


Article

# Generating Loop Patterns with a Genetic Algorithm and a Probabilistic Cellular Automata Rule

Rolf Hoffmann 

Computer Science Department, Technical University of Darmstadt, 64289 Darmstadt, Germany; hoffmann@rbg.informatik.tu-darmstadt.de

**Abstract:** The objective is to find a Cellular Automata (CA) rule that can generate “loop patterns”. A loop pattern is given by ones on a zero background showing loops. In order to find out how loop patterns can be locally defined, tentative loop patterns are generated by a genetic algorithm in a preliminary stage. A set of local matching tiles is designed and checked whether they can produce the aimed loop patterns by the genetic algorithm. After having approved a certain set of tiles, a probabilistic CA rule is designed in a methodical way. Templates are derived from the tiles, which then are used in the CA rule for matching. In order to drive the evolution to the desired patterns, noise is injected if the templates do not match or other constraints are not fulfilled. Simulations illustrate that loops and connected loops can be evolved by the CA rule.

**Keywords:** loop pattern generation; overlapping tilings; probabilistic cellular automata; pattern generation by genetic algorithm

## 1. Introduction

### 1.1. Main Aim

The main aim is to generate interesting pattern classes (from a global point of view) that are defined by local conditions only. An optional aim is to optimize a global criterion for such patterns, preferably in a local way. For instance, one could aim to fill a space with particles where the distance between them should be kept in a certain range and their number should be maximized.

### 1.2. The Problem

Our problem is to find loop patterns in a 2D space (grid/field of cells). We consider a square field of size  $n \times n$  under cyclic boundary conditions. First, we have to define what we mean by *loop* and *loop pattern*.

#### 1.2.1. Loop

We define a *loop* as a cyclically closed path (sequence of *path cells*) on a background. The value 1 is assigned to path cells (depicted in black/blue), and the value 0 to background cells (depicted in white/green). The loop length (the number of path cells) has to be greater than 1 (we forbid a single black cell to be a trivial loop).

A *loop path cell* is a black cell that has exactly two black cells in NESW where NESW are the direct orthogonal neighbors. We call this condition the “loop path condition”. This means that three path cells in sequence form a vertical or horizontal straight line or a corner (four variants by rotation). In order to form a loop, it is required that the neighboring cells of a path cell are also path cells, and all path cells in continuation form a closed path.

A loop path shall be surrounded by zero cells (*hull cells*). This is already partly fulfilled by the definition that a path cell has two white (just as two black) neighbors in NEWS. Additionally, at each corner (convex side) an extra zero has to be provided on the corner’s outer diagonal cell, for example 
$$\begin{matrix} 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{matrix} \Rightarrow \begin{matrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{matrix}.$$



**Citation:** Hoffmann, R. Generating Loop Patterns with a Genetic Algorithm and a Probabilistic Cellular Automata Rule. *Algorithms* **2023**, *16*, 352. <https://doi.org/10.3390/a16070352>

Academic Editors: Luca Mariot and Luca Manzoni

Received: 31 May 2023

Revised: 4 July 2023

Accepted: 20 July 2023

Published: 24 July 2023



**Copyright:** © 2023 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

The hull ensures that loops keep a (visible) minimal distance between each other, and also within a loop itself. Hull cells are allowed to be used by several paths, from the same or other loops.

*Remark.* We may imagine that an agent walks around in one direction on the surface of a torus and returns back to its starting point without touching its own path (except for the cyclic connection) or another loop. When walking along the path, the path cells are colored black, assuming that the field was initially colored in white. The agent follows its path by going straight, to the right, or to the left, assuming that the agent has a direction.

We distinguish the following loop types by counting the number of turns to the right (#R) and the number of turns to the left (#L) when walking for one cycle along a loop the surface of a torus in one direction.

- *Plane Loop.* Such a loop is a loop we commonly understand as a loop that can be easily recognized. The number of right turns minus the number of left turns sums up to +4 (360° clockwise) or −4 (360° counter-clockwise):  $|\#R - \#L| = 4$ .  
A plane loop can be visualized in a plane, but not necessarily in a field of size  $n \times n$ . Plane loops can use the inherent cycles available in a torus, but not necessarily. Then, they do not fit into a field of size  $n \times n$  but they can be visualized by repeating the field periodically.
- *Wrapped Loop.* A wrapped loop is a loop that makes use of the inherent cycles in a torus. We can distinguish *straight* loops and *wave* loops.
  - A *straight loop* is just a horizontal or vertical line wrapped around using one or the other inherent cycle. The condition is  $\#R = \#L = 0$ .
  - A *wave loop* shows a wave-like structure vertically and/or horizontally wrapped around. The condition is  $\#R = \#L = k, k \geq 2$ . A wave loop can be very long, e.g., it can cycle many times around as long as there is enough space.

### 1.2.2. Loop Pattern

A *loop pattern* is a cell field with a white background that contains at least one loop. Several loops of different types may appear in such a pattern, and a loop may enclose other loops. Note that connected and intersecting loops are not allowed. The field size  $n > 1$  can be arbitrary, but we have to notice that not every loop type can appear for an arbitrary size.

For instance the plane loop  $\begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{matrix}$  needs at least a space of  $4 \times 4$ .

An *optimal loop pattern* is a loop pattern that maximizes or minimizes a global criterion. A global criterion is a measure that may take all cell states into account. Usually it is easily computed on the global level, often by counting some local conditions or relations between cells. In the GA (Genetic Algorithm) we maximize the number of tile matches that are locally computed but globally counted. In the CA, we try to minimize the number of uncovered cells (to zero) by using a local rule only.

We aim at patterns that show loops only. Nevertheless, there may appear *faulty loop patterns* (loop-related patterns) during or at the end of the pattern evolution by the used GA or the CA. A *faulty loop pattern* is a pattern that shows some similarity to a loop pattern. One could define a similarity measure such as the minimal number of pixels that need to be changed in a loop-related pattern in order to yield a loop pattern. Objectionable black cells could be deleted (faulty cells, not-needed black cells, branches, or connections between loops, see Section 2.6). *Remark:* This task could also be used for mind training or as a game between two players.

The main challenge is how to form a loop (its path and closing it) by local conditions. The solution presented here is the definition of a set of tiles (local patterns acting as local conditions) that are allowed and forced to overlap (aggregate) in a way that loops are formed. Such a set of tiles is designed and tested by a GA in a preliminary stage. The GA tries to find optimal loop patterns by covering the space (totally or partially) with the defined tiles. The GA works on a *global* criterion, namely maximizing the number of matching tiles.

After having found an appropriate set of tiles, the second challenge is to find a *local* Cellular Automata (CA) rule that can evolve the desired loop patterns. To solve this problem, we extend the approach already used to generate point patterns [1], domino patterns [2], and sensor networks [3]. Local matching patterns (templates) are derived from the tiles and tested by the CA rule. If there is no template hit, noise is injected that finally drives the evolution to a loop pattern.

### 1.3. Related Work

*Own Previous Work.* Complex patterns were constructed in a local way by moving CA agents [4]. The agents use a finite state machine evolved by a GA. Though the results are impressive, it takes quite some effort to train the agents. In addition, it is very difficult to train them for small and large grid sizes at the same time in order to make them work for any grid size. It would be a new and interesting task to train them for forming loop patterns and to find out the limitations.

Then, a probabilistic CA rule was used to generate point patterns [1], sensor point patterns [3], and domino patterns [2,5]. It was possible to aim at a maximal or a minimal number of dominos by injecting noise depending on the level of tile overlapping. In this paper, the idea of using a probabilistic CA is also followed. The rule here was adapted to a new set of tiles and uses an additional logical condition in order to evolve loop patterns *only*.

*Genetic Algorithm for Pattern Generation.* In our internal research work, we used for several years a GA to generate patterns with a global fitness function based on local conditions and local pattern matchings. This method is first published in this article. GA is a generally accepted method for optimization. It dates back to John H. Holland [6,7] and overviews are given in [8,9]. We just used the classical techniques (crossover, mutation, selection) in a simple way, i.e., we randomly selected the second parent (the mate) without giving preference to parents with high fitness. This was sufficient for our purpose and it was not the intention to find a more efficient evolutionary algorithm or heuristic.

