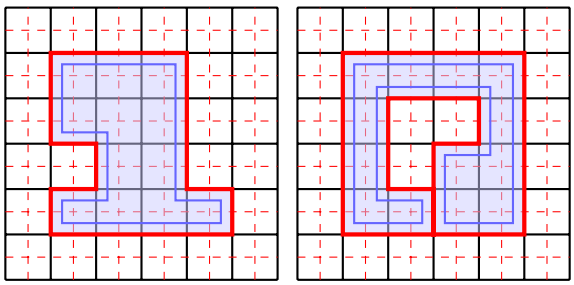
3.2. Patch word

We are going to design an algorithm able to compare patches. Because a patch is a rather complex object, we have to find a canonical representation that



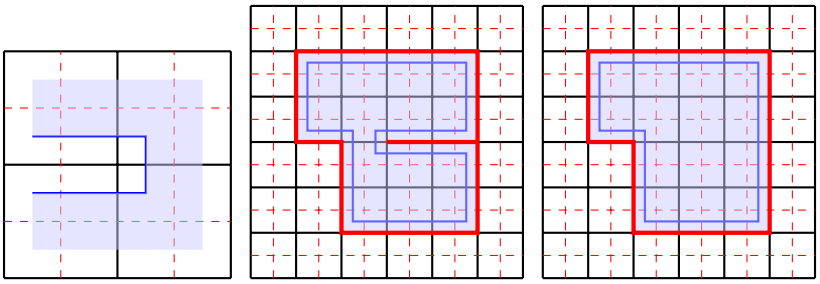
(a) Acceptable patch (b) Acceptable patch (c) Forbidden patch

Figure 5: Examples of simple closed curves that may define patches



(a) Without a hole (b) With a hole

Figure 6: Examples of patches and their borders



(a) Forbidden pattern (b) Non-proper patch (c) Proper patch

Figure 7: Transforming a raw patch into a proper one

our computational model can handle. A patch will be represented by a word encoding both the shape of its border and its content.

Definition 11 (Border word)**.** *We define the* border word *associated to a patch as the word over the alphabet* $\Gamma = \{\uparrow, \downarrow, \leftarrow, \rightarrow\}$ *that represents its border in clockwise order, starting from the upper left angle of the patch leader.*

For instance, the border word of Fig 8 is $\rightarrow\rightarrow\rightarrow\downarrow\downarrow\downarrow\rightarrow\downarrow\leftarrow\leftarrow\leftarrow\leftarrow\uparrow\rightarrow\uparrow\leftarrow\uparrow\uparrow$

Now we must extend our concept of border word to take the content of the patch into account. A trivial solution that first comes to mind is to enumerate

the content of all the cells in reading order, starting from the patch leader. However, this solution raises some issues due to the fact that it is hardly locally computable. We propose instead the construction of a *spanning tree* over the cells of a patch, whose root is the leading cell. We admit for the moment, for the sake of simplicity, that we can easily construct such a spanning tree on a proper patch with local rules, once the leading cell is identified (the proof of this point will be given in section 5.4.3). The spanning trees associated to different shapes of patches can be seen on Fig 9.

Definition 12 (patch word)**.** *Let Σ be the input alphabet of our automaton, we define the* patch word *associated to a proper patch as the word $w_c = w_\Gamma w_\Sigma$, where $w_\Gamma \in \Gamma^*$ is the border word of the patch, and $w_\Sigma \in \Sigma^*$ is the content of the cells of the patch, ordered by the prefix order depth-first search of the spanning tree associated to the patch (for a fixed order of the directions).*

For instance, here follows the patch word of the patch of Fig 8, whose spanning tree is the one on Fig 9a :

$$w_c = \rightarrow\rightarrow\rightarrow\downarrow\downarrow\downarrow\rightarrow\downarrow\leftarrow\leftarrow\leftarrow\leftarrow\uparrow\rightarrow\uparrow\leftarrow\uparrow\uparrow\ 011010011010$$

We note that there is a one-to-one correspondence between proper patches and their patch word, assuming the algorithm used to construct the spanning tree is deterministic and only depends on the shape of the patch. We will outline a deterministic algorithm in section 5.4.3.

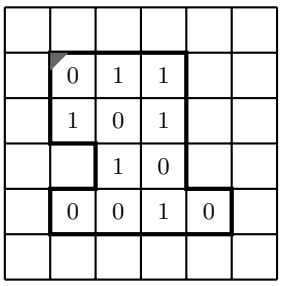


Figure 8: A proper patch and its content, with its leading cell highlighted

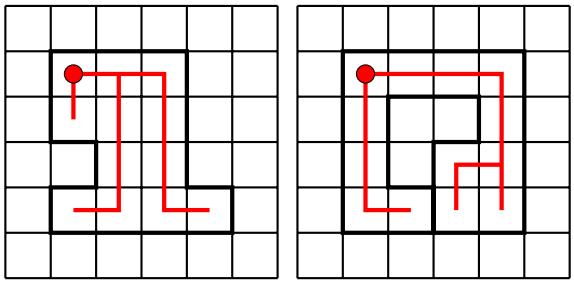


Figure 9: Example of spanning trees associated to proper patches, with their root highlighted

3.3. Local neighbourhoods

We introduce the partition of the border of a proper patch into *local neighbourhoods* as follows:

Definition 13. *A local neighbourhood is a maximum, connected component of the border for which the neighbouring patch does not change. We then canonically define the local neighbour associated with this neighbourhood as the adjacent patch.*

As we want the entire configuration to be partitioned into patches, the assumption that each patch must have neighbouring patches is satisfied during the algorithm.

We can also see on Fig 10 that the number of local neighbours of a patch can be greater than the number of its distinct adjacent patches. We also note that a proper patch can be its own local neighbour.

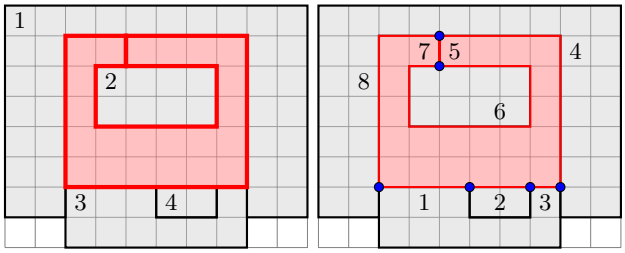


Figure 10: This patch has 4 adjacent patches, but 8 local neighbourhoods.

