University of Ljubljana
Faculty of Computing and Informatics

Iztok Jeras

Preimages of 2D Cellular Automata

Master's thesis
At university study

Mentor: Prof. Dr. Branko Šter

Ljubljana, 2016

ABSTRACT

While computing preimages of 1D cellular automata is a well researched and documented problem, for 2D cellular automata there is less research available. For the 1D problem we know algorithms for counting and listing preimages where computational complexity is a linear function of the size of the problem. It is possible to determine whether a 1D cellular automaton is reversible, and what is the Garden of Eden sequence regular language. For the 2D problem we know a few algorithms, but they are poorly theoretically researched. We also know that the reversibility problem is in general undecidable for 2D cellular automata.

The preimage network, first developed for 1D cellular automata, was proved to be a useful tool for explaining algorithms and for constructing proofs. Here I explain how to construct the preimage network for 2D cellular automata. While for the 1D problem this network is a normal graph, for 2D it was extended into the third dimension. Preimages are transformed from paths in the graph in 1D into surfaces on the network in 2D. Edge conditions are transformed from weights for vertices ending a path in the 1D problem into weights for the closed path around a preimage surface in the 2D problem.

While developing the algorithm, it proved impossible to count preimages of 2D cellular automata with processing requirements growing linearly with problem size. Instead, processing requirements grow exponentially with the size in one of the dimensions. The described algorithm does not differ much from the existing ones. The cellular automaton is split into rows, the preimage list is first determined for each row from the first to the last. The row results are then combined into the result for the whole 2D problem. In a similar fashion to the 1D approach, the algorithm splits into two passes. In the first pass preimages are counted, in the second optional pass preimages are listed. The second pass is performed in the opposite direction, while rows are also observed from the opposite side.

I see the main advantage of the described algorithm in using existing solutions for row processing. Solutions proved to be effective in solving the 1D problem. Using progressive encoding of intermediate solutions also enables reducing memory consumption.

Key words:

Cellular automata, preimages, predecessors, ataviser, computational complexity, reversibility, Garden of Eden, Conway's Game of Life, trid, quad

LIST OF ABBREVIATIONS AND SYMBOLS USED

1D	one-dimensional
2D	two-dimensional
3D	three-dimensional
CA	cellular automata
GoL	Conway's Game of Life
GoE	Garden of Eden
trid	CA environment consisting of three cells on a hexagonal grid
quad	CA environment consisting of four cells on a square grid
DFS	depth first search
SAT	boolean satisfiability problem

C	arbitrary constant
S	set of cell states
$\|S\|$	number of possible cell states
c	state of an individual cell (integer value)
(x, y)	state of the cell at the coordinates (x, y) within the 2D field of the cells
c^t	state of the cell in the present
c^{t+1}	future state of the cell (one step)
N_x	size of a rectangular 2D field of cells in dimension X
N_y	size of a rectangular 2D field of cells in dimension Y
N	number of cells in a 1D or 2D array
M_x	size of the rectangular 2D cell surroundings in dimension X
M_y	size of the rectangular 2D cell surroundings in dimension Y
M	number of cells in the 1D or 2D environment

n	state of the environment of an individual cell (integer value)
(x, n)	state of the cell environment at the coordinates (x, y) within the 2D field
$n^{t^{-1}}$	state of the cell environment in the past (one step)
n^t	state of the cell environment in the present
f	transition function that defines the temporal evolution of an automaton
f^{-1}	inverse transition function
$o_{\leftrightarrow}$	overlapping surroundings in dimension X
$o_{\updownarrow}$	overlapping surroundings in dimension Y
$o_{\times}$	overlapping surroundings in the diagonal direction

CHAPTER 1

INTRODUCTION

1.1 Motivation

1.1.1 Cellular automata as a model of the universe

Since any universal system can model any other universal system, we can assume that we can model the universe with universal cellular automata (CAs). The modeling of the universe itself is still beyond our reach, and it attempts to at least bring CA theory closer to theoretical physics. From the point of view of information theory and thermodynamics, the most interesting model is gravity (as an entropic gravity) [42], which assume that the 3D universe is a projection of processes that take place on a 2D plane. On the other hand, CAs also allows the observation of abstract copying of information (replication) and evolution [39]. Both are important information phenomena in our universe.

1.1.2 Information dynamics

CA information dynamics is most often described only as reversible or irreversible. There are also some articles that observe the entropy of the system. With a reversible machine, all information is retained. Each state has exactly one future and one past. For a reversible cellular automaton, we can also define a rule or function that simply processes the automaton into the past instead of the future. For reversible CAs, therefore, pre-image (past) computation is trivial and does not require the algorithm described here. For irreversible CAs, such a reverse rule does not exist. The current state can have no, one, or more previews. The described algorithm is intended for counting and printing previews for a given current state of CA. We say that such automata lose information over time because we cannot uniquely determine the past state.

It is also common to observe particle dynamics in Game of Life (GoL) [7] (Figure 1.1) and Elementary Rule 110 [12] (Figure 1.2). Pistols, particle collisions, and similar constructs exist in both GoL and elementary rule 110, and in general for any universal cellular automaton. Guns (Figure 1.1) emit particles, the origin and emitted particles together grow to infinity. This increases the information needed to describe the system. Figure 1.2 shows two different interactions between two particles. In the left case, the two particles meet, share a common space for a while, and then continue on their own. In the right case, the particles meet and a new particle is formed after the collision. Many professional and amateur researchers study particles in GoL and their dynamics. With the help of basic building blocks, more complex systems can be constructed, the most interesting of which are the Turing machine [38] and the universal constructor [20].

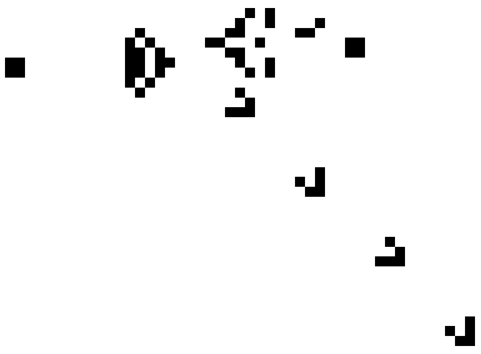
Figure 1.1: Gosper's gun with a few skates fired.

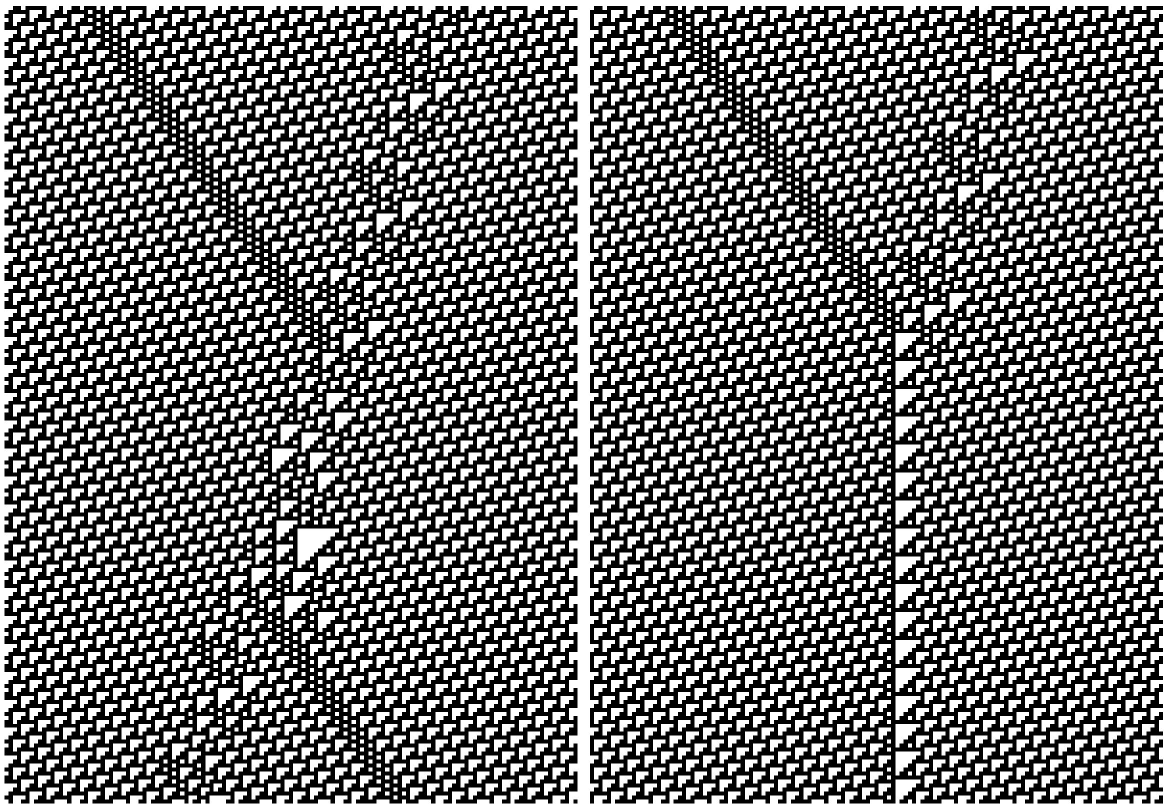
Figure 1.2: Particle pair collision in rule 110. The difference in the initial state between the images is in the distance (phase) between the particles.

Cellular Automata were conceived by mathematicians John von Neumann and Stanislaw Ulam to be able to formulate a theory of a universal constructor [13] capable of making a copy of itself from a tape recording. Christopher Langton took part of the universal constructor and simplified it into a simple loop [11] that copies itself based on information written inside the body (Figure 1.3).

```
  2 2 2 2 2 2 2 2
2 1 7 0 1 4 0 1 4 2
2 0 2 2 2 2 2 2 0 2
2 7 2         2 1 2
2 1 2         2 1 2
2 0 2         2 1 2
2 7 2         2 1 2
2 1 2 2 2 2 2 2 1 2 2 2 2 2
2 0 7 1 0 7 1 0 7 1 1 1 1 1
  2 2 2 2 2 2 2 2 2 2 2 2 2
```

Figure 1.3: Langton loop. Numbers are the states of an individual cell. Below right from the loop grows a new loop.

The examples listed above somehow describe the information dynamics in CA. However, there is no general theory of information dynamics in CA that would describe dynamics quantitatively and spatially. In my article [30] and conference papers [25, 27, 29], I graphically depicted pre-images of the current situation for the 1D problem. From the representation (Figure 1.4) it can be seen that in some places more information is lost than elsewhere, which indicates the possibility of deriving a general theory of information dynamics. Unfortunately, this possibility has not yet materialized. Similarly, it is possible to graphically represent pre-images of 2D CA, and to infer from the graphs the loss of information in 2D CA.

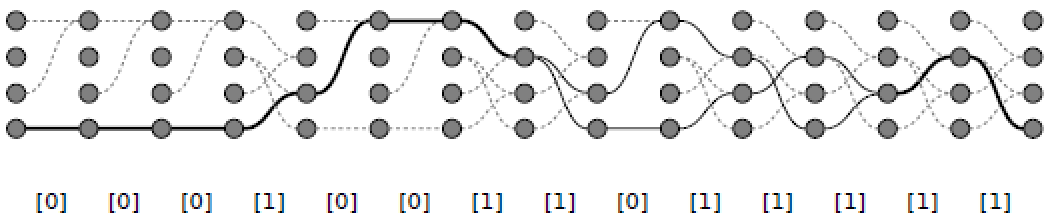

Figure 1.4: 1D CA pre-grid (Rule 110) for silent background with cyclic sequence 00010011011111. Each of the two highlighted paths between left and right represents a pre-image. In one part the prefaces are the same, in the other they are different. As a layman, we can conclude that where the previews differ, 1 bit of information is lost.

1.1.3 Attractor trough

An important tool for the analysis of temporally and spatially discrete dynamical systems is the basin of attraction (Figure 1.5). Andrew Wuensche has been studying attractor troughs of random binary networks and cellular automata for years [46, 47]. This is a graph where the nodes of the state are cyclic finite CA, and the directed paths connect each state with its time successor. All states can be collected in one or more troughs, depending on the system. In any CA that is not locally inject-able, pre-imaging states called Garden of Eden (GoE) occur [34, 35]. GoE states are sheets in a trough graph.

The attractor itself is a cycle in the trough graph. This is the time periodic state in which the final temporal evolution of each finite discrete dynamical system takes place. A cycle can contain one or more states, the number of which is called the attractor period. The attractor also represents stable objects in an infinite CA. For GoL, there are catalogs of such objects, where they are arranged by size, period, and speed of movement. As an example, I would cite a quiescent background (period 1) and a glider (period 4).

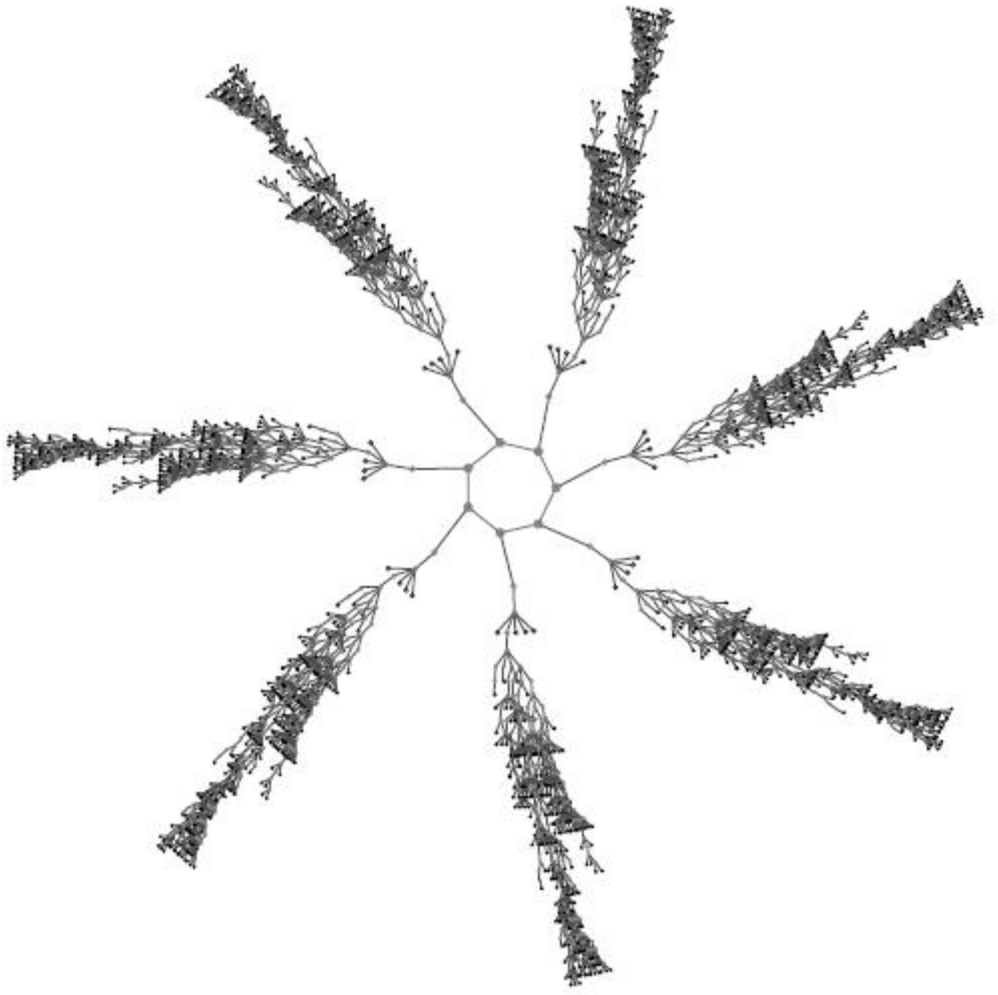

Figure 1.5: Attractor trough for elemental rule 110 and configuration 00010011011111 inside the attractor. The trough contains the central cycle of the attractor and the trees that grow from the cycle and end in leaves (GoE states). The image was taken with the program www.ddlab.com.

1.2 Algorithms for counting and printing CA previews

So far, I have developed advanced algorithms for counting and printing previews of 1D CA [30]. Throughout history, such algorithms have advanced so that their computational complexity and descriptive/implementation complexity has declined:

1. *Brute force* algorithms [6] have a complexity of $O(C^N)$;
2. *Depth-first search* algorithms (DFS) [9], *Backtracking* algorithms [4], algorithms that solve the *Boolean satisfiability problem* (SAT) [5];
3. Optimal algorithms, where printout and counting are strictly separated.

The essence of the optimal algorithm is that the pre-image counting process is optimal. For 1D CA, this is achieved because the counting complexity is linearly dependent on the size of the problem (N) (N is the number of cells observed). I started researching the 2D CA pre-counting algorithm with the assumption that it is also possible to perform pre-counting with linear complexity here. It turns out that the 2D problem is not that simple. Examples are presented from which it can be seen that an algorithm with linear complexity cannot correctly describe all situations. The complexity of the described algorithm grows exponentially with the size of one of the dimensions of the field CA (C^{Nx}), and linearly with the size of the other dimension (N_y) (C is a constant depending on the number of cell states and the size of the environment, but not on the field size). However, since I have not proved the optimality of the described algorithm for counting, I allow the possibility that there is an algorithm with a lower complexity.

The preview process always has an exponentially growing component. As the number of all states of the system grows exponentially with the size of the problem, consequently the average number of pre-images through all states grows exponentially. The optimal algorithm for printing previews depends on the number of previews linearly in time and memory. My 1D and 2D CA pre-print algorithms are optimal and allow pre-print previews as a result of a detailed analysis of the counting problem.

Other known algorithms fall into the second category. For the most part, they only use local knowledge of the system (local pre-counting), but not global counters. Therefore, they go blind and have to return to the previous state or abandon inappropriate solutions to which they have already allocated process time.

The pre-image grid plays an important role in the development of the described algorithm. It is a graphical representation of a problem aimed at making it easier to understand the problem and the solution. The basis for creating a network of pre-images is De Bruijn diagrams. A single De Bruijn diagram is a grid of pre-images of a single cell. The grids of the individual cells are then connected so that they finally describe the entire field of cells.

McIntosh and his students were the initiators of the use of De Bruijn diagrams for 1D CA analysis [33]. They were used to count and print previews and to analyze particles and their interactions. Paulina Léon and Genaro Martínez (McIntosh's student) [32] are the pioneers of applying De Bruijn's diagrams to 2D CA, but they do not use them to count and print previews.

Most of the research in the field of 2D CA pre-images was done with the aim of finding GoE states in the GoL automaton. From the point of view of the pre-counting algorithm, the GoE state is a state whose number of pre-images is zero. The pre-image counting algorithm can be converted to a less demanding algorithm to check whether a given state is a GoE state by converting operations on integers into logical operations on Boolean states.