*Overlapping Tiles and Loops.* The recommendable book [10] provides much information about tilings but it does not deal with overlapping tiles. Overlapping 1D tilings are treated theoretically in the context of recognizable languages, monoids, and two-way automata [11], and it was partly motivated by an application in computational music theory. This paper is a good basis for further studies and extensions to the  $n$ -dimensional case. There are a lot of applications for overlapping tilings, like the dense parking of cars in a parking lot or constructing a sieve for maximal particle flow (using domino tiles) [5], optimizing task scheduling (overlapping communication and computation) [12], or building nano-structures [13].

Our problem is related to open and closed (loop) *space-filling curves* [14–16]. In principle, such loops can be generated with our approaches (GA and CA), among others. The wave loop (Figure 11 (fourth in the second row)) is a space-filling wave loop. It is also an open space-filling curve if the horizontal wrap-around of the torus is not used. The wave loop (Figure 11 (second in the first row)) is a space-filling wave loop in the torus. The proposed GA could be tuned (by additional global fitness measures) to generate such curves *only*. Generating *only* space-filling curves by a CA rule is a task that has to be performed. For space-filling curves, the field size has to be considered with respect to the possible solutions. Note that in our approach, the space between loops is not constant 1, it can vary between 1 and 2 (or even more if allowing uncovered cells). So our loops can be arranged in a more flexible way for any grid size, they can keep a suitable distance out of a certain range. This feature is also valuable for describing particles that can attract and repulse. Hamiltonian cycles can algorithmically be computed in grid graphs. In [17], the grid graph is interpreted as a field of cells as here, whereas in [18] a checkerboard marked grid graph is used. In [19], families of intersecting closed curves (connected loops) are discussed in the context of string topologies. This paper could be of interest for further work on generating connected loops.

*Cellular Automata.* A CA is a grid of automata (cells). Every cell changes its state depending on its own state and the state of its local neighbors according to a local rule. The model of computation is parallel, simple, powerful, and has a wide range of applications [20].

#### 1.4. New Results

- A GA was designed that can use an arbitrary set of tiles (small matching patterns) in order to find optimal patterns depending on a *global fitness function*. The fitness function used here is the *number of local tile matches* being maximized. The working principle is demonstrated for an example: Generating patterns with points and squares.
- Then, the GA was used for generating loop patterns. A *set of overlapping tiles* was defined, which can aggregate to loops. Loop patterns appear by maximizing the number of tile hits. Most of the generated patterns are loop patterns, but not all of them.
- A CA Rule was defined that *evolves stable loop patterns only*. It uses (i) templates derived from the prior defined set of tiles, and (ii) an extra logical condition, the loop path condition. This extra condition can easily be modified in order to allow connections between loops.

#### 1.5. Paper Structure

The paper is structured into two main parts: (i) using GA to generate loop patterns (Section 2), and (ii) using a CA rule (Section 3). In the first part, the GA is described, which can generate such patterns based on a set of designed tiles. The purpose is to affirm that the designed tiles do the job and can then be used in the second part. In the second part, a CA rule is designed using templates derived from the set of tiles. Different loop patterns evolved by the CA rule are then presented. In Section 4, the findings are discussed.

## 2. Generating Loop Patterns with a Genetic Algorithm

First, we will show how 2D patterns can be generated by a GA using a set of local matching patterns (tiles). Then, a tile set is designed that can generate loop patterns.

We aim at binary patterns  $S$  of size  $n \times n$ , where the elements  $S(x, y) \in \{0, 1\}$ , and the coordinates  $x, y \in \{0, \dots, n - 1\}$ . We may call the elements also “cells” as we will later deal with cellular automata cells and patterns of them. We will use also the term “field” or “space” for an array of cells.

### 2.1. Tile

A *tile*  $L$  is a pattern of  $m_1 \times m_2$  elements  $L(x, y) \in \{0, 1, -\}$  where  $m_1, m_2 \leq n$ . We call the elements “pixels” in order to distinguish them from cells. The symbol “-” represents a *null*, a pixel that is not part of a tile. A non-null pixel 0/1 is also called a *valid pixel*. We want to use only tiles that are much smaller (or smaller for small fields) than the size of the whole field. The reason is that we do want to allow that a few large tiles of size  $m_1 \approx n$  and/or  $m_2 \approx n$  may simply cover the whole global space. Then, a trivial solution could be that just one tile is equal to the aimed pattern. In fact, we want to restrict the size of the tiles,  $m_1 \leq B_{m_1}$  and  $m_2 \leq B_{m_2}$ , where  $B_{m_1}, B_{m_2} \geq 1$  are the boundaries of a tile, and where  $B_{m_1} \times B_{m_2} \geq 2$ . Here, we want to solve the problem with small tiles of size  $B_{m_1}, B_{m_2} \leq 5$ . The coordinates  $(x, y)$  within a tile  $L$  are locally defined separately for each tile where the origin  $(0, 0)$  is assigned to the central pixel of a tile.

*Remark.* It would be possible to define a tile as a set of pixels  $(x, y, v)$  where  $(x, y)$  are the coordinates and  $v \in \{0, 1\}$  are the states of a pixel. In addition, the name of the tile and the anchor’s position have to be related.

### 2.2. Overlapping Tiles

Now, we search for global patterns that can be tiled by tiles from a set of tiles  $\underline{L} = \{L_0, L_1, \dots\}$  that may overlap. Two tiles  $L_1, L_2 \in \underline{L}$ ,  $L_1 \neq L_2$ , *overlap* if they are overlaid (shifted relative to each other) and all overlaid non-null pixels have the same

value. The result of overlaying a pixel  $p_1$  from  $L_1$  with a pixel  $p_2$  from  $L_2$  is summarized in Table 1.

**Table 1.** The result of overlaying two pixels.

| Pixel $p_1$ | Pixel $p_2$ | Result    |
|-------------|-------------|-----------|
| 0           | 0           | 0         |
| 1           | 1           | 1         |
| -           | 0/1/-       | -         |
| 0/1/-       | -           | -         |
| 0           | 1           | forbidden |
| 1           | 0           | forbidden |

It is possible that  $k \geq 2$  tiles overlap. In this case, there exists pairwise an overlap and the resulting overlap forms a *compound tile* consisting of  $k$  sub-tiles. In terms of particle systems, we may associate a basic particle with a (basic) tile, and a compound/complex particle with a compound tile.

Two tiles  $L_1, L_2$  may also overlap (by inclusion) if  $L_1$  is smaller than  $L_2$  but compatible with it ( $L_1$  has the same pixel structure as  $L_2$  except that some of the valid  $L_2$ -pixels are null in  $L_1$ ). Overlap by inclusion is symmetric.

We define a field of cells as *fully covered* if all cells are covered by tiles (the number is not restricted) taken from the given set of tiles, and tiles are allowed to overlap as explained already. In order to cover the field, not only the basic tiles but also compound tiles can be used. Due to a not properly defined set of tiles, it may happen that a given field cannot totally be covered by valid pixels. Then, the field is *partially* (not fully) covered because it contains uncovered cells (null pixels, gaps, wholes, spaces). Nevertheless, partially covered fields may be of interest as patterns too.

### 2.3. Genetic Algorithm

In the first stage, we use a GA to find optimal patterns with respect to a global fitness function and defined by tiles. The main purpose here is to validate that a certain set of designed tiles define the patterns that are aimed at the user and that can be used later by the CA rule in the second stage.

The algorithm used is given in Algorithm 1. A possible solution (an individual) is a tuple (*Pattern, Fitness*). An array/list of  $M$  solutions is the data and output that is manipulated by the algorithm. (1) The pattern list is initialized randomly. (2) The while loop is repeated until a termination condition becomes true, i.e., the number of iterations reaches a certain limit, and / or another global condition, like a certain fitness level or certain pattern features. (3) In each block, better solutions are searched for. For each current existing individual (tentative solution), an offspring (a new individual) is generated by crossover and mutation. The current individual is replaced by the offspring if it has higher fitness. (4) A mate  $S_j$  is selected at random from the list. (5) A new offspring pattern is computed by crossing over  $S_i$  with  $S_j$ , and then applying a mutation. (6) The fitness of the offspring pattern is computed and stored within the offspring. (7) The offspring replaces the current individual  $S_i$  if its fitness is higher. (8) The list of solutions is sorted for output.