4. Overview of the leader election algorithm

All along the algorithm, the configuration will be entirely partitioned into patches, as defined previously. Instead of talking about the behaviour of cells, and for a better preliminary comprehension of the algorithm, we will talk about *meta-states* of patches, and how those meta-states must evolve according to a *meta-algorithm*. At the beginning of the algorithm, the configuration will be entirely decomposed into patches of size 1, *i.e.* each cell will be alone in its own patch. The final goal of the algorithm is to merge patches together, until the configuration is only composed of the periodical repetition of the same big patch. Fig 11 presents the evolution of a configuration from the starting point to the final state.

4.1. Meta-states and meta-algorithm

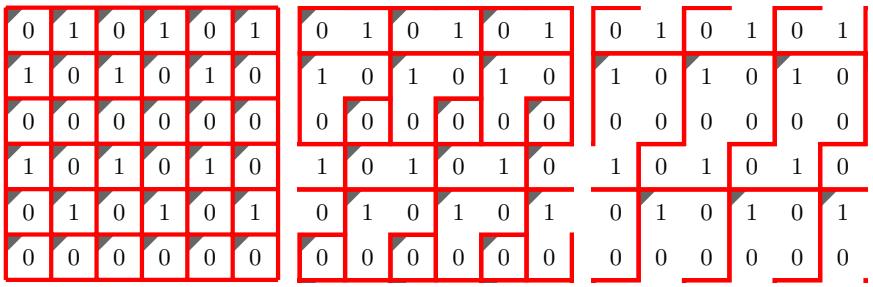
Each proper patch will have a meta-state for each of its local neighbourhoods (we will call them *merging* states), plus one special state (the *waiting* state). Merging states can be interpreted as if the patch was trying to merge with the local neighbour associated with the state, while the waiting state means that the patch is performing some lower level computation.

The behaviour of a proper patch will follow those rules:

- At the beginning of its existence, a patch will be in the waiting state for a finite time.

- After this time is over, the state will cycle over the merging states until the patch merges.

The essential point is that each pair of adjacent patches simultaneously in a merging state that corresponds to the same local neighbourhood must merge with each other (see Fig 12). The actual merging process will be detailed later.



(a) Initial state (b) Intermediate state (c) Final state

Figure 11: A possible evolution of a periodic configuration over time. Dark corners represent patch leaders.

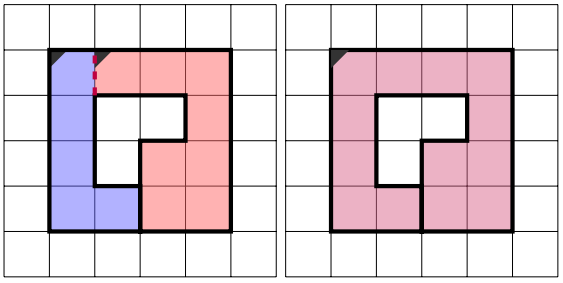


Figure 12: Two patches merging along a common local neighbourhood

4.2. Reformulating the main result

We will now explain the link between patches and leader election. It is clear that a single patch cannot contain two representatives of the same equivalence class, because two cells of the same equivalence class cannot have different behaviours during the computation. It follows that the maximum size of a patch (*i.e.* the maximum number of cells it contains) is the number N of equivalence classes. Moreover, if a patch of size N is eventually constructed by the algorithm, then the configuration is entirely tiled by translations of it. We can then elect a single equivalence class by simply selecting all the patch leading cells of the configuration (those cells obviously belong to the same equivalence class). In order to stick with the formalism used in theorem 1, we say that a cell is in the final subset $F \in \boldsymbol{Q}$ if it currently is a patch leading cell.

The goal of the algorithm is therefore to perform merging between patches as long as there exist two adjacent patches that are different. Indeed, if there

exists two different adjacent patches on the configuration, then they are not of maximum size.

Keeping this in mind, proving the following lemma is sufficient to prove the main result:

Lemma 1. *If at a certain time there exists two adjacent patches whose patch words are different, then at least one of them must merge within polynomial delay.*

5. Detailed algorithm

5.1. Cell division

As we stated before, we will subdivide each cell of the configuration into four sub-cells that will hold actual computation. We will talk about the *divided configuration* when it is necessary to consider all four sub-cells separately, and *united configuration* when it is not. The set of states $\boldsymbol{Q}$ of the automaton is therefore the product $\Sigma \times \boldsymbol{Q}_s^4$, where Σ is the input alphabet and $\boldsymbol{Q}_s$ is the states set of sub-cells. As we want to retain the input information at all time, the projection of $\boldsymbol{Q}$ over Σ will never change.

If we associate each sub-cell to a corner of the original cell, we obtain the informal division we used to define patches.

5.2. Detailed implementation of the meta-algorithm

In order to simulate the merging meta-states, each patch will construct and maintain a signal that will travel along its border. More precisely, this signal will cycle along each edge of the border, sometimes waiting longer than one time step on each edge. From an intuitive point of view, the signal will travel on the intersecting points between the closed curve we used to define a patch and the subdividing lines in the cells (see Fig 13). As there exist at most four of those points in a single cell, we can match each of them to a single sub-cell. Moreover, each of those points can also be associated to a single neighbouring cell, and therefore to a single local neighbourhood of the patch. Keeping this in mind, we now claim that a patch is in the merging state corresponding to the local neighbourhood associated to the point where its signal currently stands.

We note that, as we use Moore neighbourhood on the united configuration, the signal can travel from an intersection point to the next one in one time step.