1.2.1 Algorithm implementation

The algorithm is implemented as a computer program in C language [26]. The GMP library is used to record integers larger than 64 bits. In addition to the algorithm for counting and printing previews, I also prepared a tool for simulation of binary CA with quad environment [28], based on a simulator for GoE [45]. The simulator can be started on the Internet browser.

1.2.2 Examples and illustrations

Most examples and illustrations describe a binary 2D CA with a small quad environment (four cells per square field, Toffoli 2008 [41]). Most examples and illustrations for GoL in general would be too complex and opaque due to the $2^{3x3} = 512$ possible environmental conditions (see Appendix A). In contrast, the binary quad environment has only $2^{2x2} = 16$ possible states.

I wanted to use a certain rule for the quad environment, which would allow for interesting particle dynamics, but there is not much research in this area yet. There is only evidence of universality for a binary automaton with a trid environment (Powley 2008 [37]).

Isometric projection is used in the images, because for the purposes of analysis, a third dimension is added to the basic 2D field, which describes the space of the pre-images (pre-image grid).

1.3 Content overview

In Chapter 2, I give the definition of 1D and 2D CA. In Chapter 3, I describe the construction of a 2D CA preview grid and explain its relationship to the preset image set for a given configuration. In Chapter 4, I describe a newly developed algorithm for counting and printing previews and explain why an algorithm with linear complexity is not possible. In Chapter 5, I review the existing algorithms and compare them with my algorithm in Chapter 6. Here I also list the contribution of this master's thesis to the field of cellular automata and list some problems that I would like to explore in the future. In Appendix B I give an example of how the algorithm works and in Appendix D I give the source code of the algorithm.

CHAPTER 2

DEFINITION OF CELLULAR AUTOMATA

In this part, we limit ourselves to cellular automata, which are:

- Spatially discrete (cells),
- Time discrete (step),
- Homogeneous (the rules are the same for all cells and all cells are updated at once), and
- Deterministic (the new state depends only on the current state).

Each cell has a discrete value c from the set of cell states S. Balances are numbered:

$c \in S$ in S = {0, 1, ..., $|S|$ - 1} (2.1)

For the purposes of implementing the algorithm, it is important that we are able to index the state of CA elements, such as the rule, the value of the environment, and the like. The method used was basically intended to write the rule. In general, the values of cells (i) are stacked in the series A = [(0), (1), ..., $c(|A|$ - 1)] of length $|A|$, interpreted as $|A|$ a positive integer num in the S-digit number:

$$num = \sum_{i=0}^{i=|A|-1} |S|^{i}.c(i) \qquad (2.2)$$

For the given examples, binary CA is used where S = 2, so that the numbers are in binary composition.

2.1 Definition of 1D cellular automata

The definition of 1D CA is summarized according to [30]. For 1D CA, cells are grouped into a regular array. In general, the length of a string can be infinite, but finite strings of length N are more common. The strings are written in lowercase Greek letters α or β, where (x) is a cell on the coordinate x:

α = ((0), (1), ..., $c(N$ - 1)] (2.3)

2.1.1 Surroundings

For ease of implementation, the cell environment expands only in the positive direction (Figure 2.1). Thus, for a cell (x), its neighborhood of size M is equal to (x) = {(x), $c(x$ + 1), ..., $c(x$ + M− 1)}. In practice, the given definition of the environment differs from the more usual centered environment only in how the time sequence of states is aligned in the graphical representation. I used a somewhat unusual definition because it does not need a separate interpretation for odd-size environments. At the same time, I thus avoided the use of negative numbers in the implementation of the algorithm.

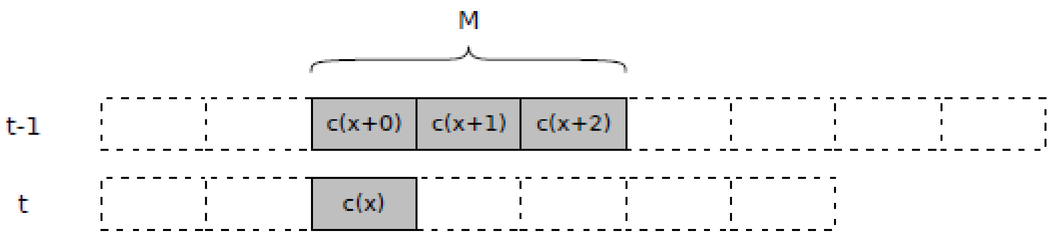

Figure 2.1: Surrounding of 1D CA of size M = 3.

The neighborhood index (x) and the set of possible neighborhoods are:

$$n(x) = \sum_{i=0}^{i=M-1} |S|^i . c(x+i) \tag{2.4}$$

$$n \in \{0, 1, \ldots, |S|^M - 1\} \tag{2.5}$$

2.1.2 Overlapping surroundings

The given neighborhood (x) of $M - 1$ cells overlaps with the surroundings of neighboring cells (Figure 2.2). On the left we have the overlap $o_{\leftarrow}(x)$ between the neighborhoods $n(x$ - 1) and $n(x)$. On the right we have the overlap $o_{\rightarrow}(x)$ between the neighborhoods $n(x)$ and $n(x$+1).

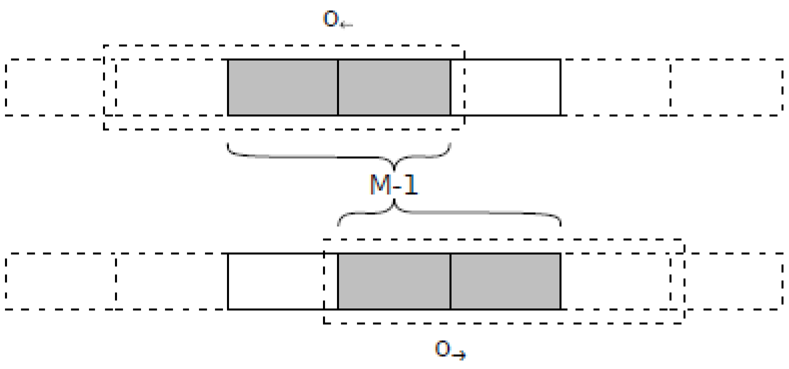

Figure 2.2: Overlap of 1D CA environments of size M - 1 = 2.

The indices of the left overlap $o_{\leftarrow}(x)$ and the right overlap $o_{\rightarrow}(x)$ and the set of possible overlaps are:

$$o_{\leftarrow}(x) = \sum_{i=0}^{i=M-2} |S|^i . c(x+i) \tag{2.6}$$

$$o_{\rightarrow}(x) = \sum_{i=1}^{i=M-1} |S|^i . c(x+i) \tag{2.7}$$

$$o \in \{0, 1, \ldots, |S|^M - 2\} \tag{2.8}$$

2.1.3 Local transition function and rule

The mapping of the current environment (x) to the future parallel cell $c^{t+1}(x)$ is defined by the transition function f, which assigns a cell value to each environment value:

$$c^{t+1}(x) = ((x)) \qquad (2.9)$$

For the purpose of searching for pre-images, the inverse function f^{-1} is interesting, which, given the state of the current cell (x), returns the set of neighborhoods $n^{t-1}(x)$ that are mapped to this value:

$$f^{-1}(c^t(x)) = \{n^{t-1}(x) \in S^m \mid f(n^{t-1}(x)) = c^t(x)\} \qquad (2.10)$$

The transition function can be defined by the rule r. The rule is the index of the selected function within the whole set of $|S|^{|S|^\wedge M}$ functions. It is defined as an integer in a S-digit number structure, where the digits are a sequence of values of the cells into which the transition function maps each of the $|S|^M$ possible environments:

$$r = \sum_{n=0}^{n=|S|^M - 1} |S|^n . f(n) \qquad (2.11)$$

$$\mathrm{r} \in \{0, 1, \ldots, |S|^{|S|^\wedge \mathrm{M}} - 1\} \qquad (2.12)$$

The global transition function is often written as a transition table indicating the value of the transition function (n) for each neighborhood n.

2.1.4 Boundary conditions

An infinite set of cells has no edge and we do not need a definition of boundary conditions for it. At the same time, in practice, we cannot count to infinity, so we consider finite arrays of cells. If we consider the observed finite string to be part of an infinite string, then we need to define how we will treat the cell values outside the observed string. Only an open boundary condition is described here, where cells outside the boundary are not defined, or can occupy any value. For a set of length N, therefore, only N - (M - 1) neighborhoods are defined.

Another common boundary condition is cyclic, where a set of cells is a closed loop. The cyclic coordinate of cell $x_{\circlearrowleft}$ is calculated from x modulo N:

$$x_{\circlearrowleft} = x \bmod N \qquad (2.13)$$

2.1.5 Global transition function

The global transition function F uses the local transition function $f(n)$ to map each neighborhood $n^t(x)$ from a set of cells α^t at time t to a cell $c^{t+1}(x)$ from a set of cells β^{t+1} at time t + 1:

$$\beta^{t+1} = (\alpha^t) \qquad (2.14)$$

The global transition function for the input string of length N and for the open boundary condition returns the output string of length N - (M - 1), while for the cyclic boundary condition returns the output string of length N.

2.1.6 Pre-images

Each pre-image $\beta^{t^{-1}}$ of length N + M - 1 of a given set α^t of length N must satisfy two conditions:

1. For each cell $c^t(x)$, the environment $n^{t^{-1}}(x)$ must correspond to the inverse transition function f^{-1}:
$$\forall x : c^t(x) = f(n^{t-1}(x)) \qquad (2.15)$$
2. Each pair of neighborhoods $\{n^{t^{-1}}(x), n^{t^{-1}}(x+1)\}$ must match in the overlap:
$$\forall x : o^{t-1}(x) = o_{\rightarrow}^{t-1}(x) = o_{\leftarrow}^{t-1}(x+1) \qquad (2.16)$$

Observing the overlap of surroundings is a major element of pre-image search algorithms.

2.2 Definition of 2D cellular automata

For 2D CA, cells are grouped into a regular 2D array. The grid of the field can be rectangular, hexagonal or even quasicrystalline. Here we will confine ourselves to a rectangular grid. In general, the size of a field can be infinite, but more common are finite fields, defined as a rectangle of size $N_x \times N_y$. The total number of cells in the final field is $N = N_x$ x N_y.

For the purpose of indexing CA elements, the plots are divided into rows, and then the rows are assembled into a string that is interpreted as a number. The numbers in the number follow from bottom left to top right within the surroundings (Figures 2.3 and 2.4).

2.2.1 Surroundings

The future state of a cell (x, y) with coordinates (x, y) depends on the current state of the corresponding environment (x, y) (Figure 2.5). We will also limit the shape of the surroundings to a rectangle of size $M_x \times M_y$. The number of cells in the environment is M = M_x x M_y. The neighborhood index (Figures 2.3 and 2.4) and the set of possible neighborhoods are:

$$n(x,y) = \sum_{\begin{cases} i=0 \\ j=0 \end{cases}}^{\begin{cases} i = M_x - 1 \\ j = M_y - 1 \end{cases}} |S|^{M_x j+1} . c(x+i, y+j) \qquad (2.17)$$

$$\mathrm{n} \in \{0, 1, \ldots, |S|^{Mx \cdot My} - 1\} \qquad (2.18)$$

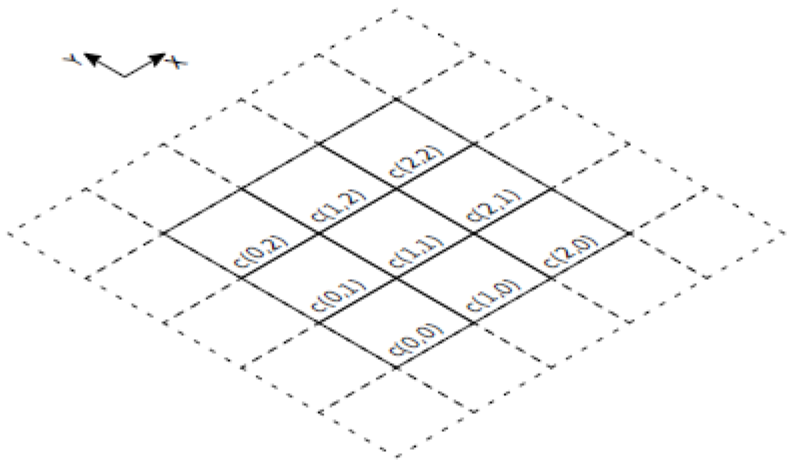

Figure 2.3: Indexing of the cell environment with dimensions M_x = M_y = 3.

It is generally thought that the cell is aligned with the center of its surroundings. Here, however, the cell is aligned in the lower left corner of its surroundings. This allows the same definition to be used for even-sized barrels and eliminates the need to use negative numbers in the algorithm.

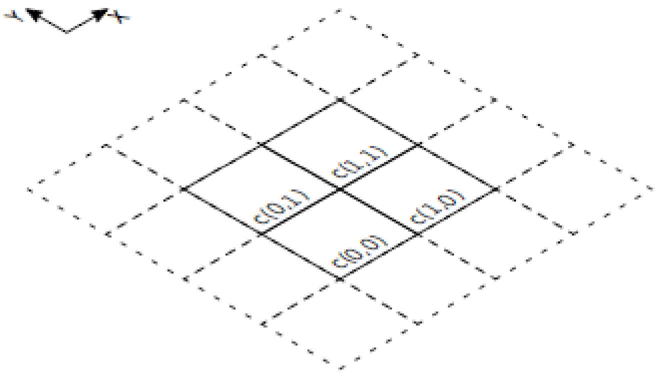

Figure 2.4: Indexing of the cell environment with dimensions M_x = M_y = 2.

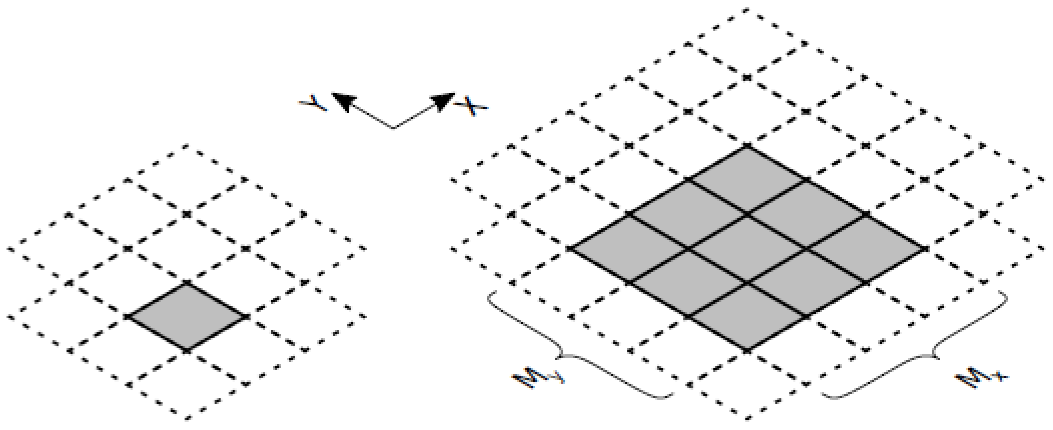

Figure 2.5: Cell (x, y) and the corresponding surroundings (x, y) with dimensions $M_x = M_y = 3$.

2.2.2 Overlapping surroundings

The perimeters in the X dimension overlap by a plot of size $(M_x - 1) \times M_y$ (Figures 2.6 and 2.7). This when indexing gives a set of:

$$o_{\leftarrow}(x,y) = \sum_{\begin{cases} i=0 \\ j=0 \end{cases}}^{\begin{cases} i=M_x-2 \\ j=M_y-1 \end{cases}} |S|^{(M_x-1)j+1} . c(x+i, y+j) \quad (2.19)$$

$$o_{\rightarrow}(x,y) = \sum_{\begin{cases} i=1 \\ j=0 \end{cases}}^{\begin{cases} i=M_x-1 \\ j=M_y-1 \end{cases}} |S|^{(M_x-1)j+1} . c(x+i, y+j) \quad (2.20)$$

$$o_{\leftrightarrow} \in \{0, 1, \ldots, |S|^{(Mx-1)\cdot My} - 1\} \quad (2.21)$$

The encirclements overlap in the Y dimension for a surface of size $M_x \times (M_y - 1)$ (Figures 2.6 and 2.7). This when indexing gives a set of:

$$o_{\downarrow}(x,y) = \sum_{\begin{cases} i=0 \\ j=0 \end{cases}}^{\begin{cases} i=M_x-1 \\ j=M_y-2 \end{cases}} |S|^{M_x j+1} . c(x+i, y+j) \quad (2.22)$$

$$o_{\uparrow}(x,y) = \sum_{\begin{cases} i=0 \\ j=1 \end{cases}}^{\begin{cases} i=M_x-1 \\ j=M_y-1 \end{cases}} |S|^{M_x j+1} . c(x+i, y+j) \quad (2.23)$$

$$o_{\updownarrow} \in \{0, 1, \ldots, |S|^{Mx\cdot(My-1)} - 1\} \quad (2.24)$$

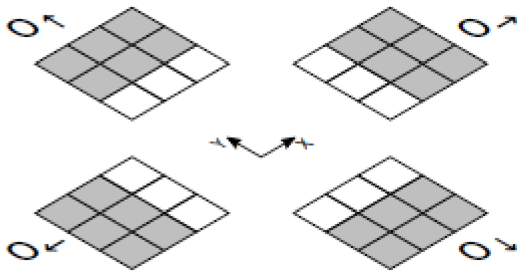

Figure 2.6: Overlap of the surroundings of adjacent cells in the direction of dimensions X and Y, for the size of the surroundings Mx = My = 3. The neighborhoods overlap in 6 cells out of 9.

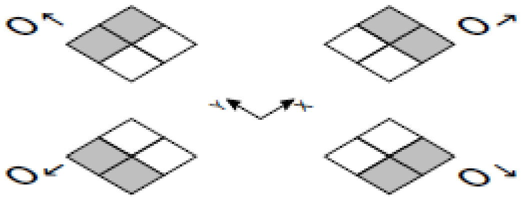

Figure 2.7: Overlap of the neighborhoods of adjacent cells in the direction of dimensions X and Y, for the size of the neighborhood Mx = My = 2. The neighborhoods overlap in 2 cells out of 4.