The *fitness* function computes the number of tile matches that occurs within a pattern. At every site of the pattern, all tiles from the given set are tested if they match locally, i.e., test if all tile pixels (aligned to the tile’s anchor/center) are equal to the pattern values in the local neighborhood. A null tile pixel (do not care) excludes the testing of the corresponding pattern value.

The used GA is a simple form using the classical GA algorithm basics. The goal was to generate optimal patterns in a simple way and not to optimize the algorithm itself, which is a topic for further research. Only one list of individuals is used and not two (old and new generation).

**Algorithm 1:** Generating a Binary Pattern with a Genetic Algorithm

---

```

Types:          Pattern = array [0 ... n1 - 1, 0 ... n2 - 1] of {0, 1}
                Solution = (Pattern, Fitness)
Data and Output: S is an array [1 ... M] of Solution
Temporary:     Offspring is a Solution
(1) S.Pattern ← RandomPatterns
    (2) while not TerminationCondition
        (3) foreach Si in S
            (4) Select randomly a Mate Sj
            (5) Offspring.Pattern = Mutate (Crossover (Si, Sj))
            (6) Offspring.Fitness ← fitness (Offspring.Pattern)
            (7) if Offspring.Pattern ∉ S.Pattern and Offspring.Fitness > Si.Fitness
                then Si ← Offspring
        (3) foreach end
    (2) while end
(8) S ← SortByFitness (S)

```

---

Algorithm 1 that generates patterns with a maximal number of tile matches. The fitness function computes the number of matches. Every site of the pattern field is tested if there is a match of a tile out of a given set of tiles.

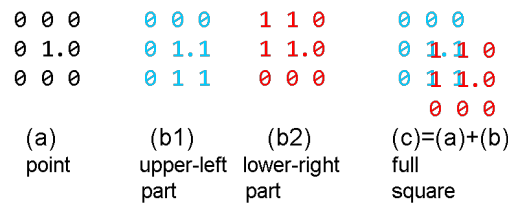
Each individual is treated separately and is expected to improve by crossover with any other individual, not depending on their fitness. Thereby a high diversity is supported, though the speed of improvements may not be so high. Crossover is performed in the following way (uniform crossover with a certain probability): (1) Each bit of the offspring  $Crossover(S_i, S_j)$  (without mutation) is taken from the mate  $S_j$  with probability  $p_1$  otherwise unchanged from the parent  $S_i$ . (2) Then, mutation is performed on each bit with probability  $p_2$  yielding  $Mutate(Crossover(S_i, S_j))$ . Then, the fitness of the offspring was computed and used for replacement in the case of improvement.

The probabilities used were  $p_1 = 0.2, p_2 = 0.05$ . Optimal or near-optimal solutions were found within a number of iterations between 1000 and 10,000 in a short time (a few minutes for  $12 \times 12$  patterns on a desktop PC with an Intel Core i5-3470 CPU @ 3.20 GHz and 8 GB RAM under Windows 10). The programming language was Free Pascal using one thread only. The size of the population (the number of possible solutions) was 40.

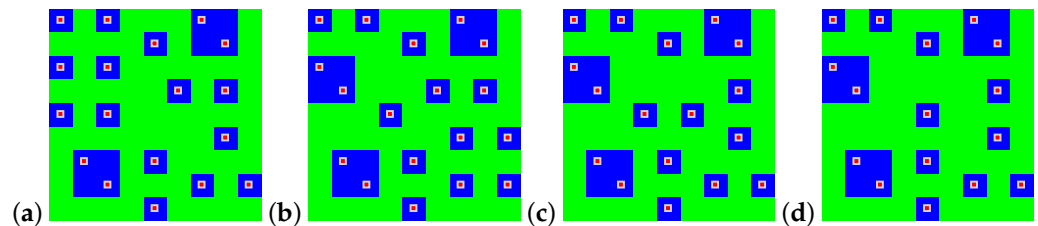
#### 2.4. Sample Pattern: Squares and Points

In this section, we demonstrate that the GA can produce optimal “square/point” patterns (SP patterns) defined by three tiles. The pattern contains simple 1-points and squares  $\frac{1}{4} \frac{1}{4}$  surrounded by zeroes. In order to keep the computations as local as possible, we restrict the size of the tiles to  $3 \times 3$ . The designed three tiles are shown in Figure 1. The *point* tile (a) is just a 1 surrounded by 0’s. The center is marked and is the anchor/origin of the tile. A square surrounded by zeroes can easily be defined by a  $4 \times 4$  tile, but not by a  $3 \times 3$  tile. Therefore, the square is composed of two parts (b1) and (b2), which then aggregate to a full square (c) (without 2 zeroes at two of the corners) during the GA evolution.

Some solutions of the GA are shown in Figure 2. The tile matches are marked. All patterns are valid as desired (show only squares and points surrounded by zeroes) and each cell is covered (the field is fully covered). Patterns (a) and (b) are optimal with a maximal possible number 18 of matches. (a) Contains 2 squares and 14 points, whereas (b) contains 3 squares and 12 points. Patterns (c) and (d) are non-optimal with respect to the number of matches.



**Figure 1.** The three tiles used for the first pattern generation by GA: (a) point; (b1) a square (11/11) with upper-left zeroes; and (b2) with the lower-right zeroes. The tile centers are marked by “.”. The tiles b1 and b2 can overlap forming a “full” square surrounded by 10 zeroes (at relevant positions), where “1” marks an overlap.



**Figure 2.**  $9 \times 9$  patterns generated by a Genetic Algorithm using three tiles, a point, and two overlapping partial squares that yield together a full square. The number of tile matches was optimized. The positions where tile centers match are marked. (a) 18 matches. (b) 18 matches. (c) 17 matches. (d) 16 matches.

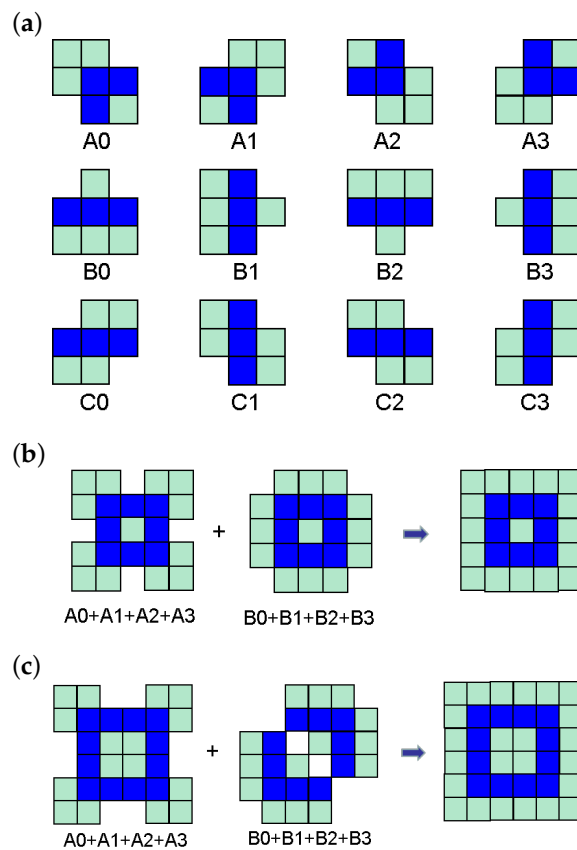
### 2.5. Tiles for Generating Loop Patterns

Now, we want to generate a loop pattern using the GA. A set of 12 tiles was designed specifying such patterns (Figure 3a). The size of the tiles was restricted to  $3 \times 3$ . The tiles were designed in the following way: (1) possible loop patterns were constructed on a grid (the *aimed patterns*). (2) The aimed patterns were scanned by small windows at several relevant positions. Such *window patterns* were stored or remembered. (3) Tiles were designed that can cover the window patterns and that can overlap.

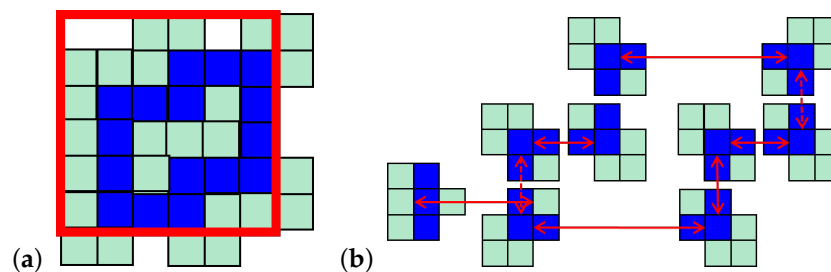
The tiles A (Figure 3) define corners, B define straight continuations, and C define possible turns. The null pixels are necessary in order to enforce connections. Simply speaking, the tiles’ surfaces need properly structured interfaces in order to aggregate. At the moment it is not clear how the interfaces can be defined in a methodical way.