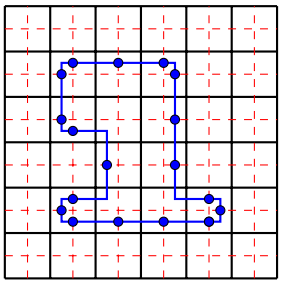


Figure 13: Highlighting of the waiting points of a patch

This signal will encode the shape and content of its patch into waiting delays along the border (this encoding will be detailed in section 5.4.4). It will behave so that the signal of two different patches will eventually de-synchronize and meet each other, thus leading to a merging.

5.3. First step

At the first time step, the automaton will construct patches of size 1 all over the configuration. The result of this operation, which can be performed locally, can be seen on Fig 14. Note that patches retain their input information at any time.

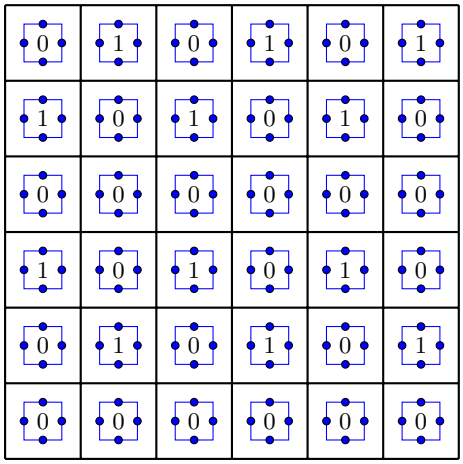


Figure 14: Patches of the initial configuration. Red lines are omitted for the sake of clarity.

5.4. Ongoing behaviour

In this section, we will consider the life cycle of a patch, from its birth to its death. We make the assumption that the border of this patch is *synchronized*, as done by a firing-squad-like algorithm (See [2] or [12]). This assumption is satisfied at the beginning of the computation, because this beginning in itself trivially is a synchronizing event. We will see later how a patch can synchronize itself when it is newly created. We do not make the assumption that the patch is a proper patch at the beginning of its existence.

The existence of a patch can be summarized as follows:

- The patch selects its leading cell (5.4.1),
- The raw patch is transformed into a proper patch by local modifications (5.4.2),
- The spanning tree is constructed (5.4.3),
- A signal is launched around the border of the patch (5.4.4).
- When two signals encounter each other, both patches are merged and synchronize their borders (5.4.5).

5.4.1. Patch leader selection

Let us consider a newly created patch, that is assumed to be synchronized. The first thing we want to do is to select the patch leading cell, as defined previously. Let us also consider the set of sub-cells defining the border of the patch. Those cells can be identified locally and form a ring-like structure. If we select the leftmost amongst the uppermost of those sub-cells, it can be trivially matched to the patch leading cell (see Fig 15).

It happens that the election of a particular cell on a ring-like structure on cellular automata is a well studied problem, and that [11] provides an algorithm that exactly fits our needs. We assumed that the border of the patch is synchronized, so all we have to do is to run the algorithm presented in [11] on the sub-automaton (*i.e.* the automaton whose cells match our sub-cells), then select as a patch leading cell the one whose upper left sub-cell has been selected by the algorithm. We note that this algorithm runs in polynomial time.

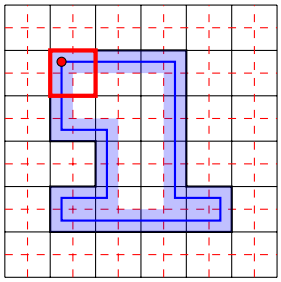
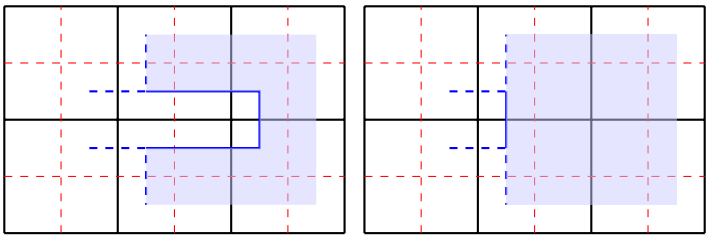


Figure 15: Highlighting the ring-like structure of the border

5.4.2. Healing the patch

We now want to turn our raw patch into a proper patch. This will be done by deleting all occurrences of the local pattern forbidden into a proper patch (see Fig 7a). From a practical point of view, a signal will be sent from the leading cell and travel along the border. This signal will replace local patterns according to the local rule presented in Fig 16. It is immediate to see that the iterative application of this local rule will turn the patch into its associated proper patch (see Fig 7).



(a) Before (b) After

Figure 16: Local transformation turning a raw patch into a proper patch. Blue area denotes the inside of the patch.

5.4.3. Building and browsing the spanning tree

Now that the patch is proper and the leading cell is identified, we can build the spanning tree starting from the leading cell with a very simple set of rules: If a cell has a neighbour in the patch in which the tree is constructed, an edge will be created between this cell and its neighbour at the next time step. If a cell has more than one neighbour fulfilling the conditions, it chooses one according to a predefined order among the direction (*e.g.* up > left > down > right). Fig 17 shows the ongoing construction of a spanning tree over an example patch. For practical reasons, this tree is oriented toward the root.

We now want to browse the tree and provide the leading cell with the next symbol of the tree on demand. This can be trivially done by simulating a finite state 2D-automaton performing a depth-first search on the tree and providing a symbol to the root each time it discovers a new cell. Note that this technique can guarantee a time interval between the reception of two symbols that is linear in the size of the tree. Specifically, the retrieval time is upper bounded by n^2, where n is the length of the border of the patch.