The circles overlap diagonally by a plot of size (M_x - 1) × (M_y - 1) (Figures 2.8 and 2.9). This, when indexed, gives a set of:

$$o_{\swarrow} = \sum_{\begin{cases} i=0 \\ j=0 \end{cases}}^{\begin{cases} i = M_x - 2 \\ j = M_y - 2 \end{cases}} |S|^{(M_x-1)j+1} . c(x+i, y+j) \tag{2.25}$$

$$o_{\searrow} = \sum_{\begin{cases} i=1 \\ j=0 \end{cases}}^{\begin{cases} i = M_x - 1 \\ j = M_y - 2 \end{cases}} |S|^{(M_x-1)j+1} . c(x+i, y+j) \tag{2.26}$$

$$o_{\nwarrow} = \sum_{\begin{cases} i=0 \\ j=1 \end{cases}}^{\begin{cases} i = M_x - 2 \\ j = M_y - 1 \end{cases}} |S|^{(M_x-1)j+1} . c(x+i, y+j) \tag{2.27}$$

$$o_{\nearrow} = \sum_{\begin{cases} i=1 \\ j=1 \end{cases}}^{\begin{cases} i = M_x - 1 \\ j = M_y - 1 \end{cases}} |S|^{(M_x-1)j+1} . c(x+i, y+j) \tag{2.28}$$

$$o_{\times} \in \{0, 1, ..., |S|^{(Mx - 1) \cdot (My - 1)} - 1\} \tag{2.29}$$

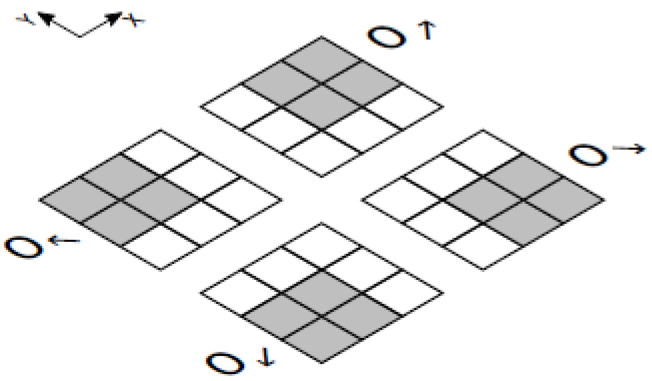

Figure 2.8: Overlapping the surroundings of adjacent cells in the diagonal direction, for the size of the surroundings Mx = My = 3. The neighborhoods overlap in 4 cells out of 9.

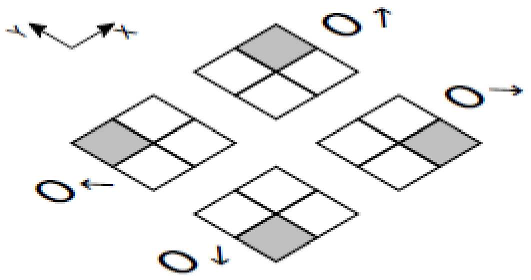

Figure 2.9: Overlapping the surroundings of adjacent cells in the diagonal direction, for the size of the surroundings Mx = My = 2. The neighborhoods overlap in one cell of 4.

2.2.3 Local transition function and rule

The mapping of the current environment (x, y) into the future parallel cell $c^{t+1}(x, y)$ is defined by the transition function f, which assigns a cell value to each environment value:

$$c^{t+1}(x,y) = f(n^t(x,y)) \qquad (2.30)$$

For the purpose of searching for pre-images, the inverse function f^{-1} is interesting, which, given the state of the current cell (x, y), returns the set of neighborhoods $n^{t-1}(x, y)$ that are mapped to this value:

$$f^{-1}(c^t(x, y) = \{n^{t-1}(x, y) \in S^m \mid f(n^{t-1}(x, y)) = c^t(x, y)\} \qquad (2.31)$$

The transition function can be defined by the rule r. The rule is the index of the selected function within the whole set of $|S|^{|S|\text{^}MxMy}$ functions. It is defined as an integer in a S-digit number structure, where the digits are a sequence of cell values into which the transition function maps each of the $|S|^{MxMy}$ possible environments:

$$r = \sum_{n=0}^{n=|S|^{M_xM_y}-1} |S|^n . f(n) \tag{2.32}$$

$$r \in \{0, 1, ..., |S|^{|S|^{\wedge MxMy}} - 1\} \tag{2.33}$$

2.2.4 Boundary conditions

An infinite field has no edge and we do not need a definition of boundary conditions for it. At the same time, in practice, we cannot count to infinity, so we consider the finite fields of the cells.

If we consider the observed finite field to be part of an infinite field, then we need to define how we will treat the cell values outside the observed field. Only an open boundary condition is described here, where cells outside the boundary are not defined, or can occupy any value. For a field of size $N_x \times N_y$, therefore, only $(N_x + (M_x - 1)) \times (N_y + (M_y - 1))$ neighborhoods are defined.

Another common boundary condition is cyclic, where the field of cells is a torus. The cyclic coordinate of the cell $(x_\circlearrowleft, x_\circlearrowleft)$ is calculated from (x, y) modulo N_x and N_y, respectively.

$$x_\circlearrowleft = x \bmod N_x \qquad y_\circlearrowleft = y \bmod N_y \tag{2.34}$$

2.2.5 Global transition function

The global transition function F, using the local transition function $f(n)$, maps each neighborhood $n^t(x, y)$ from a set of cells A^t at time t to a cell $c^{t+1}(x)$ from a set of cells B^{t+1} at time $t + 1$.

$$B^{t+1} = (A^t) \tag{2.35}$$

The global transition function for the input field of size $N_x \times N_y$ and for the open boundary condition returns the output field of size $(N_x + (M_x - 1)) \times (N_y + (M_y - 1))$, while for the cyclic boundary condition returns the output field of size $N_x \times N_y$.

2.2.6 Pre-images

Each pre-image B^{t-1} of size $(N_x + 1) \times (N_y + 1)$ of a given field A^t of size $N_x \times N_y$ must meet the following conditions:

1. For each cell $c^t(x, y)$ the environment $n^{t-1}(x, y)$ must correspond to the binding function f^{-1}:
$$\forall x, y : c^t(x, y) = f(n^{t-1}(x, y)) \tag{2.36}$$
2. Each pair of neighborhoods in dimension X $\{n^{t-1}(x, y), n^{t-1}(x + 1, y)\}$ must match in the overlap:

$$\forall x, y : o_{\leftrightarrow}^{t-1}(x, y) = o_{\rightarrow}^{t-1}(x, y)$$
$$= o_{\leftarrow}^{t-1}(x+1, y) \qquad (2.37)$$

3. Each pair of neighborhoods in dimension Y $\{n^{t-1}(x, y + 0), n^{t-1}(x, y + 1)\}$ must match in the overlap:
$$\forall x, y : o_{\updownarrow}^{t-1}(x, y) = o_{\downarrow}^{t-1}(x, y+1)$$
$$= o_{\uparrow}^{t-1}(x, y+0) \qquad (2.38)$$
4. Each four of diagonal neighborhoods $\{n^{t-1}(x, y), n^{t-1}(x + 1, y), n^{t-1}(x, y + 1), n^{t-1}(x + 1, y + 1)\}$ must match in the overlap:
$$\forall x, y : o_{\times}^{t-1}(x, y) = o_{\swarrow}^{t-1}(x+1, y+1)$$
$$= o_{\searrow}^{t-1}(x+0, y+1)$$
$$= o_{\nwarrow}^{t-1}(x+1, y+0)$$
$$= o_{\nearrow}^{t-1}(x+0, y+0) \qquad (2.39)$$

For the purposes of some illustrations, a more general definition of the inverse local transition function is used. Each cell can have its own set of pre-images $n^{t-1} \in S^M$, which is generally independent of the cell state. This generalization is used to construct artificial preview networks that highlight specific problems related to the complexity of the algorithm.

2.2.7 Overlap previews and cross section

If the field of 2D cellular automata is divided into two parts, we can define a pre-image for each part separately. Here we will limit ourselves to straight sections of the field, although it is generally possible to define any section.

The cross-sectional index in the direction of dimension X on the coordinate y and its set are defined as:

$$e_{\leftrightarrow}(y) = \sum_{\begin{cases} i=0 \\ j=0 \end{cases}}^{\begin{cases} i = N_x + M_x - 1 \\ j = M_y - 1 \end{cases}} |S|^{(N_x-1)j+1} . c(i, y+j) \qquad (2.40)$$

$$e_{\leftrightarrow} \in \{0, 1, ..., |S|^{(Nx + Mx - 1) . (My - 1)} - 1\} \qquad (2.41)$$

The cross-sectional index in the direction of dimension Y on the coordinate x and its set are defined as:

$$e_{\updownarrow}(x) = \sum_{\begin{cases} i=0 \\ j=0 \end{cases}}^{\begin{cases} i = M_x - 1 \\ j = N_y + M_y - 1 \end{cases}} |S|^{(N_x-1)j+1} . c(x+i, j) \qquad (2.42)$$

$$e_{\updownarrow} \in \{0, 1, ..., |S|^{(Mx - 1) . (Ny + My - 1)} - 1\} \qquad (2.43)$$

Sections are used to record intermediate results in 2D CA preview image algorithms.

2.2.8 Examples

Classic 2D surroundings

For classical 2D CAs, von Neumann and Moore surroundings are defined (Figure 2.10). Von Neumann neighborhood is a subset of Moore's neighborhood where the states of the corner neighborhoods are ignored. This means that the general definition described here encompasses both classical environments.

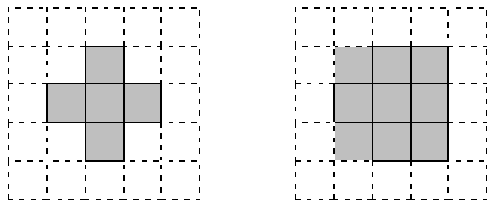

Figure 2.10: Classic 2D surroundings, von Neumann (left) and Moore (right).

Quad neighborhood

In the examples, a binary ($|S|$ = 2) 2D CA with quad M_x = M_y = 2 neighborhood is used (Figure 2.4). In the vicinity of n, M = $M_x \cdot M_y$ = 2 · 2 = 4 cells, which gives a set of sizes $|S|^4$ = 16. In the overlaps of $o_{\leftrightarrow}$ and $o_{\updownarrow}$ in the X and Y directions, M = (M_x - 1) · M_y = M_x · (M_y - 1) = 2 cells, which gives a set of sizes $|S|^2$ = 4. In overlaps $o_{\times}$ in diagonal directions, M = (M_x - 1) · (M_y - 1) = 1 · 1 = 1 cells, which gives a set of sizes $|S|^1$ = 2.

Conway's GoL

GoL is a binary cellular automaton ($|S|$ = 2) that uses the Moore neighborhood M_x = M_y = 3 (Figure 2.3). In the vicinity of n, M = $M_x \cdot M_y$ = 3 · 3 = 9 cells, which gives a set of sizes $|S|^9$ = 512. In the overlaps of $o_{\leftrightarrow}$ and $o_{\updownarrow}$ in the X and Y directions, M = (M_x - 1) · M_y = M_x · (M_y - 1) = 6 cells, which gives a set of sizes $|S|^6$ = 64. In overlaps $o_{\times}$ in diagonal directions, M = (M_x - 1) · (M_y - 1) = 2 · 2 = 4 cells, which gives a set of sizes $|S|^4$ = 16.

The set of all rules n for this environment is of size $|S|^{|S|\hat{}M}$ = 2^{512}. The local transition function is defined descriptively [7]. Cell states are described as dead (c = 0) and alive (c = 1). At each time step, the following happens to each cell:

1. a living cell with less than two living neighbors dies,
2. a living cell with two or three living neighbors lives on to the next step,
3. a living cell with more than three living neighbors dies,
4. a dead cell with exactly three living neighbors comes to life.

The GoL rule is a 512-bit binary value, in hexadecimal notation is r:

00000000000100010001000101170116000100010117011601170116177E1668
0001000101170116011701161 77E166801170116177E1668177E16687EE86880

CHAPTER 3

PRE-IMAGE GRID CONSTRUCTION

The pre-image grid is a graphical construct that allows the representation of the surroundings, the overlap of the surroundings and the transition function as separate graphic elements (nodes, paths and plots). The relationships between these elements define the rules on which the pre-image algorithms are built.

3.1 De Bruijn diagram

The basic element of graphic representation is a somewhat free interpretation of De Bruijn's diagram [8]. Basically, it deals with cyclic shifts of finite sequences of symbols and their overlap. Harold V. McIntosh [33] and his group [40] adapted De Bruijn graphs for 1D CA analysis (Figure 3.1 left). Nodes represent a set of possible overlaps between environments, and paths between them represent environments that match these overlaps in cell value. The transition function assigns a cell state to each environment; consequently, the paths between the nodes are marked with values from the transition table. De Bruijn graphs are then divided into subgraphs for each cell state. The subgraphs contain only those pathways whose associated surroundings are mapped to one of the cell states. The De Bruijn subgraph represents a set of all cell pasts with a given state.

3.2 Network of 1D cellular automata preimages

I developed a modified representation of De Bruijn's graph, where the original nodes (overlaps o) are duplicated and the paths (surroundings n) always go from the original $o_{\leftarrow}$ to the duplicates $o_{\rightarrow}$ (Figure 3.1 right) [30]. I called this representation a **graph of pre-images**.

The pre-image graph allows you to **string** graphs into a **pre-image grid**. The node duplicates for the current cell $o_{\rightarrow}(x)$ match the original nodes for the next cell $o_{\leftarrow}(x + 1)$ (Figure 1.4). While the basic De Bruijn graph describes the surroundings of a single cell, the string graph describes the surroundings of a set of cells. I named the string graph a grid of pre-images because it represents a set of all the pre-images of the entire configuration. The number of pre-images is equal to the number of all paths connecting one and the other edge.

The **boundary conditions** of 1D CA are defined at the left and right ends, which limit the finite number of cells within an infinite array (line). The boundary conditions define which overlaps $o_{\leftrightarrow}$ and with what weights are allowed at the edge. They can also be imagined as the influence of a half-infinite set of cells (half-strip) extending beyond the edge of the observed configuration. The boundary conditions of 1D CA are defined in more detail in [30].

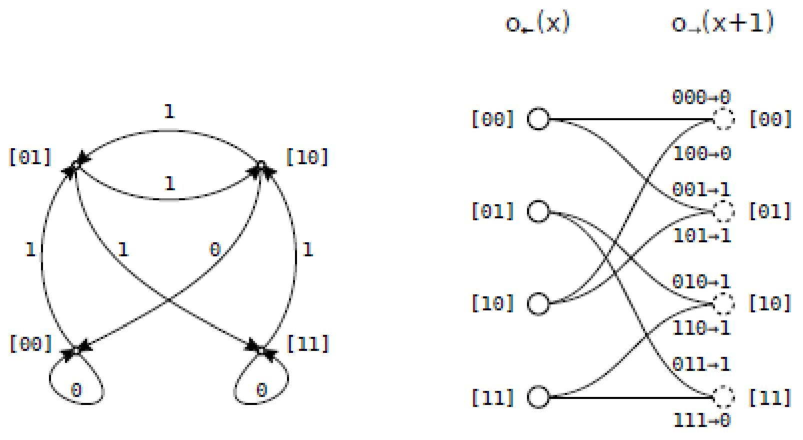

Figure 3.1: De Bruijn graph for elementary 1D CA, rule 110. On the left is McIntosh's representation, and on the right is mine, which allows stringing. Copies of nodes are represented by a dashed line.

3.3 Network of 2D cellular automata preimages

3.3.1 Graph of pre-image

For the purposes of the 2D CA description, a new dimension has been added to the pre-image graph element. The connections between the nodes change to plots, and the nodes change to the edges of the plots. The elements of the cellular automata that are mapped to the graph are:

- set of **surroundings** (n) becomes the set of **plots** (Figure 3.3),
- set of **overlaps in the direction of dimensions X and Y** ($o_{\leftrightarrow}$ and $o_{\updownarrow}$) (Figure 2.7) becomes a set of **connections**,
- set of **overlaps in the diagonal direction** ($o_{\times}$) (Figure 2.9) becomes a set of **nodes**.

The pre-image graph is placed on a square representing the surroundings n. Above each corner is a set of nodes representing a set of diagonal overlaps $o_{\swarrow}$, $o_{\searrow}$, $o_{\nwarrow}$ and $o_{\nearrow}$.

The nodes of adjacent corners in dimension X ($o_{\swarrow}$ and $o_{\searrow}$) are connected if they are part of the same cell configuration that represents the connection in the X direction ($o_{\downarrow}$). Nodes of adjacent corners in dimension Y ($o_{\swarrow}$ and $o_{\nwarrow}$) are connected if they are part of the same cell configuration that represents the connection in the Y direction ($o_{\leftarrow}$).

The resulting graph (Figure 3.2 right) has plots in addition to the nodes and connections between them. The corners $o_{\swarrow}$, $o_{\searrow}$, $o_{\nwarrow}$ and $o_{\nearrow}$ belong to a particular plane n if they are part of the same cell configuration. The same is true for edges; $o_{\leftarrow}$, $o_{\rightarrow}$, $o_{\downarrow}$ and $o_{\uparrow}$ belong to a certain plane n if they are part of the same cell configuration.

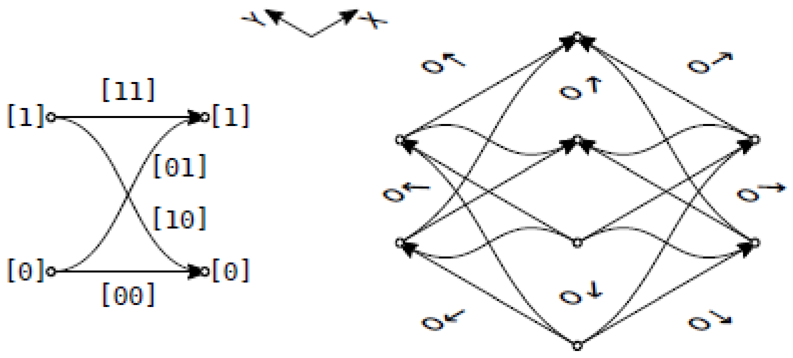

Figure 3.2: Single cell grid for binary CA with quad neighborhood M_x = M_y = 2.

The binary quad environment is used as an example. The plane n = $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ has angles $o_{\swarrow}$ = 1, $o_{\searrow}$ = 0, $o_{\nwarrow}$ = 0 and $o_{\nearrow}$ = 1 and edges $o_{\leftarrow}$ = $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$, $o_{\rightarrow}$ = $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $o_{\downarrow}$ = $[0 \quad 1]$ and $o_{\uparrow}$ = $[1 \quad 0]$.

3.3.2 Pre-image grid

Just as with 1D CA it is possible to string pre-image graphs into a pre-image grid; with 2D CA it is possible to pave pre-image graphs into a pre-image grid that describes a multi-cell field. The set of pre-images of the entire field is equivalent to the set of all continuous plots that overlap the entire field (the requirement to overlap the surroundings) and can be composed of sets of plots of individual cells (requirement to match the local transition function).

The mapping from the continuous plane in the pre-image grid to the pre-image cell configuration B^{t-1} is unique. From the plot of each neighborhood (x, y) we take the starting cell $c_{\swarrow}(x, y)$ (cell bottom left inside the neighborhood). Then add a row of cells of height M_y − 1 from the upper overlap at the top $o_{\uparrow}(x, y = N_y)$, and a column of cells of width M_x - 1 from the right overlap on the far right $o_{\rightarrow}(x = N_x, y)$.

The examples given use binary CA with quad environment. For this CA, the size of the diagonal overlap of the surroundings is a single cell (Figure 2.9); consequently, the set of nodes has only two values that directly represent the cell values in the preimage (Figure 3.4).