Figure 3b,c shows how certain small loop patterns can be assembled by tiles. A small open square (Figure 3b) can be formed as shown by  $(A0 + A1 + A2 + A3) + (B0 + B1 + B2 + B3)$  where ‘+’ means adding a tile or compound tile that overlays without conflict. The resulting composition can fit into a cyclic field of size  $4 \times 4$  or  $5 \times 5$ . A larger open square (Figure 3c) can be formed as shown fitting into a cyclic field of size  $5 \times 5$  or  $6 \times 6$ . Figure 4 shows (a) another small loop and (b) how it can be assembled by tiles.

Until now, it seems difficult to automate this process, and therefore, it is a topic of further research. It is also a question of what is a minimal set of  $m_1 \times m_2$  tiles, or what is the smallest size of the tiles in order to produce the same class of patterns. Note that there are different classes of loop patterns that differ in certain details like allowed crossings (and how) or allowed diagonal connections. We discuss here only the class of loop patterns that are defined by the tile set shown in Figure 3a.



**Figure 3.** (a) The twelve tiles defining loop patterns. (b) A small open square compound tile. (c) A larger open square compound tile (“open” means here not fully surrounded by zeroes).

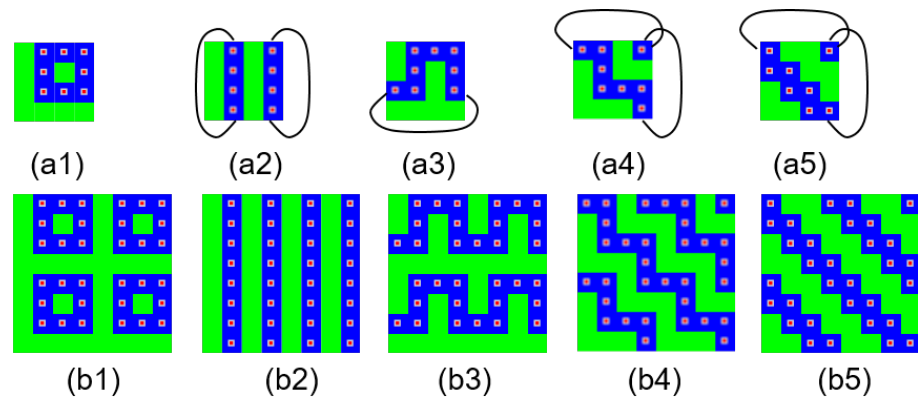


**Figure 4.** A  $6 \times 6$  loop pattern. (a) The field boundary is marked in red. Pixels outside the boundary wrap around and thus make the field fully covered. (b) The used tiles and their connections. Arrows mark pixels (and their associated tiles) that overlay. Dotted arrows show adjacent pixels/tiles that connect but do not overlay.

2.6. Loop Patterns Generated by GA

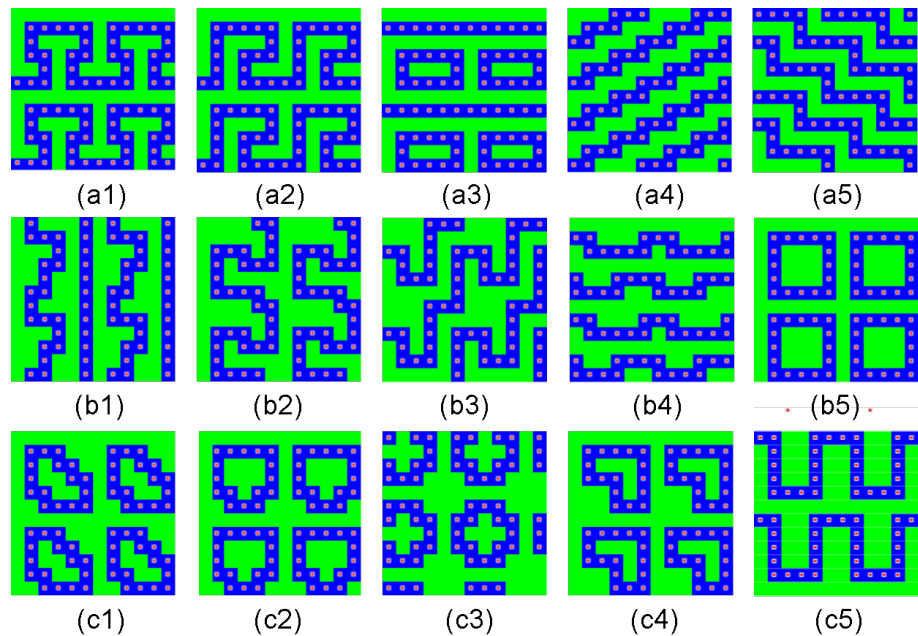
Let us consider generated patterns of size  $n \times n$ . We consider patterns as ‘equal’ if they are equivalent under rotation, shift, or mirroring. For  $n = 2$ , there exists only the loop pattern  $\begin{matrix} 0 & 0 \\ 1 & 1 \\ 0 & 0 \end{matrix}$ , and  $\begin{matrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{matrix}$  for  $n = 3$ . Note that a straight line of one-pixels forms a loop under cyclic boundary conditions. For  $n = 3$ , there exist five patterns as shown in Figure 5a1–a5. All patterns contain one loop, except (a2), which contains two loops. Patterns (a4) and (a5) use two cyclic connections each, one horizontal and one vertical. The patterns (b) are the patterns (a), doubled vertically and horizontally, altogether they form quadruplicates, and that is why we call them *quad patterns* or simply *quads*.





**Figure 5.** GA evolved loop patterns of size  $4 \times 4$ . The *quad* patterns (b1–b5) repeat the patterns (a1–a5) horizontally twice and vertically twice in order to exhibit the inherent cyclic structures. The cyclic connections relevant for the loops in (a1–a5) are noted. (a1) is a plane loop, (a2) shows two straight loops, and (a3–a5) are wave loops.

For  $n = 6$ , some of the GA evolved patterns are shown in Figure 6 as quads. The patterns (a) have the same number of 1- and 0-pixels, and therefore, we can call them *balanced*. They are also very *dense* if we consider 0-pixels as space between the 1-pixel loops. Patterns (b, c) are less dense, because their loops are more commodiously embedded in the space. The used GA favors dense patterns, but it could be modified to aim at low-density loop patterns. Figure 6a3,b1,b4 show two loops, and the other patterns only one. Patterns (b5, c1–c4) contain plane loops. Pattern (a3) contains a plane and a straight loop, (b1) a wave and a straight loop, and (a1, a2, a4, a5, b2, b3, b4, c5) are wave loop patterns.

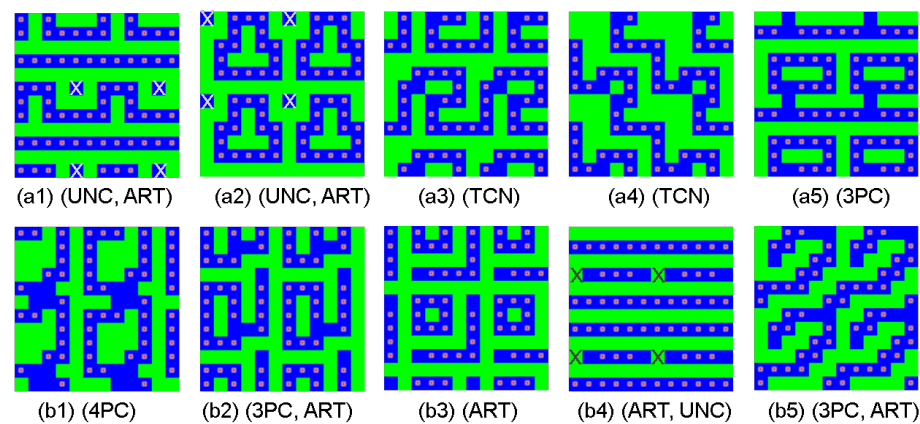


**Figure 6.** GA evolved loop patterns of size  $6 \times 6$ , depicted in quad representation. Patterns (a1–a5) consist of 18 one-pixels (blue) and 18 zero-pixels (green). Patterns (b1–b5,c1–c4) have 16 one-pixels. Pattern (c5) has only 14 pixels.