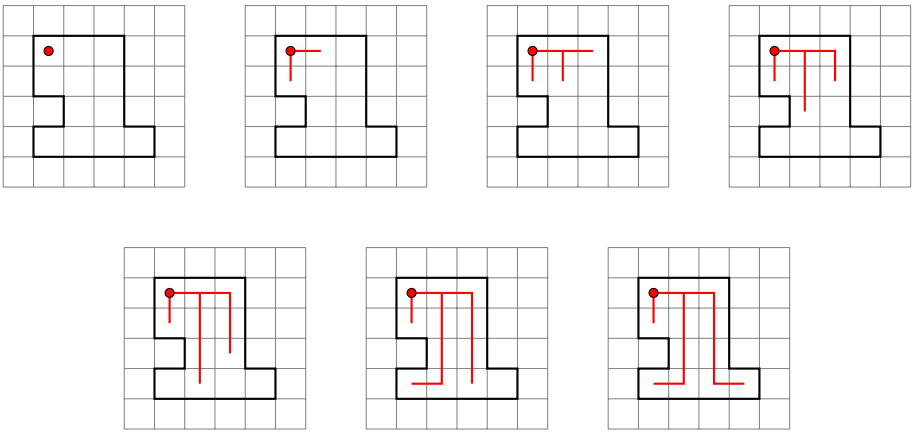


Figure 17: The construction of a spanning tree

5.4.4. Detailed signal behaviour

As soon as the patch is healed, the leading cell will send the signal that will trigger fusion between patches. The behaviour of this signal will be detailed in this section.

Definition 14. *We define the* length *of the border of a patch as its number of waiting points, as defined previously.*

The length of the border of the patch we consider all along this section will be denoted as n. Let us now consider the one-dimensional circular cellular automaton composed by the waiting points of a patch. We will call this particular automaton the *border automaton* of the patch. Clearly this border automaton can be simulated step-by-step by our real two-dimensional cellular automaton.

Definition 15. *We define the* leading waiting point *of the border automaton as the upper waiting point of the leading cell of its patch.*

Once the patch is healed, the leading waiting point will send a merging signal along the border automaton. This signal will cycle around the border automaton at maximum speed, and will sometimes carry a pebble, which we will call the *waiting pebble* (See Fig 18). At the beginning, the waiting pebble is positioned on the leading waiting point, which stores the symbol a_0. The signal then obeys the following rules, which are explained on Fig 18:

- Cycle around the border automaton at speed 1 until the waiting pebble is encountered (see Fig 18a).
- When the waiting pebble is encountered, move it to the next waiting point and wait for a certain *waiting time* τ_n on that point (see Fig 18b).
- If the signal has just waited on the leading waiting point, the signal will perform some *extra cycles* around the border automaton. The exact number of those cycles depends on a_i. The leading cell will then retrieve the next symbol a_{i+1} (or a_0 if a_i was the last symbol of the word), and the signal will resume normal behaviour (see Fig 18c).

Fig 20 sums up the behaviour of the signal and the pebble on a sample border automaton with $n = 4$. At any point during the existence of this signal, if it encounters a merging signal from another patch (see Fig 19a), then both signals are destroyed and a merging occurs. This merging process will be detailed in the next section.

Definition 16 (waiting times)**.** *We define the waiting time as* $\tau_n = k.n^2$ *where:* $k = \|\Sigma \cup \Gamma\| = \|\Sigma\| + 4$ *is the number of different symbols in a patch word;* n *is the length of the border, as defined previously.*

Definition 17 (extra cycles)**.** *The number of times the signal must cycle over the border automaton is given by* $val(a_i)$*, where val is any bijection from* $\Sigma \cup \Gamma$ *into* $[\![0; k-1]\!]$.

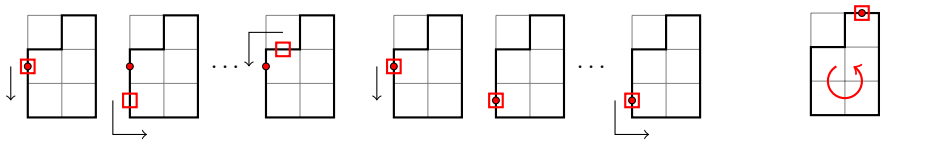
Those extra cycles will only happen when the pebble is dropped on the patch leading cell, thus inducing a time shift of $n.val(a_i)$ in the behaviour of the signal.

Note that the time τ_n is constructible[2] whatever the value of n is. For more general matter on time constructibility, please refer to [10].

This behaviour forces the leading cell to store the active symbol of the patch word. This can be done by browsing the border of the patch and its spanning tree, and sending the right symbol back to the leading cell. The cell will store only one symbol at a given time, starting from the first one (always a "$\rightarrow$" symbol) and sending a signal when it needs to retrieve the next one. We saw earlier that the time to retrieve a symbol of the patch word is bounded by n^2. Let us note that this time does not introduce a delay in the algorithm, since the time between which two successive symbols are requested is $k.n^3 + O(n^2)$.

In the next sections, we will denote the patch word as $a = a_0 a_1 \dots a_l$, and i will denote the position of the symbol currently held by the leading waiting point.

[2]There exists a computation on a 1-dimensional CA which marks a distinguished cell every τ_i, *i.e.* we can "count" up to τ_n.



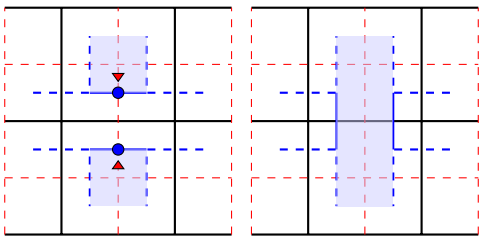
(a) Cycling phase (b) Pebble moving and waiting (c) Extra cycles

Figure 18: Overview of the behaviour of the signal in an example patch. The circle represents the waiting pebble, while the square represents the signal.

5.4.5. Local fusion and synchronisation

When two merging signals encounter on the common border of two patches, they must merge together, according to the global rule of our meta-algorithm. This can be done locally, following the process depicted on Fig 19: It is sufficient to delete the border portions on which the signals currently are, and "re-wire the border " so that they now form the border of a new, probably non proper patch. We can see that the resulting curve still respects the restrictions for defining a patch.

Once this fusion is done, we need to delete both spanning trees, by sending signals that will travel among them and destroy them, and both leading cells of the two former patches.



(a) Before (b) After

Figure 19: Local fusion when two signals face each other

We must now synchronize the whole patch border to get back to the situation of 5.4.1. This can be done by considering the new border as a one-dimensional cyclic cellular automaton of the divided network. We can identify a particular cell of this CA, *e.g.* a cell in which the merging process occurred. This cell can be used as the general of a firing squad algorithm, whose execution will re-synchronize the whole border, in order to perform a new leading cell election (see 5.4.1) on the merged patch. For general matter on firing squad algorithms, see [8]; here we merely note that they perform in linear time.