The correctness of the extension of the diagram of one cell into the grid can be proved by induction. We want to prove that a certain configuration of cells is a pre-image of a given present, *if and only if* it is equivalent to a continuous plane in the pre-image grid.

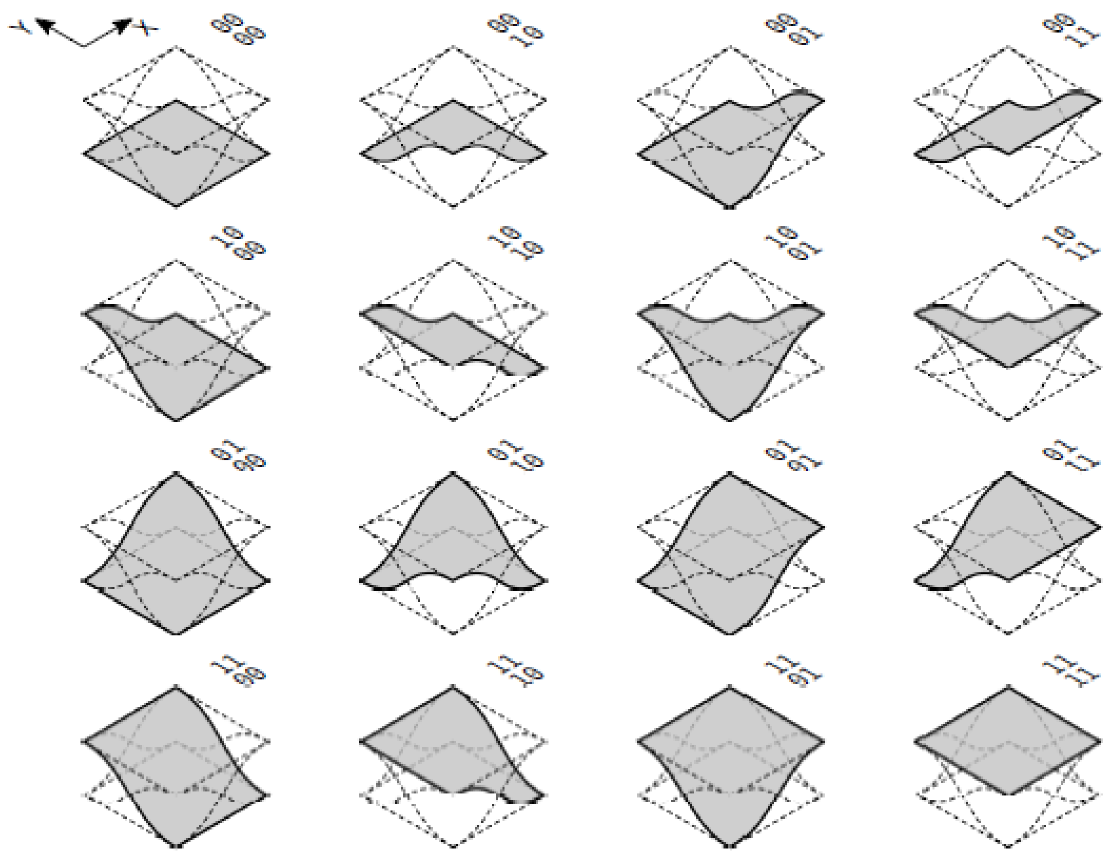

Figure 3.3: Set of plots for all 16 possible ambient values for a binary CA with quad neighborhood Mx = My = 2.

First element: From the definition of environment/plot, for a single cell in the grid, the set of plots is equal to the set of all preimages. **Next item**: Add a new cell to the existing pre-image grid. A plane from a set of added cells is continuously bound to a plane from an existing set of continuous surfaces when it matches it at the edge (nodes and the connection between them). The edges of the plots match exactly when the surrounding overlaps match. This is because the indices of nodes and connections are equivalent to the values of the overlaps of the surroundings in the diagonal and in the direction of the coordinate axes.

3.3.3 Boundary condition

In 2D CA, the boundary condition is defined as the weight of the closed path around the plane of the observed cell configuration. In the 2D CA pre-image grid, the connections between the nodes define the edge of the plot (Figure 3.4). Each continuous surface in the grid has a continuous edge. The boundary condition determines how this plot is treated, or whether it is treated at all. Since the useful value of the general boundary condition is not yet known, and the generality would markedly increase the complexity of the pre-image algorithm, here all edges are treated equally, with a weight of one. We will call this the open edge because it does not define any restrictions on which continuous surfaces in the pre-image grid are allowed and which are not.

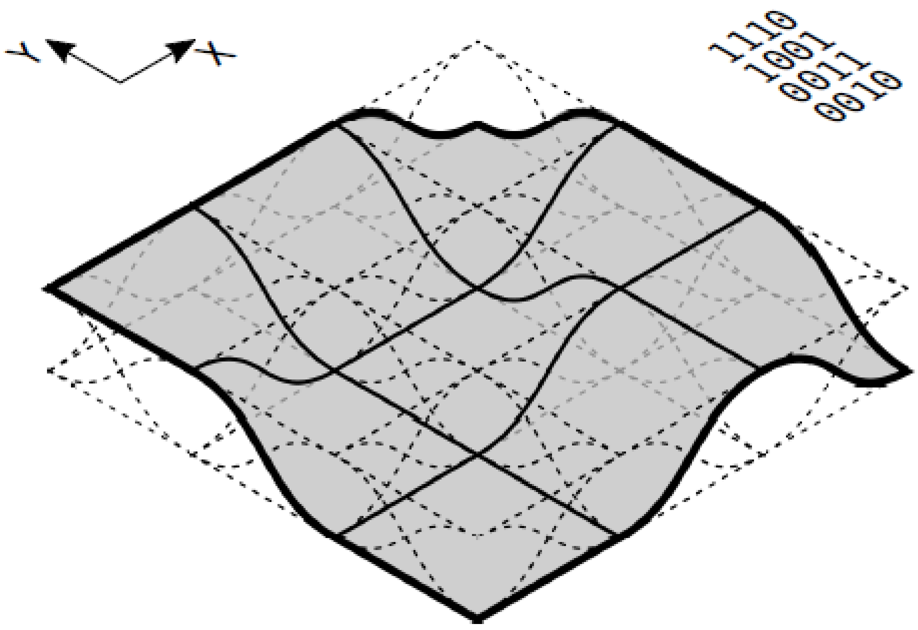

Figure 3.4: Grid of size N_x = 3 and N_y = 3 for binary CA with quad environment. One continuous surface and its continuous edge are highlighted. The configuration of the corresponding pre-image is displayed.

There is another simple boundary condition defined for cyclically closed end fields CA (torus). This type of boundary condition will not be considered here because it further increases the complexity of the algorithms. Namely, it would be necessary to perform the entire process of finding pre-images for each initial edge separately. After all, a continuous plane is a preimage in a cyclic field only if its initial and final edges match. The concepts of start and end edge are described in the section on algorithm. This problem for 1D CA is described in more detail in [30].

CHAPTER 4

ALGORITHM FOR COUNTING AND LISTING PREIMAGES

For 1D CA [30], the pre-image counting algorithm has linear (N) process and memory complexity. In general, this means that each cell appears in the calculation only once. Each counting algorithm also has a logarithmic component, as the number of bits required to write counters grows logarithmically with the number counted. This component is known in the implementation only when the size of the counters exceeds the size of the registers. 64 bits is enough for an 8 × 8 binary cell field.

For 2D CA, problems have been shown to be unsolvable with linear complexity. The maximum process complexity of the presented algorithm increases exponentially depending on one dimension and linearly depending on another dimension, i.e. $O(C^{N_x}N_y)$ or $O(N_xC^{N_y})$ (C = $|S|^M$ is a constant size of the set of surroundings). It is unknown at this time if there is an algorithm with a lower maximum complexity.

The algorithm for listing pre-images, regardless of the number of dimensions, inevitably has exponential complexity (C^N) (C is a constant that depends on optimizations), since the maximum and average number of pre-images grows exponentially depending on the number of cells.

4.1 Counting a path in a graph

The counting algorithm uses principles from graph theory [14]. The number of paths between two nodes in an acyclic directed graph is calculated by assigning a weight w to each node. In the first step, the initial node gets the weight (start) = 1. All paths originating from the weighted node have a weight equal to the weights of the source node. In the following steps, the node weights are calculated so that the output weight is the sum of the input weights w_{exit} = Σ w_{entry}. The algorithm terminates when the number of steps performed reaches the length of the longest path. Finally, the weight of the end node (end) is equal to the number of all paths between the start and end node.

If we want to know how many of these paths go through a single node or after a single connection between nodes, then we repeat the process in the opposite direction. The roles of the start and end nodes are reversed, and the directions of all paths are reversed. The total weight of a node is the product of the weights for back and forth w_{common} = $w_{\text{forward}} \cdot w_{\text{backward}}$.

In cellular automata, the observed graph is a grid of preimages that has a very regular structure. The length of the longest path is equal to the size of one of the dimensions of the CA field. In addition, at each step we observe only a part of the nodes belonging to one cross section.

4.2 Counting pre-images

The described algorithm divides the field of cells into rows in dimension X. Dividing by columns in dimension Y would be equivalent, so this is an arbitrary decision. From the point of view of process complexity, it is best to choose a shorter dimension.

Since the grid of pre-images is not an ordinary graph, but is extended by the term plot, the grid of pre-images must first be mapped to an ordinary graph before counting. Nodes represent all possible edges $e_{\leftrightarrow}(y)$ at all intersections between rows. Nodes are fixed elements in a graph. They exist regardless of the configuration and state of the pre-images.

The connections between the nodes are the areas between the two edges $e_{\leftrightarrow}(y)$ and $e_{\leftrightarrow}(y + 1)$. Each link represents one pre-image of a row of cells at coordinate y. A link in a graph exists if there is an equivalent row pre-image, so to build a graph you need to find all the pre-images for each row. Since the row in the 2D array is a 1D object, we can convert the problem to a 1D CA and use algorithms to count and list pre-images of the 1D CA to calculate the pre-images of the row.

By using Boolean algebra instead of multiplication and addition operations, it is possible to simplify the search for preimages only on their existence for the observed boundary condition. Weights become binary values $w \in \{0, 1\}$. A pre-image or a partial pre-image exists if its weight is non-zero $w > 0$. No counting is required to check this condition, logical operations are sufficient.

The implementation of the algorithm described here (source code is in Appendix D) does not search for all pre-images for a line at once, but is limited to pre-images that are consistent with one of the possible edges $e_{\leftrightarrow}$ and does so for all available edges.

4.2.1 Processing a string in dimension X

Processing starts with a 1D set of cells (row). Each cell in the array has its own grid of pre-images, and the one-dimensional array of cells consequently belongs to a connected set of grid pre-images (Figure 4.1, grid 1). Each network segment in a string has its own set of valid environments, which is defined by a transition function (or is a generalized arbitrary set). Initially, the surfaces surrounding the adjacent cells do not yet connect into a continuous plane across the entire set. It is necessary to exclude all surfaces of the surroundings that do not connect with the surroundings of their neighboring cells.

Since each line connects to the previous and next line, it is also necessary to take into account the continuity of the surfaces at the transition between the lines. This connection between the lines is a cross section of the observed area at the boundary between the lines. Each surface is treated separately and the cross section is expressed as a boundary condition. The initial boundary condition $e_{\downarrow}$ for each row is the set of paths between the nodes at the bottom of the row. Each initial condition is considered separately (Figure 4.1, grid 2, thickened path).

The initial boundary condition $e_{\downarrow}$ is applied by excluding from the consideration all plots that do not have a common edge with the boundary condition (Figure 4.1, transition from grid 1 to grid 2). After this step, all surroundings are defined on the overlap below. Continuity can only occur at the edge opposite the initial edge $e_{\uparrow}$.

The problem is reduced from 3D grid and surface processing to 2D grid, where paths between nodes are processed (Figure 4.1, transition from grid 2 to grid 3). This is a problem equivalent to counting and listing pre-images for 1D CA, for which I used the algorithm described in [30]. The result is a set of final boundary conditions $e_{\uparrow}$ (Figure 4.1, grid 4, the final boundary condition is thickened).

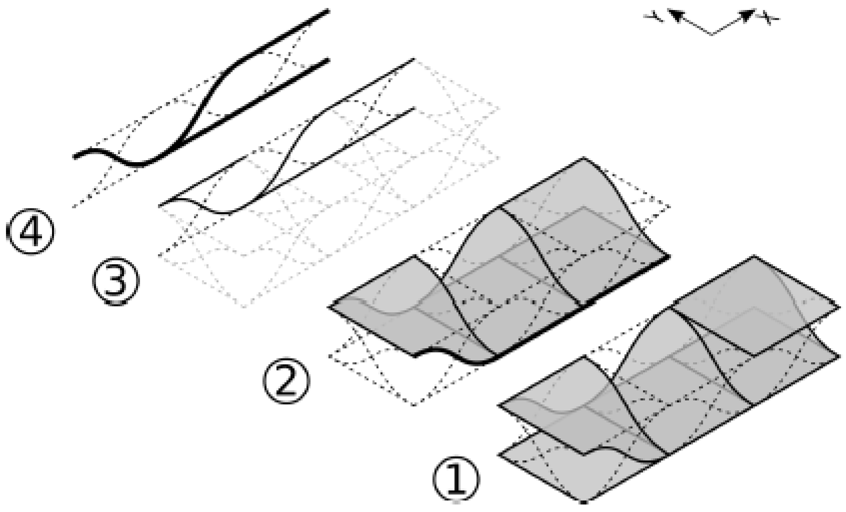

Figure 4.1: Line grid of size N_x = 3 for binary CA with quad environment. The set of environments/plots for an individual cell is arbitrary. It is chosen to emphasize the steps of the algorithm.

After processing the set of all initial boundary conditions $e_{\downarrow}$, a set of all final boundary conditions $e_{\uparrow}$ is created. This set is used in the next step as the initial boundary condition for the next line.

Due to the process of plotting the path in the graph, the process complexity grows exponentially with the length of the line (C^{Nx}).

4.2.2 Field processing in dimension Y

In the second dimension, the sequence of lines from the first to the last is processed (Figure 4.2). Each line begins and ends with a boundary condition. Each line needs to be processed only once in the second dimension, which means that the computational complexity is linearly dependent on the number of lines (N_y).

A weight is assigned to each boundary condition. The start edges for the first line have an assigned weight w = 1. In general, several start edges can be mapped to the same end edge of one line. The weight assigned to the end edge of a given line is the sum of the weights of all the initial edge conditions mapped to it. The weight thus represents the number of all pre-images for a given and previous lines ending with the observed edge. The edges that do not end with any of the pre-images get a weight w = 0.

After all the rows have been processed, the final edge of the set of all continuous plots on the entire pre-image grid is known. The sum of the weights of all the end edges gives the number of pre-images of the entire grid.

4.3 Listing pre-images

The pre-image algorithm (source code is in Appendix D) is a continuation of the counting algorithm. It starts with the known number of pre-images given by the count and lists the pre-images line by line in the reverse direction as the counting took place (from the last to the first line). The obtained list of pre-images is sorted. The sorting direction depends on the processing direction. The direction of processing can be changed if necessary.

It is not necessary for each line initial condition to be mapped to a set of final conditions. It is possible that no path at the end edge satisfies the initial condition and the grid of pre-images. So, according to the previous steps, it is not yet determined exactly which plots are merged into a federal whole and which are not. Areas that overlap the field only up to a certain line and not forward are possible.

Processing according to the second dimension takes place in the opposite direction as in counting; starts at the last line and ends at the first line (Figure 4.2). With each step, the remaining blind paths are eliminated from the pre-image grid. For each line, the 1D problem needs to be solved again, except that the start and end edges are swapped.

At the same time, it is possible to list pre-images. Already at the beginning of processing in the opposite direction, the number of all pre-images is known which enables memory reservation and initialization of the pre-image list. The end boundary paths and their weights are used to list the current row of cells in the pre-images. The boundary path index (a specific sequence of overlapping environments) gives the value of the cells in the pre-image row. The weights calculated in the counting direction, however, give the number of pre-images that need to be listed with a given value for a row of cells.

With each step in the opposite direction, a new line is listed for each of the counted pre-images. When the algorithm returns to the starting line, all pre-images are listed. At the same time, all blind paths are eliminated from the pre-image grid; only the planes remaining that form continuous planes over the entire cell field remain.

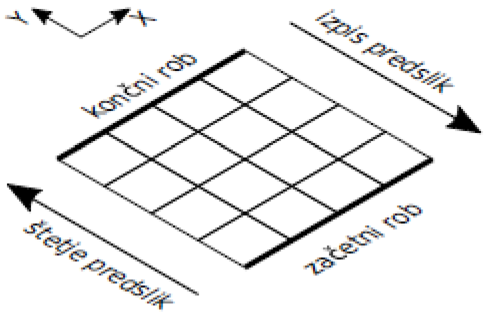

Figure 4.2: The course of the processing direction in the algorithm for counting and listing pre-images.

4.4 Inability to count with linear complexity

An example is given that shows why processing with linear complexity is not possible.

The aim of the research work was to find an efficient algorithm for finding 2D CA pre-images. Based on my experience with 1D CA, I optimistically expected that it would be possible to solve the problem in linear time.

An algorithm in linear time would consider each cell only once; in general, the number of readings would be a small constant (4 times if processing through a pre-image grid in 4 directions/transitions). Such an algorithm assumes that the entire set of initial boundary conditions can be considered simultaneously in a single transition. In step 2 in line processing (Figure 4.1), only plots that do not meet any of the initial boundary conditions would be excluded. After a few attempts, I found that such an algorithm did not work properly. At least some of the paths from the set need to be considered separately.

In the given case (Figure 4.3), the first two lines end with a set of two end paths **0110** and **1111**. If these two paths were used simultaneously as the initial boundary condition for the next row, we would get the same result as if they were also part of the boundary condition paths **0111** and **1110**. This is because an algorithm that does not treat the path separately cannot know where the path began so that it can also end it correctly. Consequently, a linear algorithm can take the beginning of one of the paths and continue it along the other path at the node common to both paths. It can be seen from the grid that these two paths are not the intersection of a federal plane. In short, simultaneous treatment of the whole set of boundary conditions is not possible, which means that the number of treatments of a cell depends on the number of paths and consequently on the exponent of the number of cells in a row.

It is possible that there are different algorithm optimizations that can reduce process complexity.