The GA does not always produce loop patterns (Figure 7). GA generated *optimal* patterns are patterns with a maximal number of tile matchings. Such patterns correspond to dense loops, as shown in Figure 6a1–a5, and their number of one- and zero-pixels is balanced (may differ by one if the whole number of pixels is not even). The generated non-optimal patterns, in which the number of tile matches is not maximal, can be (a) loop

patterns, or (b) faulty loop patterns. We can observe special types of faults that may appear in combination (Figure 7).

- (UFC) Uncovered cell. The given space is only partially filled with tiles. The isolated and marked blue cells in Figure 7a1,a2 are such cells. The GA generates for such uncovered cells either 0 or 1. Such cells can automatically be detected and then shown in a distinct color (e.g., white or green).
- (TCN) Touching corners. There exists a pair of one-pixels that touch each other on a diagonal.
- (3PC) 3-pin-connector. There exists a 3-pin connector in a loop path (Figure 7a5,b2,b5).
- (4PC) 4-pin-connector. Such a connector exists in a loop path (Figure 7b1).
- (ART) Artifact. There exist other black pixel structures that may be a part of a possible loop or an appendix of a loop. Typical such structures are: a line, a partial rectangle, or a branch (Figure 7b2–b5).



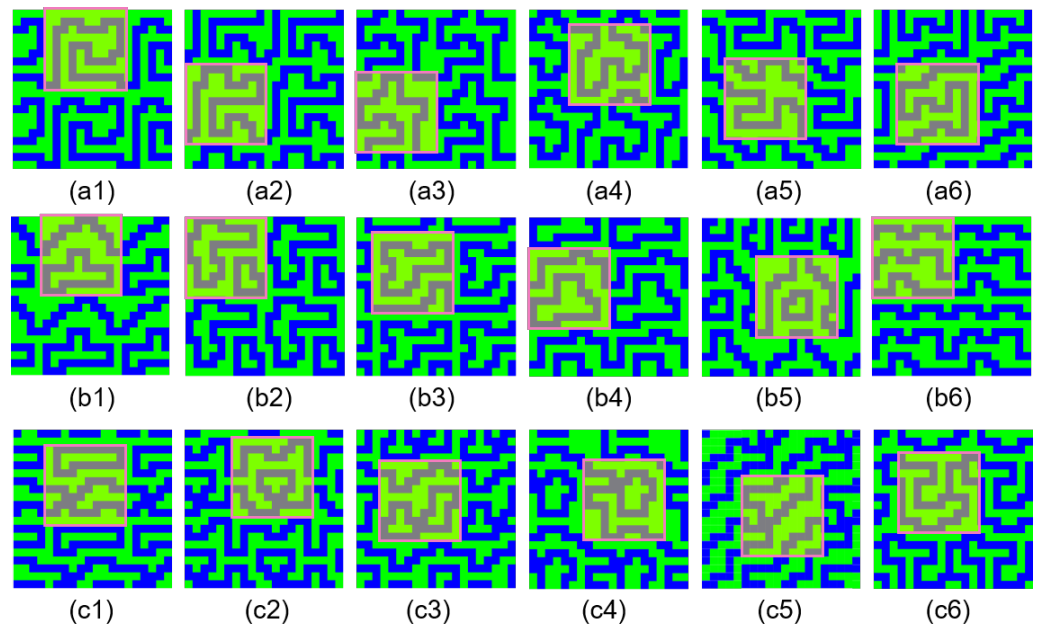
**Figure 7.** Faulty loop patterns with different faults: (UNC) uncovered with wrong color (marked by 'x'); (TCN) touching corners; (3PC) 3-pin-connector; (4PC) 4-pin-connector; (ART) artifact.

*Remark.* In the course of the further GA evolution of the (b4) pattern, the uncovered cells can be eliminated such that only line loops will appear. In (b5), a tile can be added, closing the gap and thus forming connected loops.

It is a matter of definition what kind of loops are allowed. It would be possible to modify the fitness function of the GA in a way that (a) TCN and certain ART structures are suppressed (adding a penalty to the fitness function); (b) 3PN and/or 4PN are promoted (adding a reward); (c) UFC are allowed in a certain range in order to control the space between loops.

For  $n = 10$ , some of the GA evolved patterns are shown in Figure 8 as quads. The patterns contain one loop (a), two loops (b1–b5), or three loops (b6). The patterns (c) contain connected loops. Loops are connected through connectors. A connector with four pins is used in (c1–c5), and one with three pins in (c6). We find  $10 \times 10$  windows marked in the quads intended to exhibit the inherent structures. In order to show explicitly the loops for the original  $10 \times 10$  patterns (as marked) external connections as added in Figure 5 would often be necessary. Some of the loops cross the  $10 \times 10$  window boundary back and forth, e.g., (a4) and (b5). Most of the loops in the loop patterns are wave loops.

We have seen that a GA can generate patterns of a certain class (such as square/point, loop, defined by a set of tiles). The GA uses a globally defined and tested fitness function, here given by the number of tile matches. It was a kind of surprise that such complex patterns can easily be found by the GA although the search space is very large. The fitness function can easily be modified, for instance, including further conditions like the ratio between zero and one pixels (density), the length of loops, the kind of loops, and so on. Therefore, the GA is a powerful tool for generating patterns under complex global conditions. In the next sections, we want to show that loop patterns can also be generated by testing only local conditions by a CA rule.



**Figure 8.** GA-evolved loop patterns of size  $10 \times 10$  in quad representation. Repeating  $10 \times 10$  structures are marked. (a1–a6) Patterns with one loop. (b1–b5) Patterns with two loops. (b6) Three loops. (c1–c6) Connected loops.

### 3. Generating Loop Patterns with a Probabilistic Cellular Automata Rule

In the following Section 3.1, we want to decide what kind of updating scheme and rule we will use, then, in Section 3.2, how the tiles from the already tile set are incorporated into the CA rule (by templates derived from them), and then in Section 3.3, we will describe the rule that injects noise when a loop pattern is not reached. Simulations follow showing the resulting patterns.

#### 3.1. Choosing the Type of the Rule

The simulation of a CA is very simple, that is one reason why CA are so popular. Nevertheless, there are different types of CA and we need to choose the best-suited one for our application. There are two important properties we have to choose: (i) *synchronous* or *asynchronous* updating, and (ii) a *deterministic* or a *probabilistic* rule.

(i) Synchronous or Asynchronous Updating.

- *Synchronous updating.*

(Phase 1) For every cell  $(x, y)$  its new state  $s'$  is computed by a local rule and buffered in the *new state* variable  $s_{new}$ .

$$\forall(x, y) : s_{new}(x, y) \leftarrow s'(x, y) = f(s(x, y), s(neighbor_1(x, y)), s(neighbor_2(x, y)), \dots)$$

We use  $z \leftarrow v$  to denote that a variable  $z$  (with memory) is used to store the value  $v$ .

(Phase 2) The state of each cell is updated (replaced) by its new state.

$$\forall(x, y) : s(x, y) \leftarrow s'(x, y)$$

It is important to notice that the order in which the cells are processed in phase 1 and in phase 2 does not matter, but the phases must be separated. This model can easily be implemented in software, or in clocked hardware using d-flipflops (internally supplying a master and a slave memory).

- *Asynchronous updating.*

There are several schemes. In the pure case, there are no phases. A cell  $(x, y)$  is selected at random,  $s_{new}$  is computed, and then immediately updated:  $s \leftarrow s_{new}$ .

We want to use the following scheme. A time step  $t \rightarrow t + 1$  is considered as a block of  $N = n_1 \times n_2$  micro time steps  $\tau \rightarrow \tau + 1$ . A selected cell is processed during one micro time step. For each time step, we select the cells sequentially in a different random order (elements are mutually exclusive). During one time step, each cell is updated once, but the order is random. We display CA configurations at time steps. This scheme is called *random sequence* or *random new sweep*.

(ii) Deterministic or Probabilistic Rule. A *deterministic* rule always computes the same next state for a given input (the combined state of all neighbors). A *probabilistic* rule computes different next states with certain associated probabilities for a given input.