It may seem a waste to synchronize the whole border to elect a new leading cell, while the leading cell of the merged patch is always one of the two former leading cells. Indeed, it is possible to discriminate between the two cells with a linear time algorithm, but to the extend of our knowledge, such an algorithm has never been extensively studied and published. Detailing an *ad hoc* efficient method to elect the new leading cell would be out of the scope of this article,

which is not focused on the practical implementation of the algorithm.

Notes about Fig 20.

- We have $n = 4$ for this patch. We suppose that C_3 is the leading waiting point and $val(a_i) = 3$ for the instant we consider.
- The sum of the unlabelled time spans is n, which corresponds to the n moves of the pebble to the next cell, each one of length 1.
- The border automaton is in the same configuration at the two circled times on the left C_1 line, except that the symbol a_i pointed by the automaton has changed to the next one.

5.5. "Ending" the computation

Now that we have described our ongoing behaviour, we must talk about the ending of the computation. When all patches are of maximum size, *i.e.* when their size is N, they cannot grow any larger, because of the very definition of equivalence classes. Indeed, if two such maximal matches would merge, it would mean that two cells of the same equivalence class would have two different "roles" (intuitively, the cell of the patch on the right would be distinguishable from the cell of the patch on the left), which is impossible.

From a practical point of view, the patches will not merge because their patch words will be the same, and their signals will be completely synchronized.

Finally, at the end of the computation the configuration will entirely be decomposed into translations of the same maximal patch, into which a signal will cycle indefinitely. Note that due to the periodical nature of our computation model, we cannot have a particular cell to bear the information of the termination, like it is the case with CA whose input is bounded by persistent symbols.

5.6. Justification

In this section, we will prove that our construction satisfies the main result of 4.2, *i.e.* that if at a certain time there exists two adjacent patches whose patch words are different, then at least one of them must merge in polynomial time.

We will suppose that there exists P_1 and P_2 two patches with different patch words, and we will prove that if neither of them have merged with a third patch (in which case our main property would have been verified), then they must merge together in polynomial time. We will denote n_1 and n_2 as the border sizes and i and j as the symbol position on the patch words of P_1 and P_2 respectively. We study two disjoint cases: whether $n_1 \neq n_2$ or $n_1 = n_2$.

5.6.1. Different border sizes

We first suppose $n_1 \neq n_2$. We can assume without loss of generality that $n_1 < n_2$, namely $n_2 \geq n_1 + 1$. Let us now consider an instant when the waiting pebble of P_2 is dropped on a waiting point of the common border with P_1. The merging signal of P_2 will then wait for a certain time τ_{n_2} on that waiting point. We therefore have:

$$\begin{gathered} \tau_{n_2} = k.{n_2}^2 \\ \tau_{n_2} \geq k.(n_1 + 1)^2 \\ \tau_{n_2} > k.{n_1}^2 + 2k.n_1 \end{gathered}$$

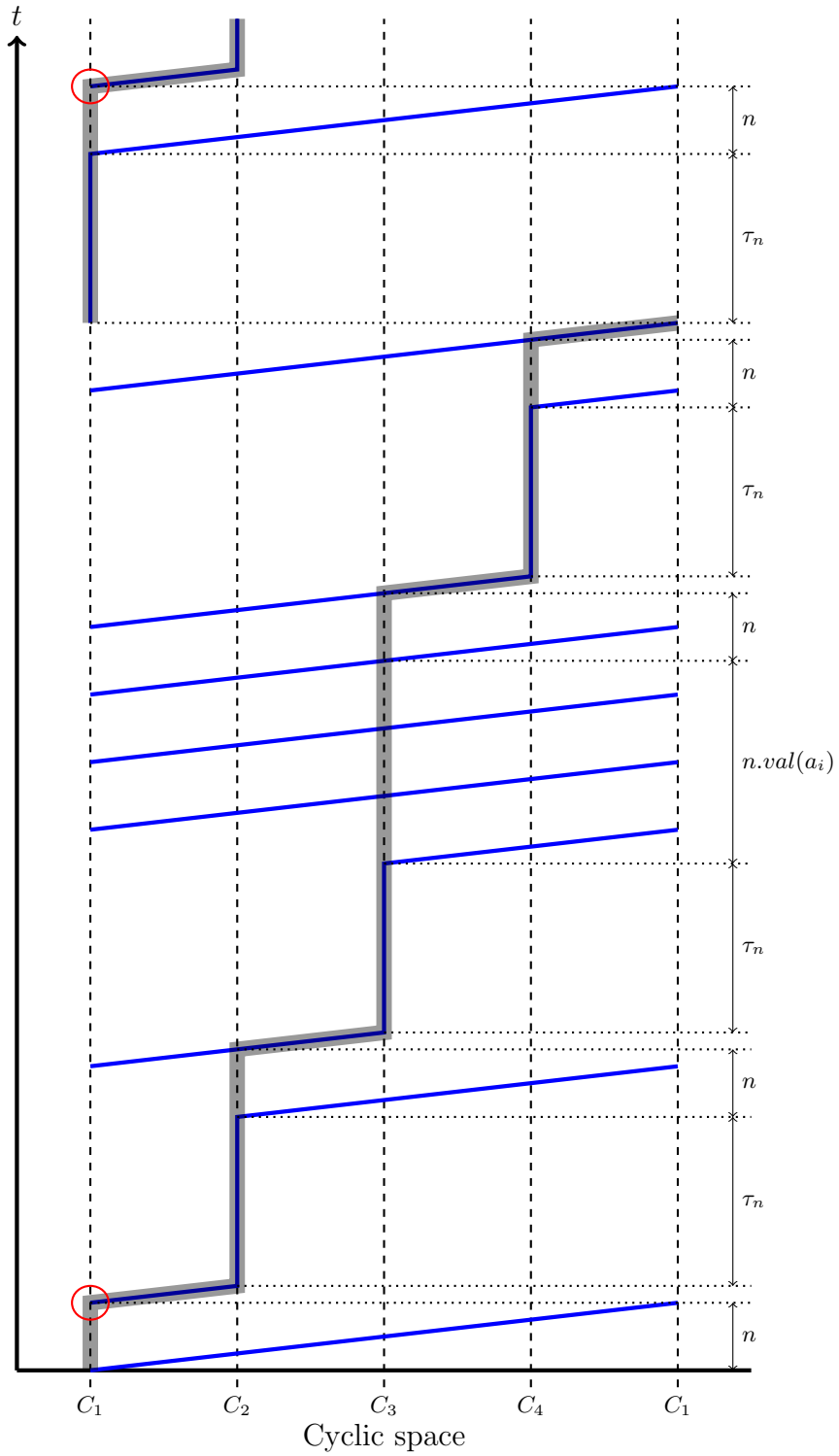


Figure 20: Evolution of the position of the signal (thin) and the pebble (thick) through time.