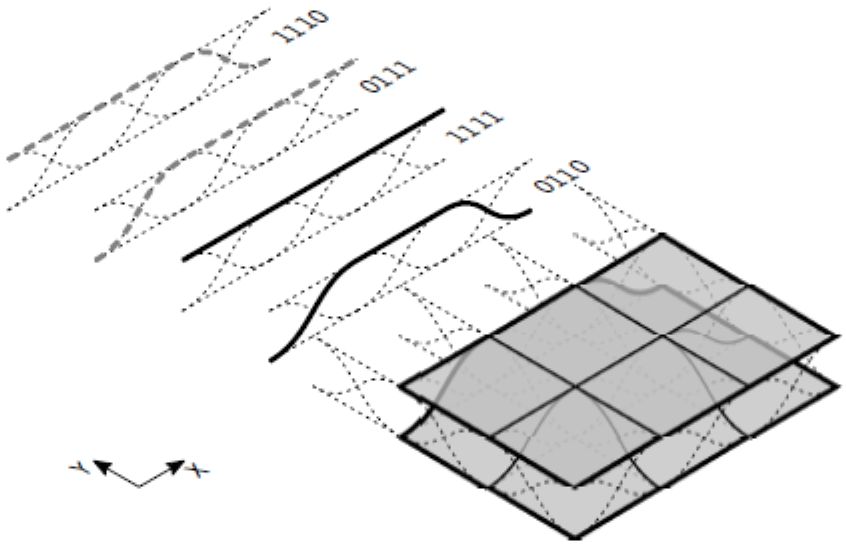

Figure 4.3: Linear complexity processing barrier. The valid boundary condition is bold and the invalid one is crossed out.

CHAPTER 5

OVERVIEW OF EXISTING ALGORITHMS AND IMPLEMENTATIONS

Existing 2D CA pre-imaging algorithms focus exclusively on GoL. Most algorithms are designed to search for GoE states, so they only check for the existence of pre-images and do not count or list them. The next group of algorithms lists pre-images, but does not do so systematically, as their goal is to find only one or a few pre-images. Algorithms designed to actually count and list all pre-images are rare. Academic papers are also very rare, where an algorithm related to 2D CA pre-images would also be professionally described.

5.1 Finding GoE states for GoL

In 1970, Conway introduced a CA called *Game of Life*, which soon began to gather enthusiasts. Part of these wondered if there were pre-image states in GoL called *Garden of Eden*. Some have undertaken research and Martin Gardner has proved the existence of GoE states theoretically [10]. Still others set out to find such a situation. As early as 1971, Roger Banks and Steve Ward introduced the first state of 9 × 33 cell GoE. The discovery was published in [43]. Later, Don Woods [43, 44] used a computer to prove that the condition was indeed GoE.

Woods' algorithm works on the principle of descent (see Appendix C). Divide the cell field into rows. It starts with one corner cell, whose set of pre-images is a set of surroundings that are mapped to the state of the cell. In the next step, it observes an adjacent cell in the row. For all combinations of surroundings, the added cell and the existing preset set check to see if they match where the surroundings overlap. If they match, it adds a composite pre-image to the set; otherwise the pre-image falls out of the set. When the algorithm reaches the end of the line, it continues in the next line, only now the overlap between the set of pre-images and the surroundings of the added cell is different (several adjacent cells instead of one). Thus, the algorithm continues until the end of the entire configuration. If the final preset set is empty, the GoE configuration is observed. The set of pre-images and thus the memory consumption and process time remain relatively small compared to the set of all states. This is because in practice, parts of the GoE configuration also have few pre-images. The algorithm would be much more memory and time consuming for configurations with many pre-images.

Nicolay Beluchenko looked for a larger number of 11 × 11 GoE states [2]. In at least one case, he used a similar algorithm as Woods [3]. He started with the central cell and added new cells in a spiral around the central one. For the value of the added cell, he used the one that gave a smaller set of pre-images. The algorithm terminates when it is no longer possible to create pre-images with the newly added cell.

In 1974, Duparc [21, 22] developed a different algorithm, using it to find GoE states of sizes 6 × 122 and 6 × 117. The algorithm uses finite state machine and regular language theory, which is basically intended for 1D systems. Duparc combined the cells from the 2D field line into regular language symbols, and the sequence of several lines represents the word. Duparc's algorithm does not treat all states of a line equally, but focuses on lines that reduce the set of preimages. Dealing with all line states would exceed the memory capabilities of then and current hardware even for short lines. It only proves that for rows of length 1 there is no column that would be the GoE state. The original article is in French, so I can only describe the algorithm based on how others describe it [10, 15].

In recent years, a similar algorithm has been used by Steven Eker (CSL, SRI International, California, USA) [19]. He proved that there are no GoE states for the rectangular configurations of heights 2 and 3. The proof for height configurations 4 with a given algorithm still exceeds the capabilities of modern hardware. Namely, the number of states of the automaton describing the GoE language grows exponentially with height. Eker searched for GoE sheets of sizes 9 × 11, 8 × 12, and 5 × 83. Unfortunately, the researcher is prohibited by the employer from publishing the results and details until an internal review is performed. Thus, for the time being, we only have the above data available.

A different algorithm is used by Marijn Heule [23]. He and his colleagues wrote down all the overlaps for a given state in the form of binary equations. These were then handed over to the SAT troubleshooting tool. If the solution does not exist, it means that the state is GoE. An additional assumption is used that the solution will be mirror-symmetric in both dimensions, which significantly reduces the size of the problem. The algorithm is further optimized to search for GoE states and allows an overview of a large number of similar states. When the value of a single cell changes, it allows the use of partial results from the previous calculation.

5.2 Unsystematic search of GoL pre-image

Erlan [18] presented a game in which pre-images are sought for a given state of GoL. This is not actually an implementation of a pre-image search algorithm, but it has encouraged others to search for such an algorithm.

The *kaggle.com* website organized a competition [1] where candidates had to solve the problem of finding pre-images. They gave a set of states for which it is necessary to look for pre-images of several steps into the past. They also provided a set of learning sequences. They expected a solution based on machine learning or optimization, so the focus was not on the exact solution. The goal was by no means to count or list out all the pre-images, but the candidates only had to get as close as possible to the learning sequence.

When searching for algorithm implementations, I found Atabot [16], a Cell Chronometer [17], and a program based on the SAT method [36]. The goal of these algorithms is to search for pre-images, but the search is not systematic and the goal is not a set of all possible pre-images. Algorithms also use certain heuristics when searching.

I also found an application written in the APL language [24]. From the above example, it is clear that the application is able to list pre-images. I was unable to contact the author, so I cannot provide more details.

5.3 Systematic algorithms for counting and listing pre-images

I found only one application called *RetroGUI*, which is designed to count and list all pre-images [15]. Neil Bickford, like the others, processes the configuration line by line, and can use various algorithms within the line (Woods, Duparc). Bickford also describes an algorithm that, instead of processing by lines, combines pre-images of square subsets of configurations. Unfortunately, it provides a comparison between different algorithms only as a process time for an individual implementation, but not a more general estimate of the maximum and average process/memory complexity.

CHAPTER 6

CONCLUSIONS

6.1 Proof

The master's thesis is more sparse with evidence. For 1D CA, I used direct evidence from graph theory [30], but for 2D CA, the grid of preimages is not a normal graph, so proving with graph theory is not so simple. It is probably possible, however, to convert a given graph into a dual form by a star-delta transformation. If there is a dual form of a graph where there is no plot but only nodes and connections, it could be used to derive and prove the actual computational complexity of the problem.

I checked the results given by the implementation of the algorithm several times. For a given input state, I converted each preimage with the transition function back to the present and compared the result with the input state. I also checked to see if all of the listed pre-images are unique. For a few smaller cases, I also looked for pre-images using the brute force method. This way I did not find any error in the algorithm.

Finally, I compared the results with RetroGUI [15] and found that they matched.

6.2 Contribution

Most known algorithms (e.g. Woods) use the descent method. They turn out if the pre-image is relatively small (e.g. GoE search). However, they are unsuitable for situations where the pre-images are relatively large because they store all the pre-images in memory throughout the execution. It might be possible to optimize memory consumption so that the algorithm only stores the current differences between the pre-images, instead of storing them entirely.

The main contribution of the master's thesis is the application of advanced algorithms developed for 1D CA to 2D problem. The pre-image counting algorithm described here has a predictable memory consumption that is asymptotically less than the memory consumption when descending (C^{NxNy}) (I give without proof). The pre-image listing algorithm is strictly separate from the count and has a linear complexity depending on the number of pre-images, which is optimal.

Graphical representation of the pre-image grid provides a better understanding of the pre-image problem and can facilitate the documentation of algorithms. The weighted grid of pre-images also indicates the spatial distribution of information loss.

Lastly, the described algorithm is conceived universally, for any 2D CA, although universality is not exactly important in practice, as outside of GoL researchers there is not much interest in theoretical problems such as finding pre-images.

6.3 Interesting thematically related problems

Problems related to finding pre-images can be divided into levels according to complexity:

1. Determining whether there are pre-images for a given current system state.
2. Count pre-images for a given current system state.
3. List pre-image configurations.
4. The question of system reversibility.
5. The language of all GoE states.

In my master's thesis, I describe an algorithm that solves the first three problems. The remaining problems are related. The problem of reversibility is generally proven unsolvable [31]. The problem of the GoE state language is easily solvable for 1D, and for 2D it is probably unsolvable similarly to the question of reversibility.

6.3.1 Analysis of 2D cellular automata using finite automata

The problem of state language without preimages would need a 2D formal language theory that does not yet exist. Nevertheless, it is possible to pose problems that can be mapped to a finite state machine. For example, if we treat lines as language symbols, we can compose a finite state machine. Symbols (line states) define the transitions between states of a finite state machine. The states of the automaton, however, are all possible sets of line edges. Only the presence of an edge is important, not the weight. The full set contains all possible edges, and the empty set does not contain any edges. Each sequence of lines (symbols) that leads from full to empty in the automaton is a GoE configuration. The loops in such a machine, however, are akin to particles.

The described approach becomes impractical even for short lines. The number of all possible edges grows exponentially with the length of the line N_x. This number, $|S|^{(M_y-1)(M_x-1+N_x)}$, is equal to the number of all ways in which the neighborhoods of the two lines can overlap. The states of the automaton, however, are all possible sets of edges. A set can be described as a binary string where each bit represents the presence or absence of one of the edges. The total state of the finite state machine is thus $2^{|S|^\wedge(M_y-1)(M_x-1+N_x)}$. The number of states of the automaton becomes unmanageable even for short lines.

The shortest word GoE within the regular GoE state language is the one that leads from the full to the empty set by the shortest path and without loops. For 1D CA, the length of the shortest word GoE has been shown to be related to the complexity of the spatiotemporal patterns that occur under a given rule. Similarly, with 2D CA, GoE configurations are probably larger in rules that allow complex particle dynamics. It would probably be possible to take advantage of this to find interesting rules around quad.

6.3.2 Improvements to the given algorithm

The implementation of the algorithm can be improved so that the complexity does not grow so fast with the size of the problem. For now, I only see a solution that focuses on average complexity instead of maximum complexity. For example, now the algorithm reserves memory for weights of all possible edges, and it would suffice if memory were reserved only for edges with a weight greater than zero. Especially when looking for GoE states, there are few edges compared to all possible ones. Such an approach would give much less memory consumption and probably also reduce processing time.

It might be possible to predetermine the upper limit of the number of non-zero edges. This would allow for an early assessment if the calculation for a given problem is feasible. For example, if you treated all the starting edges at the same time in line processing, instead of each one separately, you would get an oversized set of end edges. Such a set would always contain all the correct end edges and a few more wrong ones. But the total number could be significantly less than the full set. This number could be used to estimate memory consumption and time. In this way, it would be possible to solve problems that are on the verge of what modern hardware can do.

Only the implementation of the algorithm can also be optimized. The universality of the algorithm, especially the universality of the size of the surroundings, contributes a lot of overhead costs. If the algorithm specialized only in quad or GoL, it could avoid quite a few operations. The same is true for a set of cell states. For binary CAs, it is possible to optimize operations that convert 2D arrays of cells to integers and vice versa. Binary data can be compactly packed into memory. It is also possible to use logical operations on the whole cell word, instead of processing each cell separately.

Processing in the algorithm takes place in nested loops and in a known order. Only a small fraction of memory accesses are random. The sequence of data in itself allows for good optimization of cache access, so there are not many obvious optimizations in this area.

APPENDIX A

DE BRUIJN DIAGRAM FOR GOL

Figure A.1 shows part of the De Bruijn diagram for GoL. A graph with 16 nodes and 64 connections is obviously too complex to be suitable for explaining the algorithm. The number of all possible environments (plot in the diagram) is even 512.

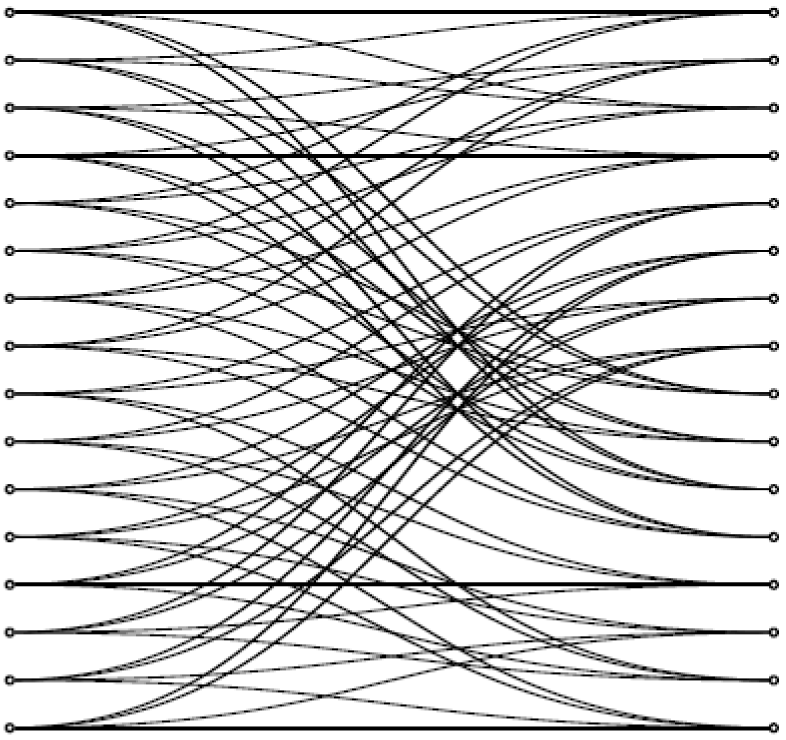

Figure A.1: Single cell network for a binary CA with a Moore environment M_x = M_y = 3.

APPENDIX B

EXAMPLE OF ALGORITHM OPERATION FOR CA WITH QUAD ENVIRONMENT

The following example shows the counting and search of preimages for a binary CA (a cell can have 2 states) with a 2 × 2 environment and rule number *0x725A*.

A small 2 × 2 configuration is stored in the *test_2x2.cas* file. I would use a larger configuration, but unfortunately I don't know a rule that would give a small number of preimages for a larger configuration.

The program [26] with version v1 was used.

```
$ ./ca2d_preimages 2 2 2 0x725a 2 2 test_2x2.cas
CA parameters:
        sts: 2
        ngb     : siz = {2,2}, a = 4, n = 16
        ovl-y   : siz = {1,2}, a = 2, n = 4
        ovl-x   : siz = {2,1}, a = 2, n = 4
        rem-y   : siz = {1,2}, a = 2, n = 4
        rem-x   : siz = {2,1}, a = 2, n = 4
        ver     : siz = {1,1}, a = 1, n = 2
        shf-y   : siz = {1,1}, a = 1, n = 2
        shf-x   : siz = {1,1}, a = 1, n = 2

RULE = 0x725a
        tab [0] = [[0,0], [0,0]] = 0
        tab [1] = [[1,0], [0,0]] = 1
        tab [2] = [[0,1], [0,0]] = 0
        tab [3] = [[1,1], [0,0]] = 1
        tab [4] = [[0,0], [1,0]] = 1
        tab [5] = [[1,0], [1,0]] = 0
        tab [6] = [[0,1], [1,0]] = 1
        tab [7] = [[1,1], [1,0]] = 0
        tab [8] = [[0,0], [0,1]] = 0
        tab [9] = [[1,0], [0,1]] = 1
        tab [A] = [[0,1], [0,1]] = 0
        tab [B] = [[1,1], [0,1]] = 0
        tab [C] = [[0,0], [1,1]] = 1
        tab [D] = [[1,0], [1,1]] = 1
        tab [E] = [[0,1], [1,1]] = 1
        tab [F] = [[1,1], [1,1]] = 0

CA configuration siz [y, x] = [2,2]:
        1 0
        0 1
```

```
NETWORK:        edge weights from forward/backward direction,
                summed preimages
net [dy=0][y=0] = [1 1 1 1 1 1 1 1 ]
net [dy=0][y=1] = [2 2 0 2 3 3 0 1 ]
net [dy=0][y=2] = [0 3 5 0 0 2 7 2 ]        cnt [0] = 19
net [dy=1][y=0] = [0 4 4 0 0 4 5 2 ]        cnt [1] = 19
net [dy=1][y=1] = [2 0 3 4 2 0 1 1 ]
net [dy=1][y=2] = [1 1 1 1 1 1 1 1 ]


preimage i=0 : CA configuration siz [y, x] = [3,3]:
        1 0 0
        0 0 0
        0 1 0

preimage i=1: CA configuration siz [y, x] = [3,3]:
        1 0 0
        0 0 0
        0 1 1

preimage i=2: CA configuration siz [y, x] = [3,3]:
        1 0 0
        0 0 1
        0 1 0

preimage i=3: CA configuration siz [y, x] = [3,3]:
        1 0 0
        0 0 1
        0 1 1

preimage i=4: CA configuration siz [y, x] = [3,3]:
        0 1 0
        1 1 0
        1 0 0

preimage i=5: CA configuration siz [y, x] = [3,3]:
        0 1 0
        1 1 0
        1 0 1

preimage i=6: CA configuration siz [y, x] = [3,3]:
        0 1 0
        1 1 0
        0 1 1

preimage i=7: CA configuration siz [y, x] = [3,3]:
        0 1 0
        1 1 0
        1 1 1
```

```
preimage i=8: CA configuration siz [y, x] = [3,3]:
        1 0 1
        0 0 0
        0 1 0

preimage i=9: CA configuration siz [y, x] = [3,3]:
        1 0 1
        0 0 0
        0 1 1

preimage i=10: CA configuration siz [y ,x] = [3,3]:
        1 0 1
        0 0 1
        0 1 0

preimage i=11: CA configuration siz [y, x] = [3,3]:
        1 0 1
        0 0 1
        0 1 1

preimage i=12: CA configuration siz [y, x] = [3,3]:
        0 1 1
        1 1 0
        1 0 0

preimage i=13: CA configuration siz [y, x] = [3,3]:
        0 1 1
        1 1 0
        1 0 1

preimage i=14: CA configuration siz [y, x] = [3,3]:
        0 1 1
        1 1 0
        0 1 1

preimage i=15: CA configuration siz [y, x] = [3,3]:
        0 1 1
        1 1 0
        1 1 1

preimage i=16: CA configuration siz [y, x] = [3,3]:
        0 1 1
        1 1 1
        1 0 0

preimage i=17: CA configuration siz [y, x] = [3,3]:
        1 1 1
        0 0 1
        0 1 0
```

```
preimage i=18: CA configuration siz [y, x] = [3,3]:
        1 1 1
        0 0 1
        0 1 1
```

APPENDIX C

DESCRIPTION OF DON WOODS' ALGORITHM

I enclose a written exchange with the author of the first GoL preimage algorithm.