Combination. In principle, we can combine *synchronous* or *asynchronous* updating with a *deterministic* or a *probabilistic* rule. This makes four options: (1) synchronous updating and deterministic rule; (2) synchronous updating and probabilistic rule; (3) asynchronous updating and deterministic rule; (4) asynchronous updating and probabilistic rule.

*Option 1:* Until now, it was not possible to design such a rule that can produce loop patterns. The problem is that the evolving patterns may get stuck in non-desired patterns or oscillating structures such as we know from the *Game of Life*. It remains an open question if it is possible to find such a rule.

*Options 2–4:* These options are related because the computation of a new configuration is stochastic. It seems that they can be transformed into each other to a certain extent. We decided to use option 4 (random sequence together with a probabilistic rule) as in our former work [1–3,5]. With asynchronous updating, we do not need buffered storage elements and a central clock for synchronization, which is closer to the modeling of natural processes.

### 3.2. Templates Used for the CA Rule

One of the key ideas is to use templates (local matching patterns) for testing them everywhere (at any point  $(x, y)$ ) in the CA space. If we find a template match we call it a *hit*. A hit means that the cell is covered by a pixel of a tile. We are searching for stable patterns where we have at least one hit everywhere.

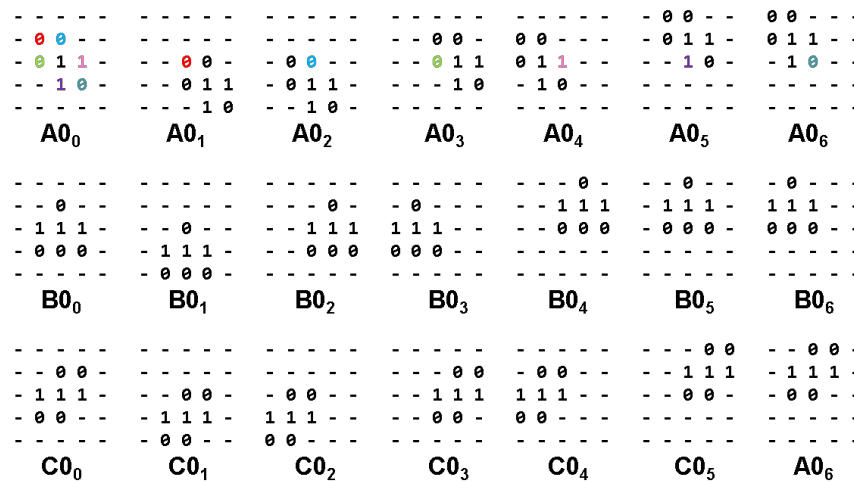
A template is a shifted tile. Let us assume that templates have the size of  $m \times m$  pixels. We have chosen  $m$  to be odd because then it is easier to define the unique center as the origin for the local coordinates and to handle symmetric templates. But this choice is not obligatory as long as all templates fit into the boundaries of  $m \times m$ . We index tile and template pixels relative to their center located at  $(x, y) = (0, 0)$ . We declare x-coordinates to increase rightwards and y-coordinates downwards.

For a given tile (the *generating* tile), we derive a set of templates by shifting the generating tile vertically and/or horizontally. We consider only the valid pixels with a value of 0/1 for the process of deriving the templates. We call a valid pixel *reference pixel* when it is selected for deriving a specific template. For each reference pixel with the (relative) coordinates  $(x_{ref(i)}, y_{ref(i)})$ , we shift the tile in such a way that after shifting, the reference pixel occupies/arrives at the center  $(0, 0)$  of the derived template, i.e.,

- for each reference point  $i$  at  $(x_{ref(i)}, y_{ref(i)})$ :  $A_i = shift(-x_{ref(i)}, -y_{ref(i)}, L)$  where the operator  $shift(\Delta x, \Delta y, L)$  shifts a tile  $L$  by the given offsets, and is shifting-in null pixels and deleting shifted-out pixels.

Figure 9 shows the templates derived from  $A_0$ ,  $B_0$ , and  $C_0$  as defined in Figure 3.  $A_0 = A_{0_0}$  is the generating tile and also the first (the main) template that needs no shift because it contains a valid pixel in its center already (what we assumed). Now, we consider the pixel  $A_0(-1, -1)$ , in green.  $A_0$  is shifted by  $(1, 1)$  steps, (1 rightwards, 1 downwards), which yields  $A_{0_1}$ .  $A_{0_2}$  is  $A_0$  shifted by  $(0, 1)$ , where the red pixel occupies the center.  $A_{0_3}$  is  $A_0$  shifted by  $(1, 0)$ , and so on. As a result, we obtain seven templates derived from  $A_0$ . In a similar way, we obtain seven templates each from  $B_0$  and  $C_0$ . As we have altogether 16 tiles with 7 valid reference pixels we obtain  $16 \times 7 = 112$  templates. We may reduce the number of templates by joining them using classical methods of minimization logical functions where the center value is the output and the neighboring pixels are the inputs.

The templates (to fit in a square array) are larger than the given tile because of shifting, the radius of the templates is twice the radius of the given tile.



**Figure 9.** (A0<sub>0</sub>–A0<sub>6</sub>) The templates derived from tile A0 = A0<sub>0</sub>. (B0<sub>0</sub>–B0<sub>6</sub>) The templates derived from tile B0 = B0<sub>0</sub>. (C0<sub>0</sub>–C0<sub>6</sub>) The templates derived from tile C0 = C0<sub>0</sub>. Templates are shifted tiles.

*Remark.* In general, we can define a template set independently of certain tiles, but this is a general topic that will not be followed here.

### 3.3. The CA Rule

The following rule is able to evolve CA loop patterns. The main idea is to inject noise if no template hit is recognized in order to drive the evolution to a stable pattern. In such a pattern, there is everywhere a template hit, i.e., every CA cell is either equal to the center of a tile or it is a part (a pixel) of a tile, corresponding to the center of a template.

The default value of the new state  $s'$  is the current state  $s$ . All templates  $A_i$  are tested against the current CA neighborhood at the selected site  $(x, y)$  where the template center is excluded from the test. If there is a template hit, the center of the template defines the new state. (Either the CA neighborhood corresponds fully to the template or only the center is false and then it is corrected). Noise is injected under certain conditions including certain probabilities. The main condition is  $C_0$ , which becomes true if there is no hit. The condition  $C_1$  takes toggling uncovered cells of the NESW-neighborhood into account in order to dissolve them. The additional condition  $C_2$  excludes faulty loop patterns.

$$s'(x, y) = \begin{cases} s(x, y) & \text{default no change} & (a) \\ center(A_{i_{lasthit}}) & \exists hit(A_i, (x, y)) & (b) \\ random \in \{0, 1\} & \text{if } C_0 \vee C_1 \vee C_2 & (c) \end{cases}$$

where

- $hit()$  is a test whether a template  $A_i$  matches in the current CA neighborhood of the selected CA cell  $(x, y)$ :

$$hit(A_i, (x, y)) = \begin{cases} 1 & \text{if template } A_i \text{ matches in the CA neighborhood of } (x, y) \\ 0 & \text{else} \end{cases}$$

*Remark.* For specialized rules, it may be useful to store the number of template hits in the state of the cell:  $h(x, y) = \sum_i hit(A_i, (x, y))$ . Certain values of  $h$  were used to inject additional noise [2,5] in order to find a minimal or maximal number of dominos covering a space.

- $center(A_{i_{lasthit}})$  is the value of the center pixel of a template match. Here, the value of the last hit is used, if there are several hits. It would be possible to use the value of the

first hit or that of any hit randomly. To which extent the choice matters is a topic of further research.