Let us now consider the maximum time during which the merging signal from P_1 can be absent from the corresponding waiting point on P_1. Clearly this time τ_{abs} is the sum of the waiting time on P_1, τ_{n_1} and the time for the signal to make a complete revolution around P_1, which is $n_1 - 1$. We therefore have:

$$\tau_{abs} = \tau_{n_1} + n_1 - 1$$
$$\tau_{abs} = k.n_1^2 + n_1 - 1$$

We clearly have $\tau_{n_2} > \tau_{abs}$, which means that the signal from P_1 will encounter the signal from P_2 before this signal leaves its waiting point. Therefore, a merging will occur between P_1 and P_2, *Q.E.D.*

5.6.2. Same border size

Now we consider the case when $n_1 = n_2 = n$. Let us call S_1, S_2, a and b the signals travelling through and the patch words of P_1 and P_2 respectively. Since P_1 and P_2 are different, a and b are also different. We now observe a single waiting point on a common border of P_1 and P_2.

Let us consider the behaviour of S_1 on this waiting point : it frequently passes through it, and sometimes waits for a time τ_n. Let us call $(t_k)_{k\geq 0}$ the sequence of instants when S_1 stops on the waiting point and waits during τ_n. By construction of the algorithm, there is exactly one instant between every t_k and t_{k+1} when S_1 will wait on the leading waiting point of P_1, then cycle around the automaton $val(a_i)$ times, for a certain a_i.

Because P_1 and P_2 have the same border size n, the behaviour of S_1 and S_2 on the point we consider are the same (because it only depends on n), except when they perform the extra cycles after waiting on the leading waiting point. Therefore, a similar reasoning also holds for S_2, which will wait on the leading waiting point of P_2, then cycle an additional $val(b_j)$ times at a single time point between t_k and t_{k+1}. We can therefore associate a couple of symbols (a_{i_k}, b_{j_k}) to each t_k.

We know that $a \neq b$, and by construction a cannot be a shift of b. Therefore there exists a k_0 for which we have $a_{i_{k_0}} \neq b_{j_{k_0}}$. Let us introduce a few particular times, shown on Fig 21:

Definition 18.

- *Let T_0 be the last time S_2 was present on the common waiting point before the time t_{k_0} associated to k_0.*
- *Let δ be $t_{k_0} - T_0$*

We note that there are two types of "holes" during which a signal is absent from a particular waiting point: holes of size n and holes of size $\tau_n + n$. We know that S_1 is present on the common waiting point for a duration of τ_n, starting at time $T_0 + \delta$. If we suppose that it does not encounter S_2 during that time[3], it means that this presence corresponds to a hole of size $\tau_n + n$ on S_2, as we clearly have $\tau_n > n$. We therefore know that S_2 will be present on the waiting point at time $T_0 + \tau_n + n$.

We note that the pebble performs a complete revolution around the automaton in time $n.(\tau_n + n + 1) + n.val(a_i)$ on P_1, and $n.(\tau_n + n + 1) + n.val(b_j)$ on

[3]If it does encounter S_2, then P_1 and P_2 will merge and the proof is over.

P_2. Now let us introduce a few more times, for the comprehension of which we encourage the reader to refer to Fig 21:

Definition 19.

- $T_1 = T_0 + \delta + n.(\tau_n + n) + n.val(a_i) + n$ *is the next time* S_1 *will wait during* τ_n *on the waiting point. Note that by our previous formalism, we also have* $T_1 = t_{k_0+1}$.
- $T_2 = T_0 + n.(\tau_n + n) + n.val(b_j) + n$ *is a time at which we are assured that* S_2 *will be back on the waiting point, due to the quasi-periodic behaviour of the signal (see Fig 20). We also know that* S_2 *will be there on time* $T_2 + \tau_n + n$.
- $\delta' = T_1 - T_2$ *is the difference between those two times.*

We will now prove that if S_1 and S_2 have not encountered before, then they will do so at time T_2 or $T_2 + \tau_n + n$.

We clearly have $0 < \delta < \tau_n + n$ by definition. Moreover, we must have $\delta < n$; otherwise it means that S_2 would come back while S_1 is still here, which would happen at $T_0 + \tau_n + n$. We finally have $0 < \delta < n$. It is also clear that $\delta' = \delta + n.(val(a_i) - val(b_j))$. Let $\Delta = val(a_i) - val(b_j)$, by construction we have $\Delta \in [\![-k+1, -1]\!] \cup [\![1, k-1]\!]$, because $a_i \neq b_j$. We will now discuss two sub-cases: whether $\Delta < 0$ or $\Delta > 0$.

Case 1: $\Delta < 0$.
Now we have $\Delta \in [\![-k+1, -1]\!]$ and $0 < \delta < n$. Therefore:

$$\delta + n.(1-k) \leq \delta' \leq \delta - n$$
$$n.(1-k) < \delta' < 0$$

These values for δ' ensure that both signals encounter at time T_2.

Case 2: $\Delta > 0$.
Now he have $\Delta \in [\![1, k-1]\!]$ and $0 < \delta < n$. Therefore:

$$\delta + n \leq \delta' \leq \delta + n.(k-1)$$
$$n < \delta' < k.n$$

These values for δ' ensure that both signals encounter at time $T_2 + \tau_n + n$ (which happens on Fig 21).