Don Woods <don@icynic.com>   Thu, Jul 28, 2016 at 11:45 PM
To: iztok.jeras@gmail.com
> Dear Mr. Woods,
>
> I am developing an algorithm for computing preimages of 2D CA. It is based
> on the latest research for computig preimages of 1D CA.
> I am using the 2x2 quad neighborhood in the examples instead of the 3x3
> neighborhood used in Life.
> https://github.com/jeras/fri-magisterij (work in progress and Slovene
> language)
> https://github.com/jeras/preimages-2D (not working yet)
>
> As part of my masters thesis I have to compare my algorithm to existing
> ones.
> You are known as the first person to prove a pattern to be a Garden of Eden
> orphan in the Game of Life.
> Could you send me some references to this proof? I was looking for an
> article to reference, but a generic description (preferably written by you)
> or source code would be as good.
>
> Regards,
> Iztok Jeras
>

Oof, it's been quite a while; let me see what I can remember.

The "proof" was programmatic, i.e. I wrote a program to compute predecessors to a given position, and the program claimed there were no predecessors to the particular position that someone had theorised was a Garden of Eden. I did not have a formal proof of correctness for the program. (I was a teenager when I wrote it!) I don't think I still have the source code anywhere, I'm afraid. (I might have a paper printout of it somewhere but don't have time to search right now.)

The program worked by considering each cell in turn and looking at all possible 3x3 predecessors that would produce the given state. It then looked at the next cell to the right, looking for 3x3 predecessors that were consistent with the 6 overlapping cells from the other, etc. (In practice, the 3x3 neighborhoods were indexed so that I could quickly find which of the 8 possible righthand extensions worked.) When it had a 3x(N+2) predecessor for the top row of N, it continued at the left of the next row, then across that row, etc. Obviously in those later rows there was only one new predecessor cell being introduced for each new current cell after the leftmost, so the combinatoric explosion was not excessive, particularly since the GoE pattern by design did not have many potential predecessors even for subsections. I also deliberately oriented the target pattern with the short dimension going across, so that I more quickly reached the point of extending the predecessor one cell at a time in later rows.

My program normally included a 1-wide border around the area being backtracked, to ensure there were no artifacts introduced, but for the GoE proof I restricted it to the non-empty rectangular boundary, which I think was 9 by 33? Thus, as I think was mentioned at the time, what I actually demonstrated was a stronger result: there was no predecessor that led to a state that included the GoE as a 9x33 subsection.

I hope that helps. Let me know if you have further questions.

-- Don Woods

Don Woods <don@icynic.com> Tue, Aug 16, 2016 at 6:42 PM
To: iztok.jeras@gmail.com
> I found the time to read your response just now.
>
> Your description of the algorithm confirms what I imagined based on third
> party descriptions.
> Your details regarding combinatorial explosion which affects memory
> consumption and processing time were extra helpful.
>
> May I cite this email in my master thesis, since there are no other direct
> sources I would know of?
>
> Regards,
> Iztok Jeras

Permission granted. Good luck with your thesis!

-- Don Woods

APPENDIX D

ALGORITHM IMPLEMENTATION SOURCE CODE

I am attaching the source code, which is otherwise located at:

https://github.com/jeras/preimages-2D

Of particular interest are the functions *ca1d_net* and *ca2d_net*, which are located towards the end of the file:

preimages–2D/ca2d_net.c

preimages–2D/ca2d.h

```
int ca2d_update (ca2d_t *ca2d) ;
int ca2d_bprint (ca2d_t ca2d) ;

#endif // CA2D_H

#ifndef CA2D_H
#define CA2D_H

// math libraries
#include <gmp.h>

#define uintca_t long long int unsigned

typedef struct {
        // basic parameters
        int unsigned y;  // Y size
        int unsigned x;  // X size
        // calculated parameters
        int unsigned a;  // area (x*y)
        int unsigned n;  // number of states
} ca2d_size_t;

typedef struct {
        // basic parameters
        int unsigned sts;  // number of cell states
        ca2d_size_t  ngb;  // neighborhood size
        mpz_t       rule; // rule
        // calculated neighborhood/overlap parameters
        struct {
                ca2d_size_t y;
                ca2d_size_t x;
        } ovl;          // overlap size
        struct {
                ca2d_size_t y;
                ca2d_size_t x;
        } rem;          // remainder size (remainder and overlap add to a whole neighborhood)
        ca2d_size_t ver;   // vertice size (corner overlap)
        struct {
                ca2d_size_t y;
                ca2d_size_t x;
        } shf;          // shift size (shift and vertice add to a whole overlap)
        // rule table
        int unsigned *tab;
} ca2d_t;

int ca2d_update (ca2d_t *ca2d);
int ca2d_bprint (ca2d_t  ca2d);

#endif // CA2D_H
```

preimages–2D/ca2d.c

```
// user interface libraries
#include <stdio.h>
#include <stdlib.h>

// math libraries
#include <stdint.h>
#include <math.h>
#include <gmp.h>

#include "ca2d.h"
#include "ca2d_rule.h"
#include "ca2d_array.h"
#include "ca2d_cfg.h"
#include "ca2d_fwd.h"
#include "ca2d_net.h"

////////////////////////////////////////////////////////////////////////////////
// main
////////////////////////////////////////////////////////////////////////////////

int main (int argc, char **argv) {
        // configuration
        ca2d_t  ca2d;
        ca2d_size_t  siz;
        char *filename;
        FILE  file;

        // read input arguments
        if (argc < 8) {
                fprintf (stderr, "Usage:\t%s STATES NEIGHBORHOOD_SIZE_Y NEIGHBORHOOD_SIZE_X RULE CA_SIZE_Y
        CA_SIZE_X ca_state_filename.cas\n", argv[0]);
                return (1);
        }
        ca2d.sts   = strtoul (argv[1], 0, 0);
        ca2d.ngb.y = strtoul (argv[2], 0, 0);
        ca2d.ngb.x = strtoul (argv[3], 0, 0);
        mpz_init_set_str (ca2d.rule, argv[4], 0);
        siz.y = strtoul (argv[5], 0, 0);
        siz.x = strtoul (argv[6], 0, 0);
        filename = argv[7];

        // update ca2d structure
        ca2d_update (&ca2d);
        printf ("\n");

        // read CA configuration file
        int unsigned cai [siz.y] [siz.x];
        int unsigned cao [siz.y] [siz.x];
        ca2d_read (filename, siz, cai);
```

```
	ca2d_print (siz, cai);
	printf ("\n");

	// calculate network
	mpz_t cnt [2];
	ca2d_size_t siz_pre = {siz.y+ca2d.ver.y, siz.x+ca2d.ver.x};
	siz_pre.a = siz_pre.y * siz_pre.x;
	int unsigned (* p_list) [] [siz_pre.y] [siz_pre.x];
	int unsigned (*   list)    [siz_pre.y] [siz_pre.x];
	ca2d_net (ca2d, siz, cai, cnt, &p_list);
	list = (void *)p_list;

	// calculate preimage from network
	int status;
	int unsigned preimage [siz_pre.y] [siz_pre.x];

	for (int unsigned d=0; d<2; d++) {
		gmp_printf ("cnt [%u] = %Zi\n", d, cnt [d]);
	}
	int unsigned error = 0;
	for (int unsigned i=0; i<mpz_get_ui(cnt[0]); i++) {
		printf ("preimage i=%u/%lu:  ", i, mpz_get_ui(cnt[0]));
		ca2d_print (siz_pre, list[i]);
		printf ("\n");
	}

	return (0);
}
```

preimages–2D/ca2d_array.h

```
#ifndef CA2D_ARRAY_H
#define CA2D_ARRAY_H

// math libraries
#include <gmp.h>

////////////////////////////////////////////////////////////////////////////////
// array <--> number  conversions
////////////////////////////////////////////////////////////////////////////////

int unsigned ca2d_array_print     (ca2d_size_t s, int unsigned array[s.y][s.x]);
int unsigned ca2d_array_to_ui     (int unsigned base, ca2d_size_t s, int unsigned array[s.y][s.x], int unsigned *number);
int unsigned ca2d_array_from_ui   (int unsigned base, ca2d_size_t s, int unsigned array[s.y][s.x], int unsigned  number);
int unsigned ca2d_array_to_mpz    (int unsigned base, ca2d_size_t s, int unsigned array[s.y][s.x], mpz_t number);
int unsigned ca2d_array_from_mpz  (int unsigned base, ca2d_size_t s, int unsigned array[s.y][s.x], mpz_t number);
int unsigned ca2d_array_slice     (ca2d_size_t is, ca2d_size_t ob, ca2d_size_t os, int unsigned ia[is.y][is.x], int unsigned oa[os.y][os.x]);
int unsigned ca2d_array_fit       (ca2d_size_t is, ca2d_size_t ob, ca2d_size_t os, int unsigned ia[is.y][is.x], int unsigned oa[os.y][os.x]);
int unsigned ca2d_array_combine_x (ca2d_size_t is0, ca2d_size_t is1, int unsigned ia0[is0.y][is0.x], int unsigned ia1[is1.y][is1.x], int unsigned oa[is0.y][is1.x+is0.x]);
int unsigned ca2d_array_combine_y (ca2d_size_t is0, ca2d_size_t is1, int unsigned ia0[is0.y][is0.x], int unsigned ia1[is1.y][is1.x], int unsigned oa[is1.y+is0.y][is0.x]);

#endif
```

preimages–2D/ca2d_array.c

```
////////////////////////////////////////////////////////////////////////////////
// array <--> number  conversions
////////////////////////////////////////////////////////////////////////////////

// user interface libraries
#include <stdio.h>

// math libraries
#include <gmp.h>

#include "ca2d.h"

int unsigned ca2d_array_print (ca2d_size_t s, int unsigned array[s.y][s.x]) {
	printf ("[");
	for (int unsigned y=0; y<s.y; y++) {
		printf ("%s[", y ? "," : "");
		for (int unsigned x=0; x<s.x; x++) {
			printf ("%s%u", x ? "," : "", array [y] [x]);
		}
		printf ("]");
	}
	printf ("]");
	return 0;
}

int unsigned ca2d_array_to_ui (int unsigned base, ca2d_size_t s, int unsigned array[s.y][s.x], int unsigned *number) {
	*number = 0;
	int unsigned mul = 1;
	for (int unsigned y=0; y<s.y; y++) {
		for (int unsigned x=0; x<s.x; x++) {
			*number += array [y] [x] * mul;
			mul *= base;
		}
	}
	return 0;
}

int unsigned ca2d_array_from_ui (int unsigned base, ca2d_size_t s, int unsigned array[s.y][s.x], int unsigned number) {
	for (int unsigned y=0; y<s.y; y++) {
		for (int unsigned x=0; x<s.x; x++) {
			array [y] [x] = number % base;
			number /= base;
		}
	}
	return 0;
}

int unsigned ca2d_array_to_mpz (int unsigned base, ca2d_size_t s, int unsigned array[s.y][s.x], mpz_t number) {
	mpz_t mul;
	mpz_init (mul);
	mpz_set_ui (mul, 1);
	mpz_set_ui (number, 0);
	for (int unsigned y=0; y<s.y; y++) {
		for (int unsigned x=0; x<s.x; x++) {
			mpz_addmul_ui (number, mul, array [y] [x]);
			mpz_mul_ui (mul, mul, base);
		}
	}
	return 0;
}
```

```
int unsigned ca2d_array_from_mpz (int unsigned base, ca2d_size_t s, int unsigned array[s.y][s.x], mpz_t number) {
        mpz_t num;
        mpz_init_set (num, number);
        for (int unsigned y=0; y<s.y; y++) {
                for (int unsigned x=0; x<s.x; x++) {
                        array [y] [x] = mpz_tdiv_q_ui (num, num, base);
                }
        }
        return 0;
}

// get slice value
int unsigned ca2d_array_slice (ca2d_size_t is,
                ca2d_size_t ob, ca2d_size_t os,
                int unsigned ia[is.y][is.x], int unsigned oa[os.y][os.x])
{
        for (int unsigned y=0; y<os.y; y++) {
                for (int unsigned x=0; x<os.x; x++) {
                        oa[y][x] = ia[ob.y+y][ob.x+x];
                }
        }
        return 0;
}

// set slice value
int unsigned ca2d_array_fit (ca2d_size_t is,
                ca2d_size_t ob, ca2d_size_t os,
                int unsigned ia[is.y][is.x], int unsigned oa[os.y][os.x])
{
        for (int unsigned y=0; y<os.y; y++) {
                for (int unsigned x=0; x<os.x; x++) {
                        ia[ob.y+y][ob.x+x] = oa[y][x];
                }
        }
        return 0;
}

int unsigned ca2d_array_combine_x (ca2d_size_t is0, ca2d_size_t is1,
                int unsigned ia0[is0.y][is0.x], int unsigned ia1[is1.y][is1.x], int unsigned oa[is0.y][is1.x+is0.x])
{
        for (int unsigned y=0; y<is0.y; y++) {
                for (int unsigned x=0; x<is0.x; x++) {
                        oa[y][x      ] = ia0[y][x];
                }
                for (int unsigned x=0; x<is1.x; x++) {
                        oa[y][x+is0.x] = ia1[y][x];
                }
        }
        return 0;
}

int unsigned ca2d_array_combine_y (ca2d_size_t is0, ca2d_size_t is1,
                int unsigned ia0[is0.y][is0.x], int unsigned ia1[is1.y][is1.x], int unsigned oa[is1.y+is0.y][is0.x])
{
        for (int unsigned x=0; x<is0.x; x++) {
                for (int unsigned y=0; y<is0.y; y++) {
                        oa[y      ][x] = ia0[y][x];
                }
                for (int unsigned y=0; y<is0.y; y++) {
                        oa[y+is0.y][x] = ia1[y][x];
                }
        }
        return 0;
}
```

preimages–2D/ca2d_base.c

```
// math libraries
#include <stdio.h>
#include <math.h>
#include <gmp.h>

#include "ca2d.h"
#include "ca2d_rule.h"
#include "ca2d_array.h"
#include "ca2d_cfg.h"

int ca2d_update (ca2d_t *ca2d) {
        // overlay sizes
        ca2d->ovl.y   = (ca2d_size_t) {(ca2d->ngb.y-1) , (ca2d->ngb.x  )};
        ca2d->ovl.x   = (ca2d_size_t) {(ca2d->ngb.y  ) , (ca2d->ngb.x-1)};
        // remainder sizes
        ca2d->rem.y   = (ca2d_size_t) {(          1) , (ca2d->ngb.x  )};
        ca2d->rem.x   = (ca2d_size_t) {(ca2d->ngb.y  ) , (          1)};
        // vertice size
        ca2d->ver     = (ca2d_size_t) {(ca2d->ngb.y-1) , (ca2d->ngb.x-1)};
        // shift sizes
        ca2d->shf.y   = (ca2d_size_t) {(ca2d->ngb.y-1) , (          1)};
        ca2d->shf.x   = (ca2d_size_t) {(          1) , (ca2d->ngb.y-1)};
        // areas
        ca2d->ngb.a   = ca2d->ngb.y   * ca2d->ngb.x  ;
        ca2d->ovl.y.a = ca2d->ovl.y.y * ca2d->ovl.y.x;
        ca2d->ovl.x.a = ca2d->ovl.x.y * ca2d->ovl.x.x;
        ca2d->rem.y.a = ca2d->rem.y.y * ca2d->rem.y.x;
        ca2d->rem.x.a = ca2d->rem.x.y * ca2d->rem.x.x;
        ca2d->ver.a   = ca2d->ver.y   * ca2d->ver.x  ;
        ca2d->shf.y.a = ca2d->shf.y.y * ca2d->shf.y.x;
        ca2d->shf.x.a = ca2d->shf.x.y * ca2d->shf.x.x;
        // number of states
        ca2d->ngb.n   = (size_t) pow (ca2d->sts, ca2d->ngb.a  );
        ca2d->ovl.y.n = (size_t) pow (ca2d->sts, ca2d->ovl.y.a);
        ca2d->ovl.x.n = (size_t) pow (ca2d->sts, ca2d->ovl.x.a);
        ca2d->rem.y.n = (size_t) pow (ca2d->sts, ca2d->rem.y.a);
        ca2d->rem.x.n = (size_t) pow (ca2d->sts, ca2d->rem.x.a);
        ca2d->ver.n   = (size_t) pow (ca2d->sts, ca2d->ver.a  );
        ca2d->shf.y.n = (size_t) pow (ca2d->sts, ca2d->shf.y.a);
        ca2d->shf.x.n = (size_t) pow (ca2d->sts, ca2d->shf.x.a);
        // rule table
        ca2d_rule_table (ca2d);

        //    ca2d_bprint (*ca2d);
        return (0);
}

int ca2d_bprint (ca2d_t ca2d) {
        printf("CA parameters:\n");
        printf("sts  : %u\n", ca2d.sts);
        printf("ngb  : siz={%u,%u}, a=%u, n=%u\n", ca2d.ngb.y  , ca2d.ngb.x  , ca2d.ngb.a  , ca2d.ngb.n  );
        printf("ovl-y: siz={%u,%u}, a=%u, n=%u\n", ca2d.ovl.y.y, ca2d.ovl.y.x, ca2d.ovl.y.a, ca2d.ovl.y.n);
        printf("ovl-x: siz={%u,%u}, a=%u, n=%u\n", ca2d.ovl.x.y, ca2d.ovl.x.x, ca2d.ovl.x.a, ca2d.ovl.x.n);
        printf("rem-y: siz={%u,%u}, a=%u, n=%u\n", ca2d.rem.y.y, ca2d.rem.y.x, ca2d.rem.y.a, ca2d.rem.y.n);
        printf("rem-x: siz={%u,%u}, a=%u, n=%u\n", ca2d.rem.x.y, ca2d.rem.x.x, ca2d.rem.x.a, ca2d.rem.x.n);
        printf("ver  : siz={%u,%u}, a=%u, n=%u\n", ca2d.ver.y  , ca2d.ver.x  , ca2d.ver.a  , ca2d.ver.n  );
        printf("shf-y: siz={%u,%u}, a=%u, n=%u\n", ca2d.shf.y.y, ca2d.shf.y.x, ca2d.shf.y.a, ca2d.shf.y.n);
        printf("shf-x: siz={%u,%u}, a=%u, n=%u\n", ca2d.shf.x.y, ca2d.shf.x.x, ca2d.shf.x.a, ca2d.shf.x.n);
        printf("CA rule table:\n");
        ca2d_rule_print (ca2d);
        return (0);
}
```