- $C_0 = Decision(\pi_0) \wedge (hit = 0)$   
is the main basic condition that drives the evolution through injecting noise if there is no hit, which means that such a cell is not covered by a tile pixel.
  - $Decision(\pi)$  is a function that is TRUE if a trial is successful under probability  $\pi$ , otherwise FALSE. In Python, the function can be defined as: “def decision(probability): return random() < probability”
- $C_1 = Decision(\pi_1) \wedge (sum(NESW, hit = 0) > 0)$   
This additional condition avoids lonely uncovered cells that cannot be destroyed by  $C_1$ . This problem will further be discussed in Section 3.5.
  - $NESW = \{(0, -1), (1, 0), (0, 1), (-1, 0)\}$   
is the North-East-South-West-Neighborhood (relative indexes)
  - $sum(Neighborhood, ConditionForANeighbor) \in \{0..|Neighborhood|\}$   
is a function that tests for each neighbor in the  $Neighborhood$  (set of relative positions of the neighbors) a local condition  $ConditionForANeighbor$  and counts the fulfilled conditions.
- $C_2 = Decision(\pi_2) \wedge (sum(NESW, s = 1) \neq 2)$   
These additional conditions realize the loop path condition. Thereby faulty loops are excluded (see end of Section 3.6).
- $C_2 = Decision(\pi_2) \wedge (sum(NESW, s = 1) < 2)$   
This alternative additional condition allows unconnected and connected loops.

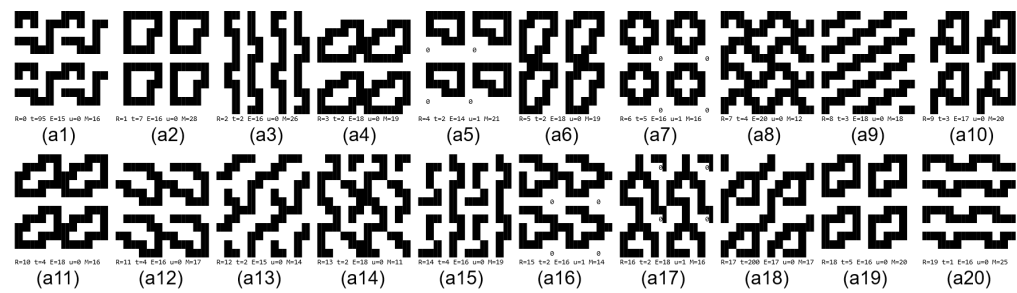
The behavior of the CA rule depends on the probabilities  $\pi_0, \pi_1, \pi_2$ . We can use them as parameters in the general rule:  $Rule(\pi_0, \pi_1, \pi_2)$ . Note that the conditions  $\pi_0, \pi_1 > 0$  have to be fulfilled to drive the evolution to a stable pattern by destroying uncovered cells. The condition  $C_2$  can be deactivated by setting  $\pi_2 = 0$ .

### 3.4. Simulation of the First Rule

The *First Rule* (also called *Basic Rule*) is the  $Rule(\pi_0 > 0, \pi_1 = 0, \pi_2 = 0)$  in which we used the probability  $\pi_0 = 0.08$ . It injects noise only when the own cell shows no hit, i.e., the cell is uncovered. Figure 10 shows the evolved patterns for 20 runs with different random initial conditions. We found different types of patterns:

1. Seven patterns (a2, a3, a5, a7, a9, a19, a20) with unconnected (separated) loops, patterns that we mainly aim at.
2. Four patterns (a4, a6, a8, a11) with loops that are connected, meaning that there are connections between them. It is a matter of definition whether such loops should be within the scope of objectives. In Section 1.2.2 we classed them as faulty patterns.
3. Nine patterns (a1, a10, a12, a13, a14, a15, a16, a17, a18) are faulty loop patterns. Such patterns are not really what we want to achieve.
4. Four patterns (a5, a7, a16, a17) contain an uncovered cell, marked by ‘0’. Such cells are toggling ( $0 \leftrightarrow 1$ ) because of noise injection through condition  $C_0$ . Such patterns can also be of interest but they are partially unstable (cycling). We may wish to exclude them, or we may fill the toggling cell constantly with zero to yield a lower-density loop pattern with less ones.

Although this is only a test case (the number of types and the actual pattern may vary) we can observe that the First Rule does not work perfectly because a faulty loop pattern can appear. Now, we aim at a rule that produces a loop pattern only. First, we want to avoid uncovered toggling cells (type 4 patterns) by the enhanced *Second Rule* described in the next Section 3.5. Then, we want to avoid faulty patterns of type 3 and type 2 (*Third Rule*, Section 3.6).



**Figure 10.** First CA Rule. the patterns evolved by 20 simulation runs. Seven patterns are loop patterns (a2,a3,a5,a7,a9,a19,a20) we aim at. Four patterns show connected loops (a4,a6,a8,a11). The patterns (a5,a7,a16,a17) contain one uncovered cell (marked by ‘0’). Patterns (a1,a10,a12–a18) are faulty loops (not closed, diagonally touching 1-cells, branches). Simulation values: R (Run), t (time step), E (number of 1-cells), u (number of uncovered cells), M (number of tile matches).

### 3.5. Simulation of the Second Rule

The *Second Rule* is the *Rule*( $\pi_0 > 0, \pi_1 > 0, \pi_2 = 0$ ), it was used with the setting (0.08, 0.2, 0). It avoids the toggling of uncovered cells. The problem was solved by injecting additional noise from a neighbor to the current cell if there exists a neighbor that is uncovered (with no hit). We have used the NESW neighborhood for this problem. This is achieved by activating the condition  $C_1$  by setting the probability  $\pi_1 > 0$ . We recall the condition  $C_1 = Decision(\pi_1) \wedge (sum(NESW, hit = 0) > 0)$ . Indeed we could not find such lonely toggling cells in the many simulation experiments. This is not proof that the NESW neighborhood is sufficient in general to solve this problem. In the general case, a larger neighborhood is probably necessary if sophisticated or larger tiles are given.

*Remark.* We may consider a situation where the given tile(s) cannot cover the space. Given a square tile of size  $n \times n$  with a black border of width one around a white square of size  $(n - 2) \times (n - 2)$ . The tile shall be used to cover a field of size  $(n + 1) \times (n + 1)$ . We can easily see that only one such tile can be used (because of overlaying conflicts) and there will always remain  $2n + 1$  uncovered cells. In such a case, it would even be better to use the First Rule in order to reach a partially stable pattern, in this case with one tile only.

### 3.6. Simulation of the Third Rule

The *Third Rule* is the *Rule*( $\pi_0 > 0, \pi_1 > 0, \pi_2 > 0$ ), it was used with the setting (0.08, 0.2, 0.8). In addition, it activates the condition

- (Case 1)  $C_2 = Decision(\pi_2) \wedge (sum(NESW, s = 1) < 2)$ , or
- (Case 2)  $C_2 = Decision(\pi_2) \wedge (sum(NESW, s = 1) \neq 2)$ .

(Case 1) Additional noise is injected if the number of ones in the NESW neighborhood is less than 2. This means that only patterns are allowed in which a one-cell has at least two orthogonal one-neighbors. Thereby faulty type 3 patterns of the First Rule simulation are excluded, but connected loops (type 2 patterns) are still allowed. Many simulations were performed (Figure 11) converging to loop patterns with separated loops or patterns with connected loops having 3- or 4-pin connectors.

(Case 2) This condition is the negation of the loop path condition (a path cell has exactly 2 path cells in NESW). Noise is injected if this condition is not fulfilled. Thereby the evolution is pushed to loop patterns *only* (Figure 12), the patterns we aimed at.

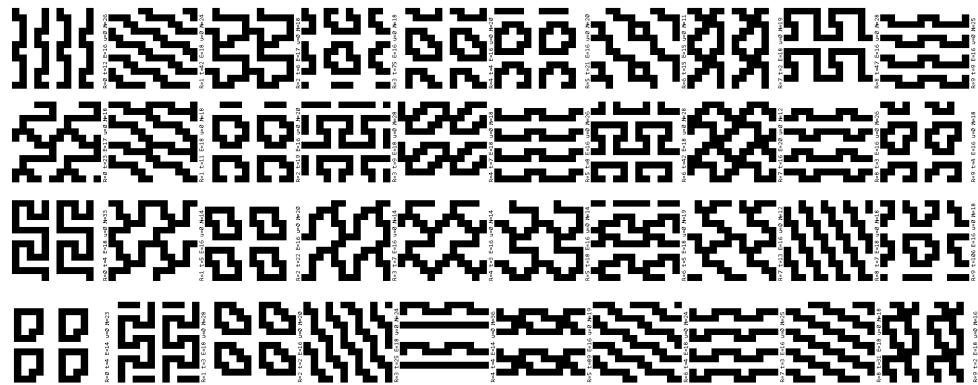


Figure 11. Simulation results of the third CA Rule (Case 1) using the conditions  $C_1, C_2, C_3$ . The evolved patterns are stable and contain unconnected or connected loops.