5.7. Temporal analysis

Now let us do a rough time analysis of the algorithm. We remember that N is the number of equivalence classes of the initial configuration, which is an upper bound to the size of any patch during the computation. As the length of the border of a patch is at worst linear in its size, we therefore have $n \in O(N)$ for any patch.

The key point of the previous section was to prove that if two different patches are adjacent, then at least one of them must merge before a certain time. We know that this merging must occur before the patches have cycled over their respective border words. We remember that the border automaton

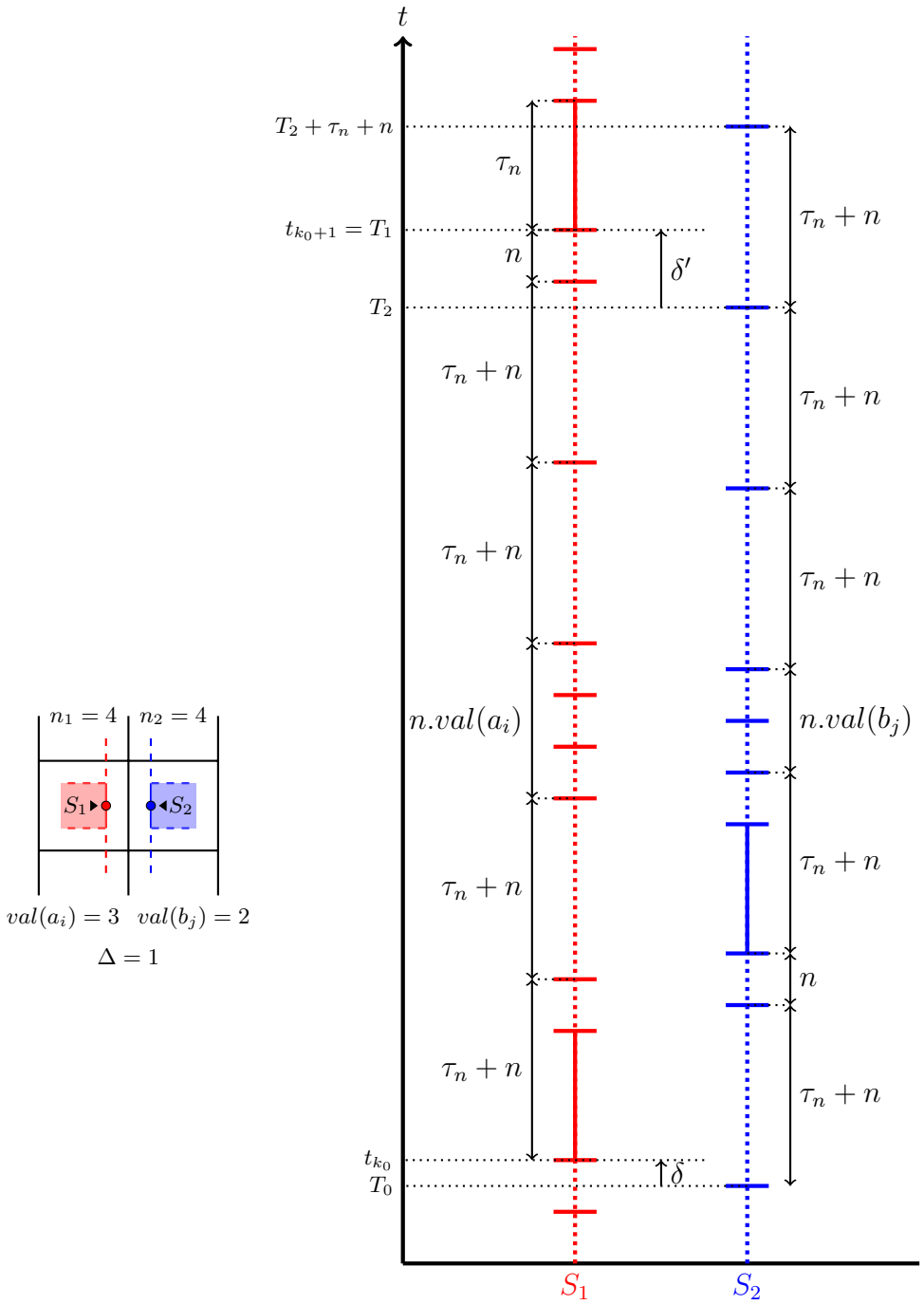


Figure 21: The signal presence diagram on a common waiting point of P_1 and P_2. The full line denotes the presence of the signal, while the dotted line denotes its absence. We note that the presence diagram of S_1 represents exactly the presence of the signal on the cell C_2 of Fig 20. Similarly, the diagram of S_2 could be the one of a neighbouring cell, for which the value of the symbol currently pointed would be $val(b_j) = 2$.

changes the symbol pointed on its patch word every time the pebble does a complete rotation on the border, which is every $O(N^3)$. As the patch word is of size $O(N)$, we are assured that two adjacent and different patches must merge before time $O(N^4)$. We note that the cost of the operations needed to merge patches properly is way less than $O(N^4)$.

Finally, as there exist N distinguishable patches at the beginning of the computation, and at least two of them must merge every $O(N^4)$, we are assured that the leading equivalence class is elected in time $O(N^5)$, which proves that our algorithm is polynomial in N.

Note that this analysis is extremely rough, as it does not take into account the parallel nature of the algorithm, and only consider two patches at a given time. We think that a more refined analysis would at least allow us to gain a N or N^2 factor, but it seems unnecessary regarding the primary goals of this paper.

5.8. Final remarks about the algorithm

An attentive reader would have noticed that we do not, in fact, *end* our computation, just have it cycle over a finite set of configurations. Informally speaking, it is a weak way to end a computation. We would rather want it to reach a fixed point, instead of a fixed cycle. We are convinced that a few tweaks to the algorithm can make it reach such a fixed point. The intuition is that if a patch has waited for a sufficient time without merging with another, it can determine that all of its neighbours are the same as itself, and can freeze its computations.