preimages–2D/ca2d_cfg.h

```
#ifndef CA2D_CONFIGURATION_H
#define CA2D_CONFIGURATION_H

////////////////////////////////////////////////////////////////////////////////
// CA configuration handling
////////////////////////////////////////////////////////////////////////////////

int ca2d_read (char *filename, ca2d_size_t siz, int unsigned ca [siz.y] [siz.x]);
int ca2d_print (ca2d_size_t siz, int unsigned ca [siz.y] [siz.x]);
int ca2d_lattice_compare (ca2d_size_t siz, int unsigned ca0 [siz.y] [siz.x], int unsigned ca1 [siz.y] [siz.x]);

#endif
```

preimages–2D/ca2d_cfg.c

```
////////////////////////////////////////////////////////////////////////////////
// CA configuration handling
////////////////////////////////////////////////////////////////////////////////

#include <stdio.h>

#include "ca2d.h"

// read CA state from file
int ca2d_read (char *filename, ca2d_size_t siz, int unsigned ca [siz.y] [siz.x]) {
	FILE *fp;
	fp = fopen (filename, "r");
	if (!fp) {
		fprintf (stderr, "ERROR: file %s not found\n", filename);
		return (1);
	}
	for (int unsigned y=0; y<siz.y; y++) {
		for (int unsigned x=0; x<siz.x; x++) {
			fscanf (fp, "%u", &ca [y] [x]);
		}
	}
	fclose (fp);
	return (0);
}

// print CA state
int ca2d_print (ca2d_size_t siz, int unsigned ca [siz.y] [siz.x]) {
	printf ("CA configuration siz [y↓,x→] = [%u,%u]:\n", siz.y, siz.x);
	for (int unsigned y=0; y<siz.y; y++) {
		for (int unsigned x=0; x<siz.x; x++) {
			printf (" %u", ca [y] [x]);
		}
		printf ("\n");
	}
	return (0);
}

// compare CA states
int ca2d_lattice_compare (ca2d_size_t siz, int unsigned ca0 [siz.y] [siz.x], int unsigned ca1 [siz.y] [siz.x]) {
	for (int unsigned y=0; y<siz.y; y++) {
		for (int unsigned x=0; x<siz.x; x++) {
```

```
                        if (ca0 [y] [x] != ca1 [y] [x]) {
                            return (1);
                        }
                }
        }
        return (0);
}
```

preimages–2D/ca2d_rule.h

```
#ifndef CA2D_RULE_H
#define CA2D_RULE_H

// math libraries
#include <gmp.h>

////////////////////////////////////////////////////////////////////////////////
// rule handling
////////////////////////////////////////////////////////////////////////////////

// function for transforming the Game of Life rule description into a number
int ca2d_rule_gol ();

int ca2d_rule_table (ca2d_t *ca2d);
int ca2d_rule_print (ca2d_t ca2d);

#endif
```

preimages–2D/ca2d_rule.c

```
////////////////////////////////////////////////////////////////////////////////
// rule handling
////////////////////////////////////////////////////////////////////////////////

// user interface libraries
#include <stdio.h>
#include <stdlib.h>

// math libraries
#include <math.h>
#include <gmp.h>

// CA libraries
#include "ca2d.h"
#include "ca2d_array.h"

// function for transforming the Game of Life rule description into a number
int ca2d_rule_gol () {
        int unsigned len = 1<<9;
        int unsigned sum, out;
        mpz_t rule;
        mpz_init (rule);
        // loop through the rule table
        for (int i=len-1; i>=0; i--) {
```

```
				// calculate neighborhood sum
				sum = 0;
				for (int unsigned b=0; b<9; b++) {
					sum += ((i & 0b111101111) >> b) & 0x1;
				}
				// calculate GoL transition function
				if      (sum == 3)  out = 1;
				else if (sum == 2)  out = (i & 0b000010000) >> 4;
				else                out = 0;
				// add rule table element to rule number
				mpz_mul_ui (rule, rule, 2);
				mpz_add_ui (rule, rule, out);
			}
			// return rule
			gmp_printf ("RULE GoL = 0x%ZX\n", rule);
			return (0);
}

int ca2d_rule_table (ca2d_t *ca2d) {
			// neighborhood area
			// check if it is within allowed values, for example less then 9==3*3
			if ((ca2d->ngb.x == 0) || (ca2d->ngb.y == 0) || ((ca2d->ngb.y * ca2d->ngb.x) > 9)) {
				printf ("ERROR: neighborhood area %u is outside range [1:9].\n", ca2d->ngb.a);
				return (1);
			}

			mpz_t range;
			mpz_init (range);
			mpz_ui_pow_ui (range, ca2d->sts, ca2d->ngb.n);
			if (mpz_cmp (ca2d->rule, range) > 0) {
				gmp_printf ("ERROR: rule is outside of range = %Zi\n", range);
				return (1);
			}
			mpz_clear (range);

			// allocate rule table memory
			ca2d->tab = malloc (ca2d->ngb.n * sizeof(int unsigned));

			// rule table (conversion to base sts)
			mpz_t rule_q;
			mpz_init_set (rule_q, ca2d->rule);
			for (int unsigned i=0; i<ca2d->ngb.n; i++) {
				// populate transition function
				ca2d->tab[i] = mpz_tdiv_q_ui (rule_q, rule_q, ca2d->sts);
			}
			mpz_clear (rule_q);

			return (0);
}

int ca2d_rule_print (ca2d_t ca2d) {
			int unsigned a [ca2d.ngb.y] [ca2d.ngb.x];
			for (int unsigned n=0; n<ca2d.ngb.n; n++) {
			// convert table index into neighborhood status 2D array
			ca2d_array_from_ui (ca2d.sts, ca2d.ngb, a, n);
			// print rule table
			printf ("  tab [%X] = [", n);
				for (int unsigned y=0; y<ca2d.ngb.y; y++) {
					printf ("%s[", y ? "," : "");
					for (int unsigned x=0; x<ca2d.ngb.x; x++) {
						printf ("%s%u", x ? "," : "", a[y][x]);
					}
					printf ("]");
				}
				printf ("] = %u\n", ca2d.tab[n]);
		}
		return (0);
}
```

preimages–2D/ca2d_fwd.h

```
#ifndef CA2D_FORWARD_H
#define CA2D_FORWARD_H

////////////////////////////////////////////////////////////////////////////////
// array <--> number  conversions
////////////////////////////////////////////////////////////////////////////////

int unsigned ca2d_forward (ca2d_t ca2d, ca2d_size_t siz,
		int unsigned ai[siz.y][siz.x],
		int unsigned ao[siz.y-(ca2d.ngb.y-1)][siz.x-(ca2d.ngb.x-1)]
);

#endif
```

preimages–2D/ca2d_fwd.c

```
////////////////////////////////////////////////////////////////////////////////
// array <--> number  conversions
////////////////////////////////////////////////////////////////////////////////

// math libraries
#include <math.h>

#include "ca2d.h"
#include "ca2d_rule.h"
#include "ca2d_array.h"

int unsigned ca2d_forward (ca2d_t ca2d, ca2d_size_t siz,
				int unsigned ai[siz.y][siz.x],
				int unsigned ao[siz.y-(ca2d.ngb.y-1)][siz.x-(ca2d.ngb.x-1)])
{
		int unsigned at [ca2d.ngb.y] [ca2d.ngb.x];
		int unsigned nt;
		for (int unsigned y=0; y<siz.y-(ca2d.ngb.y-1); y++) {
				for (int unsigned x=0; x<siz.x-(ca2d.ngb.x-1); x++) {
						ca2d_array_slice (siz, (ca2d_size_t) {y, x}, ca2d.ngb, ai, at);
						ca2d_array_to_ui (ca2d.sts, ca2d.ngb, at, &nt);
						ao [y] [x] = ca2d.tab [nt];
				}
		}
		return 0;
}
```

preimages–2D/ca2d_net.h

```
#ifndef CA2D_NETWORK_H
#define CA2D_NETWORK_H

////////////////////////////////////////////////////////////////////////////////
// CA preimage network
////////////////////////////////////////////////////////////////////////////////

int ca2d_net_print (ca2d_t ca2d, ca2d_size_t siz, int unsigned res [siz.y] [siz.x] [ca2d.ngb.n]);
int ca2d_net       (ca2d_t ca2d, ca2d_size_t siz, int unsigned ca [siz.y] [siz.x], mpz_t cnt [2], int unsigned (** p_list) [] [siz.y+ca2d.ver.y]
[siz.x+ca2d.ver.x]);

#endif
```