Figure 12. Only loop patterns are evolved by the Third Rule (Case 2). Noise is injected if a path cell has not two path cells in NESW. Field size is  $12 \times 12$ .

#### 4. Discussion and Future Work

##### 4.1. Forming Loops by the Tiles

We defined a set of overlapping tiles that are able to cover a 2D space in a way that a loop pattern can appear. For the design, it was important that the tiles can connect to each other by overlapping in a way that they can find other suitable tiles in the continuation of a path (during the evolution), and finally that the beginning and end of a path connect with each other and form a loop.

The driving force in the GA is the number of matches. Let us consider an *almost closed loop* (or we may call it a *slightly opened loop*), just one path cell is missing (is zero). Compared to the (closed) loop there are fewer matches because the endings are “free”, not connected to tiles that could be used to close the gap. Then, the GA closes the gap because it tries to maximize the number of matches through using more tiles.

The CA rule works differently because it tries to avoid uncovered cells. In the case of an almost closed loop, there are uncovered cells around the gap (if the tiles are properly designed). The gap is then closed by evolution; finally, all cells are covered, no more noise is injected and the pattern remains stable. These are only rough explanations that have to be detailed and improved in a further work.

##### 4.2. The Genetic Algorithm Generating Loops

A GA was developed that can find optimal loop patterns based on the set of tiles. It tries to maximize the number of tile matches. Some of the patterns are faulty (e.g., touching black cells on the diagonal, loops having branches). This means that the designed tiles are not able to produce loop patterns only but also faulty ones. Instead of improving the tiles to make them proper in a way that loop patterns are evolved only, this issue was left to the CA rule (with condition  $C_2$ ). Nevertheless, it is a research issue to find a proper set of tiles that enforce the forming of loop patterns only.



#### 4.3. The Cellular Automata Rule Generating Loops

A CA Rule was defined that tests templates derived from the defined tiles. In the first version (First Rule, Second Rule) it can easily evolve loop patterns or faulty ones. The extra logical condition  $C_2$  can exclude the faulty ones (Third Rule). This condition can easily be modified in order to allow connections between loops. A difficulty during the design and simulations was to make the rule simple and reliable. The used settings for the probabilities  $\pi_0, \pi_1, \pi_2$  were good working, but there is potential for optimizing them in order to minimize the average time for finding a stable pattern. For small problem sizes, the CA rule converges fast, but for large sizes, we expect an exponential increase, as already tested for domino patterns in [2]. In order to reduce the computational time, the CA rule programming code can be optimized or other techniques (e.g., divide and conquer) could be involved.

#### 4.4. Future Work

There are several open questions.

- What are (all) possible loop patterns depending on the size of a field, and what is the distribution of different loop types with certain properties?  
A special question could be: How many loops (optionally restricted to a certain type) can maximal be placed in a certain field.
- Can it be proven that the presented CA rule always (for large field sizes) produces stable loop patterns? What are the probabilities that certain loop types are evolved? What happens for large fields?  
For this mainly experimental work, the minimal loop pattern size is  $2 \times 2$ , and we see no limit for large sizes. For large sizes, we expect a mixture of different loops and loop sizes as in Figure 12. The CA rule always generated stable loop patterns so far, but the computation time increases exponentially. In the simulation, rectangular fields could be handled too. Nevertheless, the convergence and limits of this approach should be proved and further evaluated.
- How do tile matches lead to loops? The tiles were designed in a way that corners and lines can easily connect/overlap to a loop that is surrounded by zeroes. The tile matches are maximized, thereby each cell tries to obtain a match, and the idea used is that loops are paths with a maximal number of matches. Until now, the defined set of tiles did not always produce loop patterns, and therefore the CA rule needed an additional local condition (the path condition). This issue has to be further investigated, and it is the aim to find a proper set of tiles that securely stimulates the generation of loop patterns.
- How can optimal loops be generated on the basis of local conditions only when optimality depends on a global measure. Possible parameters for a global measure are:
  - The space between loops;
  - The number of loops;
  - The length of the loops;
  - The type of the loops (plane, straight, wave).

We can conclude that we found a way to generate loop patterns using local conditions only, but there are several open questions to be answered in further research.

**Funding:** This research received no external funding.

**Data Availability Statement:** All relevant data is contained within the article.

**Acknowledgments:** The author is grateful to the two anonymous reviewers for their helpful comments and suggestions. We acknowledge the Institutional Open Access Program support by the Technical University of Darmstadt.

**Conflicts of Interest:** The author declares no conflict of interest.

## References

1. Hoffmann, R. Forming Point Patterns by a Probabilistic Cellular Automata Rule. *arXiv* **2022**, arXiv:2202.06656.
2. Hoffmann, R.; Désérable, D.; Seredyński, F. A cellular automata rule placing a maximal number of dominoes in the square and diamond. *J. Supercomput.* **2021**, *77*, 9069–9087. [[CrossRef](#)]
3. Hoffmann, R.; Désérable, D.; Seredyński, F. Cellular automata rules solving the wireless sensor network coverage problem. *Nat. Comp.* **2022**, *21*, 417–447. [[CrossRef](#)]
4. Hoffmann, R. How Agents Can Form a Specific Pattern. In *International Conference on Cellular Automata*; Springer: Cham, Switzerland, 2014; pp. 660–669.
5. Hoffmann, R.; Désérable, D.; Seredyński, F. Minimal Covering of the Space by Domino Tiles. In *Parallel Computing Technologies*; Malyskhin, V., Ed.; PaCT 2021; LNCS 12942; Springer: Cham, Switzerland, 2021; pp. 453–465.
6. Holland, J.H. Genetic algorithms. *Sci. Am.* **1992**, *267*, 66–73. [[CrossRef](#)]
7. Holland, J.H. Genetic algorithms. *Scholarpedia* **2012**, *7*, 1482. [[CrossRef](#)]
8. Beasley, D.; Bull, D.R.; Martin, R.R. An overview of genetic algorithms: Part 1, fundamentals. *Univ. Comput.* **1993**, *15*, 56–69.
9. Mitchell, M. Genetic algorithms: An overview. In *Complex*; Wiley: Hoboken, NJ, USA, 1995; Volume 1, pp. 31–39.
10. Grünbaum, B.; Shephard, G.C. *Tilings and Patterns*; Courier Dover Publications: Mineola, NY, USA, 1987.
11. Janin, D. On languages of one-dimensional overlapping tiles. In *SOFSEM 2013: Theory and Practice of Computer Science: 39th International Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, 26–31 January 2013*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 244–256.
12. Goumas, G.; Sotiropoulos, A.; Koziris, N. Minimizing completion time for loop tiling with computation and communication overlapping. In Proceedings of the 15th International Parallel and Distributed Processing Symposium, IPDPS 2001, San Francisco, CA, USA, 23–27 April 2001; IEEE: Piscataway, NJ, USA, 2001.
13. Waychunas, G.A. Structure, aggregation and characterization of nanoparticles. *Rev. Mineral. Geochem.* **2001**, *44*, 105–166. [[CrossRef](#)]
14. Peano, G. Sur une courbe, qui remplit toute une aire plane. *Math. Ann.* **1890**, *36*, 157–160. (In French) [[CrossRef](#)]
15. Hilbert, D. Ueber die stetige Abbildung einer Linie auf ein Flächenstück. *Math. Ann.* **1891**, *38*, 459–460. (In German) [[CrossRef](#)]
16. Prusinkiewicz, P.; Lindenmayer, A.; Fracchia, D. Synthesis of space-filling curves on the square grid. In *Fractals in the Fundamental and Applied Sciences*; Peitgen, H.-O., Henrique, J.M., Pencdo, L.F., Eds.; Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1991.
17. Crossley, M. Hamiltonian Cycles in Two Dimensional Lattices. In *Statistical Physics in Biology*; Springer: Berlin/Heidelberg, Germany, 2013.
18. Keshavarz-Kohjerdi, F.; Bagheri, A. Hamiltonian paths in some classes of grid graphs. *J. Appl. Math.* **2012**, *2012*, 475087. [[CrossRef](#)]
19. Chas, M.; Sullivan, D. String topology. *arXiv* **1999**, arXiv:math/9911159.
20. Wolfram, S. *A New Kind of Science*; Wolfram Media: Champaign, IL, USA, 2002; Volume 5.

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.