After the computation freezes, the cellular automaton can start any other algorithm using the patches. It may cancel this new computation and resume the patch merging algorithm later, if one of its neighbours merges (that would mean that it was not the "true end" of the computation). If all the patches have determined that their neighbours are the same as themselves and have frozen their computations, then the fixed point is reached. At the opposite, if it where applied to a non-periodical configuration, our algorithm would never reach such a fixed point.

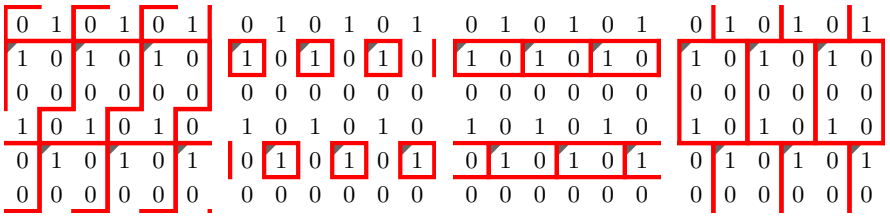
Moreover, when the computation "ends", we know that all the patches are identical. We do not know, however, if their position respectively to each other is the same. To ensure that fact, it only suffices to enrich the border word, by adding a special symbol that marks the point where the border of another patch has been encountered.

6. Conclusion and open problems

6.1. About tiling

It is important to note that our algorithm does not only compute a leading equivalence class, but also a polyomino that tiles the entire configuration by translation (this polyomino is the final patch). We know, thanks to [3], that the shape of this polyomino should be a pseudo-hexagon. In fact we can tweak our tiling patches to be rectangles (See Fig 22). This is done by starting with only the patch leading cells (Fig 22b), extending each rectangle to the right until another leading cell is encountered (Fig 22c), then extending it down until its bottom line reaches another leading cell (Fig 22d). It is easy to verify that each

rectangle exactly contains one representative of each equivalence class. This is a first step to a definition of a minimal pattern of periodical configurations of dimension 2.



(a) Final patches (b) Leading cells (c) Extending right... (d) ...then down.

Figure 22: From the final patches to a rectangular tiling of the configuration.

6.2. Toward picture language recognition

We have established that our algorithm, completed with the modifications of 6.1, can compute rectangular patches, whose size is the number of equivalence classes of the configuration. We can wonder if an extension of it could actually solve decision problems over these rectangular pictures, *i.e.* perform recognition of bi-dimensional languages (see [4]). We have studied the problem, and have concluded that it would indeed be possible, as it was the case in one-dimensional periodic configurations (see [1]). The results are not presented in this paper because the mere definition of what kind of languages are actually recognizable on a toric-CA should be discussed on its own, due to its intrinsic technicality.

6.3. Other uses of the algorithm

Our algorithm can serve other purposes than computing minimal patterns. Indeed, the patches can simulate cellular automata with bounded input, with the border of patches acting as persistent symbols. Keeping this in mind, we can for instance solve a 2D version of the density classification problem [7], *i.e.* determine if an infinite periodical configuration over the alphabet $\{0, 1\}$ contains more 0's than 1's, with more than two states: we simply have to count the number of 0's and 1's on a separate information layer inside the patches, then display the result, *e.g.* on the patch leading cells. A similar modification could also lead the algorithm to count the number of equivalence classes, instead of just electing one (as the number of equivalence classes of the original configuration is exactly the number of cells in the final patch).

6.4. Leader election in higher dimensions

We are convinced that the techniques presented in this paper could adapt to higher dimensions, and compute a leading equivalence class on periodical configurations of dimension 3, but such an extension is not trivial. For instance, the problem of a signal running through the border of a "3-dimensional patch" adds a new difficulty, such as the definition of a proper patch in dimension 3. However, even if some geometrical issues arise, the mechanism we used to encode patch differences into delay differences is robust. As it weakly depends on the dimension of the object it considers, it will perfectly adapt to periodical configurations of arbitrary dimensions.

[1] Nicolas Bacquey. Complexity classes on spatially periodic cellular automata. In Ernst W. Mayr and Natacha Portier, editors, *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France*, volume 25 of *LIPIcs*, pages 112–124. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

[2] Robert Balzer. An 8-state minimal time solution to the firing squad synchronization problem. *Information and Control*, 10(1):22–42, 1967.

[3] Danièle Beauquier and Maurice Nivat. On translating one polyomino to tile the plane. *Discrete & Computational Geometry*, 6(1):575–592, 1991.

[4] Dora Giammarresi and Antonio Restivo. Recognizable picture languages. *International Journal of Pattern Recognition and Artificial Intelligence*, 6(02n03):241–256, 1992.

[5] Jarkko Kari. Theory of cellular automata: A survey. *Theoretical Computer Science*, 334(1-3):3–33, 2005.

[6] Jarkko Kari. Basic concepts of cellular automata. In Grzegorz Rozenberg, Thomas Bäck, and Joost N. Kok, editors, *Handbook of Natural Computing*, pages 3–24. Springer, 2012.

[7] Mark WS Land and Richard K Belew. No two-state CA for density classification exists. *Physical Review Letters*, 74(25):5148–5150, 1995.

[8] Jacques Mazoyer. A six-state minimal time solution to the firing squad synchronization problem. *Theoretical Computer Science*, 50(2):183–238, 1987.

[9] Jacques Mazoyer. Computations on one-dimensional cellular automata. *Annals of Mathematics and Artificial Intelligence*, 16(1):285–309, 1996.

[10] Jacques Mazoyer and Véronique Terrier. Signals in one-dimensional cellular automata. *Theoretical Computer Science*, 217(1):53–80, 1999.

[11] Codrin Nichitiu, Jacques Mazoyer, and Eric Rémila. Algorithms for leader election by cellular automata. *Journal of Algorithms*, 41(2):302–329, 2001.

[12] Hiroshi Umeo. Firing squad synchronization problem in cellular automata. In Robert A. Meyers, editor, *Encyclopedia of Complexity and Systems Science*, pages 3537–3574. Springer New York, 2009.

[13] Hao Wang. Proving theorems by pattern recognition I. *Communications of the ACM*, 3(4):220–234, 1960.