preimages–2D/ca2d_net.c

```
// user interface libraries
#include <stdio.h>
#include <stdlib.h>

// math libraries
//#include <stdint.h>
#include <math.h>
#include <gmp.h>

#include "ca2d.h"
#include "ca2d_rule.h"
#include "ca2d_array.h"
#include "ca2d_cfg.h"

//		// print edge counters
//		printf ("NETWORK: edge weights from forward/backward direction,\n");
//		printf ("	summed preimages\n");
//		for (int d=0; d<2; d++) {
//			for (int y=0; y<=siz.y; y++) {
//				printf ("net [dy=%u][y=%u] = [", d, y);
//				for (int unsigned edg=0; edg<edg_x; edg++) {
//					gmp_printf ("%Zi ", net[d][y][edg]);
//				}
//				printf ("]");
//				if ((y-(1-d)*(siz.y))==0) {
//					gmp_printf ("  cnt [%u] = %Zi", d, cnt [d]);
//				}
//				printf ("\n");
//			}
//		}
//		printf ("\n");

// tables for geting overlap values from neighborhood values
static int ca2d_net_table_v2o (ca2d_t ca2d, int unsigned v2o_y [2] [ca2d.ovl.y.n] [ca2d.shf.y.n],
			int unsigned v2o_x [2] [ca2d.ovl.x.n] [ca2d.shf.x.n])
{
	int unsigned ovl_ay [ca2d.ovl.y.y] [ca2d.ovl.y.x];
	int unsigned ovl_ax [ca2d.ovl.x.y] [ca2d.ovl.x.x];
	int unsigned ovl_ayo[ca2d.ovl.y.y] [ca2d.ovl.y.x];
	int unsigned ovl_axo[ca2d.ovl.x.y] [ca2d.ovl.x.x];
	int unsigned shf_ay [ca2d.shf.y.y] [ca2d.shf.y.x];
	int unsigned shf_ax [ca2d.shf.x.y] [ca2d.shf.x.x];
	int unsigned ver_a  [ca2d.ver.y  ] [ca2d.ver.x  ];

	for (int unsigned ovl=0; ovl<ca2d.ovl.y.n; ovl++) {
		ca2d_array_from_ui (ca2d.sts, ca2d.ovl.y, ovl_ay, ovl);
		for (int unsigned shf=0; shf<ca2d.shf.y.n; shf++) {
			ca2d_array_from_ui (ca2d.sts, ca2d.shf.y, shf_ay, shf);
			for (int unsigned d=0; d<2; d++) {
				ca2d_array_slice (ca2d.ovl.y, (ca2d_size_t) {0, d}, ca2d.ver, ovl_ay, ver_a);
				if (!d)  ca2d_array_combine_x (ca2d.shf.y, ca2d.ver, shf_ay, ver_a, ovl_ayo);  // shift added
			to the left  of overlap
				else     ca2d_array_combine_x (ca2d.ver, ca2d.shf.y, ver_a, shf_ay, ovl_ayo);  // shift added
			to the right of overlap
				ca2d_array_to_ui (ca2d.sts, ca2d.ovl.y, ovl_ayo, &v2o_y [d] [ovl] [shf]);
			}
		}
	}
	for (int unsigned ovl=0; ovl<ca2d.ovl.x.n; ovl++) {
		ca2d_array_from_ui (ca2d.sts, ca2d.ovl.x, ovl_ax, ovl);
		for (int unsigned shf=0; shf<ca2d.shf.x.n; shf++) {
			ca2d_array_from_ui (ca2d.sts, ca2d.shf.x, shf_ax, shf);
			for (int unsigned d=0; d<2; d++) {
				ca2d_array_slice (ca2d.ovl.x, (ca2d_size_t) {d, 0}, ca2d.ver, ovl_ax, ver_a);
				if (!d)  ca2d_array_combine_y (ca2d.shf.x, ca2d.ver, shf_ax, ver_a, ovl_axo);  // shift added
			to the left  of overlap
				else     ca2d_array_combine_y (ca2d.ver, ca2d.shf.x, ver_a, shf_ax, ovl_axo);  // shift added
			to the right of overlap
				ca2d_array_to_ui (ca2d.sts, ca2d.ovl.x, ovl_axo, &v2o_x [d] [ovl] [shf]);
			}
		}
	}
	return (0);
}

// tables for geting overlap values from neighborhood values
static int ca2d_net_table_n2o (ca2d_t ca2d, int unsigned n2o_y [2] [ca2d.ngb.n],
		int unsigned n2o_x [2] [ca2d.ngb.n])
{
	int unsigned ovl_ay [ca2d.ovl.y.y] [ca2d.ovl.y.x];
	int unsigned ovl_ax [ca2d.ovl.x.y] [ca2d.ovl.x.x];
	int unsigned ngb_a  [ca2d.ngb.y  ] [ca2d.ngb.x  ];

	for (int unsigned ngb=0; ngb<ca2d.ngb.n; ngb++) {
		// neighborhood integer is converted into array
		ca2d_array_from_ui (ca2d.sts, ca2d.ngb, ngb_a, ngb);
		for (int unsigned d=0; d<2; d++) {
			ca2d_array_slice (ca2d.ngb, (ca2d_size_t) {d, 0}, ca2d.ovl.y, ngb_a, ovl_ay);
			ca2d_array_to_ui (ca2d.sts, ca2d.ovl.y, ovl_ay, &n2o_y [d] [ngb]);
		}
		for (int unsigned d=0; d<2; d++) {
			ca2d_array_slice (ca2d.ngb, (ca2d_size_t) {0, d}, ca2d.ovl.x, ngb_a, ovl_ax);
			a2d_array_to_ui (ca2d.sts, ca2d.ovl.x, ovl_ax, &n2o_x [d] [ngb]);
		}
	}
	return (0);
}

// tables for geting neighborhood values from overlap values and rest
int ca2d_net_table_o2n (ca2d_t ca2d, int unsigned o2n_y [2] [ca2d.ovl.y.n] [ca2d.rem.x.n],
		int unsigned o2n_x [2] [ca2d.ovl.x.n] [ca2d.rem.y.n])
{
	int unsigned ovl_ay [ca2d.ovl.y.y] [ca2d.ovl.y.x];
	int unsigned ovl_ax [ca2d.ovl.x.y] [ca2d.ovl.x.x];
	int unsigned rem_ay [ca2d.rem.y.y] [ca2d.rem.y.x];
```

```
        int unsigned rem_ax [ca2d.rem.x.y] [ca2d.rem.x.x];
        int unsigned ngb_a  [ca2d.ngb.y  ] [ca2d.ngb.x  ];
        int unsigned ovl_y;
        int unsigned ovl_x;
        int unsigned rem_y;
        int unsigned rem_x;

        // for every neighborhood integer
        for (int unsigned ngb=0; ngb<ca2d.ngb.n; ngb++) {
                // neighborhood integer is converted into array
                ca2d_array_from_ui (ca2d.sts, ca2d.ngb, ngb_a, ngb);
                for (int unsigned d=0; d<2; d++) {
                        // neighborhood array is split into overlay and reminder arrays
                        if (!d) {
                                ca2d_array_slice(ca2d.ngb, (ca2d_size_t) {0          , 0}, ca2d.ovl.y, ngb_a, ovl_ay);
                                ca2d_array_slice(ca2d.ngb, (ca2d_size_t) {ca2d.ovl.y.y, 0}, ca2d.rem.y, ngb_a, rem_ay);
                        } else {
                                ca2d_array_slice(ca2d.ngb, (ca2d_size_t) {0          , 0}, ca2d.rem.y, ngb_a, rem_ay);
                                ca2d_array_slice(ca2d.ngb, (ca2d_size_t) {ca2d.rem.y.y, 0}, ca2d.ovl.y, ngb_a, ovl_ay);
                        }
                        // overlay and reminder arrays are converted into integers
                        ca2d_array_to_ui (ca2d.sts, ca2d.ovl.y, ovl_ay, &ovl_y);
                        ca2d_array_to_ui (ca2d.sts, ca2d.rem.y, rem_ay, &rem_y);
                        // table is pupulated
                        o2n_y [d] [ovl_y] [rem_y] = ngb;
                }
                for (int unsigned d=0; d<2; d++) {
                        // neighborhood array is split into overlay and reminder arrays
                        if (!d) {
                                ca2d_array_slice(ca2d.ngb, (ca2d_size_t) {0, 0          }, ca2d.ovl.x, ngb_a, ovl_ax);
                                ca2d_array_slice(ca2d.ngb, (ca2d_size_t) {0, ca2d.ovl.x.x}, ca2d.rem.x, ngb_a, rem_ax);
                        } else {
                                ca2d_array_slice(ca2d.ngb, (ca2d_size_t) {0, 0          }, ca2d.rem.x, ngb_a, rem_ax);
                                ca2d_array_slice(ca2d.ngb, (ca2d_size_t) {0, ca2d.rem.x.x}, ca2d.ovl.x, ngb_a, ovl_ax);
                        }
                        ca2d_array_to_ui (ca2d.sts, ca2d.ovl.x, ovl_ax, &ovl_x);
                        ca2d_array_to_ui (ca2d.sts, ca2d.rem.x, rem_ax, &rem_x);
                        o2n_x [d] [ovl_x] [rem_x] = ngb;
                }
        }
        return (0);
}

// function for geting array of overlap integers from edge integer in X dimension
int ca2d_net_ex2o (ca2d_t ca2d, size_t siz, int unsigned e, int unsigned o [siz]) {
        ca2d_size_t se = {ca2d.ngb.y-1, ca2d.ngb.x-1 + siz};
        ca2d_size_t so = {ca2d.ngb.y-1, ca2d.ngb.x  };
        int unsigned ae [se.y] [se.x];
        int unsigned ao [so.y] [so.x];
        ca2d_array_from_ui (ca2d.sts, se, ae, e);
        for (int unsigned x=0; x<siz; x++) {
                ca2d_array_slice (se, (ca2d_size_t) {0, x}, so, ae, ao);
                ca2d_array_to_ui (ca2d.sts, so, ao, &o[x]);
        }
        return (0);
}

// function for geting edge integer in X dimension from array of overlap integers
int ca2d_net_o2ex (ca2d_t ca2d, size_t siz, int unsigned *e, int unsigned o [siz]) {
        ca2d_size_t se = {ca2d.ngb.y-1, ca2d.ngb.x-1 + siz};
        ca2d_size_t so = {ca2d.ngb.y-1, ca2d.ngb.x  };
        ca2d_size_t sr = {ca2d.ngb.y-1,            1};
        int unsigned ae [se.y] [se.x];
        int unsigned ao [so.y] [so.x];
        int unsigned ar [sr.y] [sr.x];
        // set first overlap
        ca2d_array_from_ui (ca2d.sts, so, ao, o[0]);
        ca2d_array_fit (se, (ca2d_size_t) {0, 0}, so, ae, ao);
        // apprend remaining remainders
        for (int unsigned x=0; x<siz; x++) {
                ca2d_array_from_ui (ca2d.sts, so, ao, o[x]);
                ca2d_array_slice (so, (ca2d_size_t) {0,   ca2d.ngb.x-1}, sr, ao, ar);
                ca2d_array_fit   (se, (ca2d_size_t) {0, x+ca2d.ngb.x-1}, sr, ae, ar);
        }
        ca2d_array_to_ui (ca2d.sts, se, ae, &(*e));
        return (0);
}

static int ca1d_net (// CA properties
                ca2d_t ca2d,
                size_t siz,
                // configuration
                int unsigned ca [siz],
                // direction in Y dimmension
                int unsigned dy,
                int unsigned edg_i,
                mpz_t edg_w,
                int unsigned *edg_n,
                mpz_t edg_o [(size_t) pow (ca2d.sts, (ca2d.ngb.y-1)*((ca2d.ngb.x-1)+siz))])
{
        // tables
        int unsigned n2o_y [2] [ca2d.ngb.n];
        int unsigned n2o_x [2] [ca2d.ngb.n];
        int unsigned o2n_y [2] [ca2d.ovl.y.n] [ca2d.rem.y.n];
        int unsigned o2n_x [2] [ca2d.ovl.x.n] [ca2d.rem.x.n];
        ca2d_net_table_n2o (ca2d, n2o_y, n2o_x);
        ca2d_net_table_o2n (ca2d, o2n_y, o2n_x);

        // memory allocation for preimage network
        int unsigned net [siz] [ca2d.ovl.y.n];

        // convert input edge into array of overlaps
        int unsigned ovl_i [siz];
        ca2d_net_ex2o (ca2d, siz, edg_i, ovl_i);

        int unsigned ngb_a  [ca2d.ngb.y  ] [ca2d.ngb.x  ];
        // create unprocessed 1D preimage network for the current edge
        for (int unsigned x=0; x<siz; x++) {
                // initialize network weights to zero
                for (int unsigned ovl=0; ovl<ca2d.ovl.y.n; ovl++) {
                        net [x] [ovl] = 0;
                }
                // get segment x from current edge
                int unsigned o = ovl_i [x];
                // for all neighborhoods with the curent edge check them against the table
                for (int unsigned rem=0; rem<ca2d.rem.x.n; rem++) {
                        // combine overlap and remainder into neighborhood
                        int unsigned ngb = o2n_y [dy] [o] [rem];
                        // check neighborhood against the rule
                        if (ca[x] == ca2d.tab[ngb]) {
                                // get end edge overlap from pointer table
                                int unsigned ovl = n2o_y [1-dy] [ngb];
                                // set weight to overlap
                                net [x] [ovl] = 1;
                        }
                }
        }
        // count 1D network preimages
        int unsigned v2o_y [2] [ca2d.ovl.y.n] [ca2d.shf.y.n];
        int unsigned v2o_x [2] [ca2d.ovl.x.n] [ca2d.shf.x.n];

        ca2d_net_table_v2o (ca2d, v2o_y, v2o_x);

        for (int x=1; x<siz; x++) {
                for (int unsigned ovl=0; ovl<ca2d.ovl.y.n; ovl++) {
                        // first check if the path is available
                        if (net [x] [ovl]) {
                                int unsigned sum = 0;
                                for (int unsigned shf=0; shf<ca2d.shf.y.n; shf++) {
                                        sum += net [x-1] [v2o_y[0][ovl][shf]];
                                }
                                net [x] [ovl] = sum;
                        }
                }
        }

        // calculate preimage number
        *edg_n = 0;
        for (int unsigned ovl=0; ovl<ca2d.ovl.y.n; ovl++) {
                *edg_n += net [siz-1] [ovl];
        }
        // allocate memory for preimage list
        int unsigned lst [*edg_n] [siz];

        // initialize list of 1D network preimages
        int unsigned p = 0;
        for (int unsigned ovl=0; ovl<ca2d.ovl.y.n; ovl++) {
                for (int unsigned i=0; i<net[siz-1][ovl]; i++) {
                        lst [p] [siz-1] = ovl;
                        p++;
                }
        }
        // list 1D network preimages
        for (int x=siz-2; x>=0; x--) {
                int unsigned p = 0;
                while (p < *edg_n) {
                        int unsigned ovl = lst [p] [x+1];
                        for (int unsigned shf=0; shf<ca2d.shf.y.n; shf++) {
                                int unsigned o = v2o_y[0][ovl][shf];
                                for (int unsigned i=0; i<net[x][o]; i++) {
                                        lst [p] [x] = o;
                                        p++;
                                }
                        }
                }
        }

        // put preimages into edge list
        for (int unsigned i=0; i<*edg_n; i++) {
                int unsigned edg;
                ca2d_net_o2ex (ca2d, siz, &edg, lst [i]);
                mpz_add (edg_o [edg], edg_o [edg], edg_w);
        }

        return (0);
}

int ca2d_net (ca2d_t ca2d, ca2d_size_t siz, int unsigned ca [siz.y] [siz.x], mpz_t cnt [2],
                    int unsigned (** p_list) [] [siz.y+ca2d.ver.y] [siz.x+ca2d.ver.x])
{
        // edge size
```

```
const int unsigned edg_x  = pow (ca2d.sts,  (ca2d.ngb.y-1)       *((ca2d.ngb.x-1)+siz.x));
const int unsigned edg_y  = pow (ca2d.sts, ((ca2d.ngb.y-1)+siz.y)* (ca2d.ngb.x-1)       );

// compact list of edges
int unsigned edg_n;

// memory allocation for preimage network
mpz_t net [2] [siz.y+1] [edg_x];

// initialize array variable
for (int d=0; d<2; d++) {
        for (int y=0; y<=siz.y; y++) {
                for (int unsigned edg=0; edg<edg_x; edg++) {
                        mpz_init (net [d] [y] [edg]);
                }
        }
}
// initialize starting edge to unit (open edge)
for (int unsigned edg=0; edg<edg_x; edg++) {
        mpz_set_ui (net [0] [0    ] [edg], 1);
        mpz_set_ui (net [1] [siz.y] [edg], 1);
}
// compute network weights in both directions
for (int y=0; y<siz.y; y++) {
        // loop over all edges
        for (int unsigned edg=0; edg<edg_x; edg++) {
                // only process edge if it's weight is not zero
                if (mpz_sgn (net [0] [      y] [edg]) > 0) {
                        ca1d_net(ca2d, siz.x, ca [       y], 0, edg, net [0] [      y] [edg], &edg_n, net [0] [      (y+1)]);
                }
                if (mpz_sgn (net [1] [siz.y-y] [edg]) > 0) {
                        ca1d_net(ca2d, siz.x, ca [siz.y-1-y], 1, edg, net [1] [siz.y-y] [edg], &edg_n, net [1] [siz.y-
                (y+1)]);
                }
        }
}
// count all preimages
for (int unsigned d=0; d<2; d++) {
        mpz_init   (cnt [d]);
        for (int unsigned edg=0; edg<edg_x; edg++) {
                mpz_add (cnt [d], cnt [d], net [d] [d ? 0 : siz.y] [edg]);
        }
}

// allocate memory for preimages described by edges
mpz_t weight;
mpz_init (weight);
int unsigned lst [mpz_get_ui(cnt[0])] [siz.y+1];

// initialize list of 2D network preimages
int unsigned p = 0;
for (int unsigned edg=0; edg<edg_x; edg++) {
        int unsigned max;
        max = mpz_get_ui(net[1][0][edg]);
        for (int unsigned i=0; i<max; i++) {
                lst [p] [0] = edg;
                p++;
        }
}
// list 2D network preimages
// memory allocation for preimage network
mpz_t edges [edg_x];
// initialize array variable
for (int unsigned edg=0; edg<edg_x; edg++) {
			mpz_init (edges [edg]);
		}
		for (int y=1; y<=siz.y; y++) {
			int unsigned p = 0;
			while (p < mpz_get_ui(cnt[0])) {
				// gain process 1D preimage for current edge (lst [p] [y])
				mpz_set_ui (weight, 1);
				// re initialize edges
				for (int unsigned edg=0; edg<edg_x; edg++) {
					mpz_set_ui (edges [edg], 0);
				}
				ca1d_net (ca2d, siz.x, ca [y-1], 0, lst [p] [y-1], weight, &edg_n, edges);
				for (int unsigned edg=0; edg<edg_x; edg++) {
					mpz_set (weight, edges[edg]);
					if ((mpz_sgn (weight) > 0) && (mpz_sgn (net[1][y][edg]) > 0)) {
						for (int unsigned i=0; i<mpz_get_ui(net[1][y][edg]); i++) {
							lst [p] [y] = edg;
							p++;
						}
					}
				}
			}
		}

		// allocate memory for preimages
		ca2d_size_t siz_pre = {siz.y+ca2d.ver.y, siz.x+ca2d.ver.x};
		siz_pre.a = siz_pre.y * siz_pre.x;
		*p_list = (int unsigned (*) [] [siz_pre.y] [siz_pre.x]) malloc (sizeof(int unsigned) * siz_pre.a * mpz_get_ui(cnt[0]));
		// convert edge list into actual preimage list
		for (int unsigned i=0; i<mpz_get_ui(cnt[0]); i++) {
			ca2d_size_t siz_lin0 = (ca2d_size_t) {ca2d.ver.y, siz.x+ca2d.ver.x};
			int unsigned line0 [siz_lin0.y] [siz_lin0.x];
			ca2d_array_from_ui (ca2d.sts, siz_lin0, line0, lst[i][0]);
			ca2d_array_fit (siz_pre, (ca2d_size_t) {0, 0}, siz_lin0, (**p_list)[i], line0);
			for (int unsigned y=0; y<siz.y; y++) {
				ca2d_size_t siz_lin = (ca2d_size_t) {1, siz.x+ca2d.ver.x};
				int unsigned line [siz_lin.y] [siz_lin.x];
				ca2d_array_from_ui (ca2d.sts, siz_lin0, line0, lst[i][y+1]);
				ca2d_array_slice   (siz_lin0, (ca2d_size_t) {ca2d.ver.y-1, 0}, siz_lin, line0, line);
				ca2d_array_fit (siz_pre, (ca2d_size_t) {y+ca2d.ver.y, 0}, siz_lin, (**p_list)[i], line);
			}
		}

		printf ("DEBUG: end of network\n");
		return (0);
}
```

LITERATURE

[1] Conway's Reverse Game of Life. https://www.kaggle.com/c/conway-s-reverse-game-of-life, Marec 2013.

[2] LifeWiki: Flower of Eden. http://www.conwaylife.com/wiki/Flower_of_Eden, August 2016.

[3] The On-Line Encyclopedia of Integer Sequences: A196447. https://oeis.org/A196447, August 2016.

[4] Wikipedia: Backtracking. https://en.wikipedia.org/wiki/Backtracking, August 2016.

[5] Wikipedia: Boolean satisfiability problem. https://en.wikipedia.org/wiki/Boolean_satisfiability_problem, August 2016.

[6] Wikipedia: Brute-force search. https://en.wikipedia.org/wiki/Brute-force_search, August 2016.

[7] Wikipedia: Conway's Game of Life. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life, August 2016.

[8] Wikipedia: De Bruijn graph. https://en.wikipedia.org/wiki/De_Bruijn_graph, August 2016.

[9] Wikipedia: Depth-first search. https://en.wikipedia.org/wiki/Depth-first_search, August 2016.

[10] Wikipedia: Garden of Eden (cellular automaton). https://en.wikipedia.org/wiki/Garden_of_Eden_(cellular_automaton), August 2016.

[11] Wikipedia: Langton's loops. https://en.wikipedia.org/wiki/Langton%27s_loops, August 2016.

[12] Wikipedia: Rule 110. https://en.wikipedia.org/wiki/Rule_110, August 2016.

[13] Wikipedia: Von Neumann universal constructor. https://en.wikipedia.org/wiki/Von_Neumann_universal_constructor, August 2016.

[14] Vladimir Batagelj. Efficient Algorithms for Citation Network Analysis. *CoRR*, cs.DL/0309023, 2003. Available at http://arxiv.org/abs/cs.DL/0309023.

[15] Neil Bickford. Reversing the Game of Life for Fun and Profit. https://nbickford.wordpress.com/2012/04/15/reversing-the-game-of-life-for-fun-and-profit/, April 2012.

[16] Peter Borah. Atabot. https://github.com/PeterBorah/atabot, 2013.

[17] Bryan Duxbury. Cellular Chronometer. https://github.com/bryanduxbury/cellular_chronometer/tree/master/rb, September 2013. Article: https://bryanduxbury.com/2013/09/02/cellular-chronometer-part-1-reversing-the-game-of-life/.

[18] Yossi Elran. Retrolife and The Pawns Neighbors. *The College Mathematics Journal*, 43(2):147–151, 2012. Available at http://www.jstor.org/stable/10.4169/college.math.j.43.2.147.

[19] Achim Flammenkamp. Garden of Eden/Orphan. http://wwwhomes.uni-bielefeld.de/achim/orphan.html, August 2016.

[20] Dave Greene. New Technology from the Replicator Project. http://b3s23life.blogspot.si/2013/11/new-technology-from-replicator-project.html, November 2013. Accessed: 2-aug-2016.

[21] Jean Hardouin-Duparc. À la recherche du paradis perdu. *Publ. Math. Univ. Bordeaux Année*, 4:51–89, 1972-1973.

[22] Jean Hardouin-Duparc. Paradis terrestre dans l'automate cellulaire de Conway. *Rev. Française Automat. Informat. Recherche Operationnelle Ser. Rouge*, 8:64–71, 1974. Available at http://archive.numdam.org/ARCHIVE/ITA/ITA_1974__8_3/ITA_1974__8_3_63_0/ITA_1974__8_3_63_0.pdf.

[23] Christiaan Hartman, Marijn J. H. Heule, Kees Kwekkeboom, and Alain Noels. Symmetry in Gardens of Eden. *Electronic Journal of Combinatorics*, 20, 2013.

[24] ionreq. calculate previous generations for Conway's Game of Life. https://github.com/ionreq/GoLreverse, 2013. Video post: https://www.youtube.com/watch?v=ZOuMnaarI5k.

[25] Iztok Jeras. Solving cellular automata problems with SAGE/Python. Published in Andrew Adamatzky, Ramón Alonso-Sanz, Anna T. Lawniczak, Genaro Juárez Martínez, Kenichi Morita, and Thomas Worsch, editors, *Automata 2008: Theory and Applications of Cellular Automata, Bristol, UK, June 12-14, 2008*, pages 417–424. Luniver Press, Frome, UK, 2008. Available at http://uncomp.uwe.ac.uk/free-books/automata2008reducedsize.pdf.

[26] Iztok Jeras. 2D cellular automata preimages count&list algorithm. https://github.com/jeras/preimages-2D, 2016.

[27] Iztok Jeras. Cellular Automata SAGE Toolkit. https://github.com/jeras/cellular-automata-sage-toolkit, 2016.

[28] Iztok Jeras. WebGL QUAD CA sumulator. https://github.com/jeras/webgl-quad-ca, 2016.

[29] Iztok Jeras and Andrej Dobnikar. Cellular Automata Preimages: Count and List Algorithm. Published in Vassil N. Alexandrov, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2006, 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part III*, volume 3993 of *Lecture Notes in Computer Science*, pages 345–352. Springer, 2006. Available at http://dx.doi.org/10.1007/11758532_47.

[30] Iztok Jeras and Andrej Dobnikar. Algorithms for computing preimages of cellular automata configurations. *Physica D Nonlinear Phenomena*, 233:95–111, September 2007.

[31] Jarkko Kari. Reversibility of 2D cellular automata is undecidable. *Physica D: Nonlinear Phenomena*, 45:379–385, 1990.

[32] Paulina A. Léon and Genaro J. Martínez. Describing Complex Dynamics in Life-Like Rules with de Bruijn Diagrams on Complex and Chaotic Cellular Automata. *Journal of Cellular Automata*, 11(1):91–112, 2016.

[33] Harold V. McIntosh. Linear Cellular Automata via de Bruijn Diagrams. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.8763&rep=rep1&type=pdf, August 1991.

[34] E. F. Moore. Machine models of self-reproduction. Published in *Mathematical Problems in Biological Sciences (Proceedings of Symposia in Applied Mathematics)*, volume 14, pages 17–33. American Mathematical Society, 1962.

[35] John Myhill. The converse of Moore's Garden-of-Eden theorem. Published in *Proceedings of the American Mathematical Society*, volume 14, pages 685–686. American Mathematical Society, 1963.

[36] Florian Pigorsch. A SAT-based forward/backwards solver for Conway's "Game of Life". https://github.com/flopp/gol-sat, October 2015.

[37] Edward Powley. QUAD Prize Submission: Simulating Elementary CAs with Trid CAs. *Journal of Cellular Automata*, 5:415–417, 2010. Available at http://uncomp.uwe.ac.uk/automata2008/files/quadprize_powley.pdf.

[38] Paul Rendell. A Turing Machine In Conway's Game Life. https://www.ics.uci.edu/~welling/teaching/271fall09/Turing-Machine-Life.pdf, August 2001. Accessed: 2-aug-2016.

[39] Chris Salzberg and Hiroki Sayama. Complex genetic evolution of artificial self-replicators in cellular automata. *Complexity*, 10:33–39, 2004. Available at http://www3.interscience.wiley.com/journal/109860047/abstract.

[40] José Manuel Gómez Soto. Computation of Explicit Preimages in One-Dimensional Cellular Automata Applying the De Bruijn Diagram. *Journal of Cellular Automata*, 3:219–230, 2008.

[41] Tommaso Toffoli. Background for the Quad Prize. http://uncomp.uwe.ac.uk/automata2008/files/quad.pdf, February 2008.

[42] Erik P. Verlinde. On the origin of gravity and the laws of Newton. 2010.

[43] Robert Wainwright. Lifeline newsletter - volume 3. http://www.conwaylife.com/w/index.php?title=Lifeline_Volume_3, September 1971.

[44] Robert Wainwright. Lifeline newsletter - volume 4. http://www.conwaylife.com/w/index.php?title=Lifeline_Volume_4, December 1971.

[45] Christopher Wellons. WebGL Game of Life. https://github.com/jeras/webgl-quad-ca, 2014.

[46] Andrew Wuensche. Discrete Dynamics Lab. https://www.ddlab.com, 2016.

[47] Andrew Wuensche and Mike Lesser. The Global Dynamics of Cellular Automata. Santa Fe Institute, 1992. Available at http://uncomp.uwe.ac.uk/wuensche/gdca.html.