Application of Cellular Automata for Generation of Chess Variants

Mikael Fridenfalk
Uppsala University Campus Gotland
Department of Game Design
621 55 Visby, Sweden
mikael.fridenfalk@speldesign.uu.se

***Abstract*—A system was developed for the automatic generation of chess variants in a computer game. The system is able to generate 5000 relatively unique board configurations using a modular cellular automaton based on a new variation of Conway's Game of Life in combination with modular constraints.**

Keywords-artificial intelligence; cellular automata; chess variants; game design; Game of Life; modular

I. Introduction

Ever since Conway's *Game of Life* was introduced in 1970 [3], the interest for cellular automata has constantly increased, yielding interesting results [4,11] and applications within many areas, ranging from pattern generation in arts such as music composition [2,6,8] to less obvious applications such as language recognition [9]. Today, there exist a large variety of cellular automata, of which many consist of special cases of Conway's Game of Life [1,5,7,10]. The rule of Conway's cellular automaton Game of Life can be expressed as: a cell is set to 1 if it has 3 neighbors and to 0 if it has 0-1 or 4-8 neighbors. Game of Life can be classified as a binary automaton in a Moore neighborhood, where each cell has eight surrounding neighbors in a 2D integer lattice.

In the software system described in this paper a number of artificial intelligence (AI) systems were integrated to generate a computer chess variant AI. A chess variant is known as a game that is a variation of regular chess with respect to mainly the set of rules, piece-sets or board-configurations. The aim of this paper is the presentation of one of these subsystems, using modular binary 2D cellular automata based on a new set of rules, or more specifically a new variation of the Game of Life to generate patterns that could be used as design elements in applications such as computer games or user interfaces.

The literature study performed by the author suggests that the game generation system presented in this paper is novel on two counts. To begin with, there exist today a very large number of chess variants based on other board configurations than the standard 8×8, but none seem to contain void squares within the board. Secondly, the generation of the board configurations are here based on a new type of cellular automaton, defined as one with a high reproduction rate constrained by relatively small modular grids. In other words, although modular constraints are occasionally used in context with 2D cellular automata, they are in practice never used in combination with high reproduction rate, neither applied to relatively small-sized grids. The work that was identified as the closest to the one presented in this paper is [1], which is based on 2D cellular automata with high reproduction rate, but without modular constraints.

The result of the implementation of the cellular automaton in this work led to the development of a commercial computer game by the author that was released in 2011. The game contains 5000 chess variants of which five are displayed in Figs. 1-5. The index number in each of these figures corresponds to the chess variant number in the game.

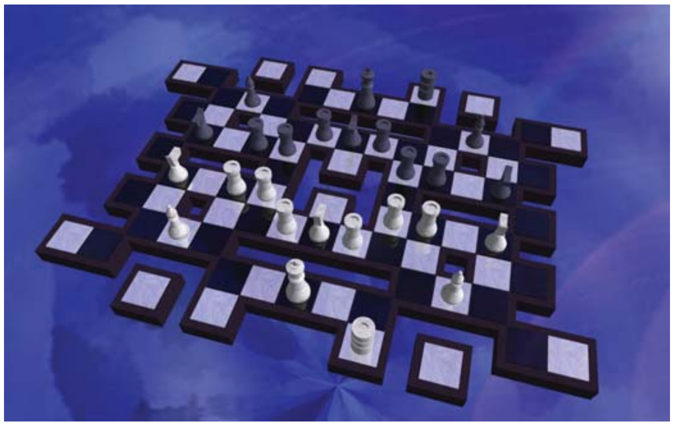
Figure 1: Index 1

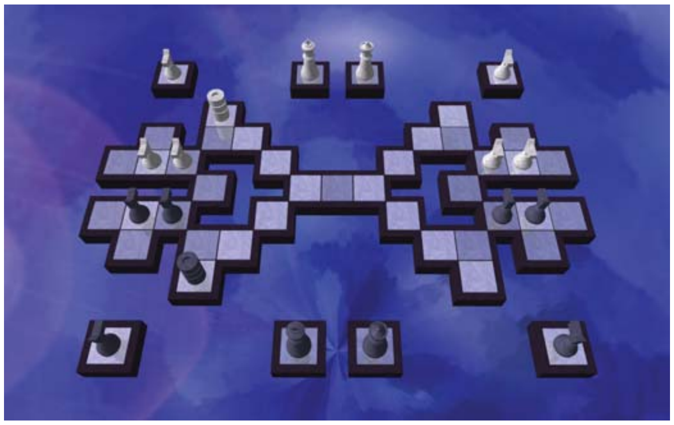
Figure 2: Index 950

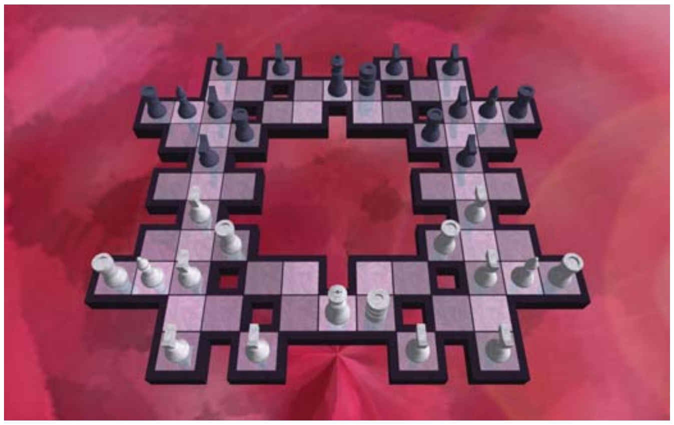
Figure 3: Index 4805

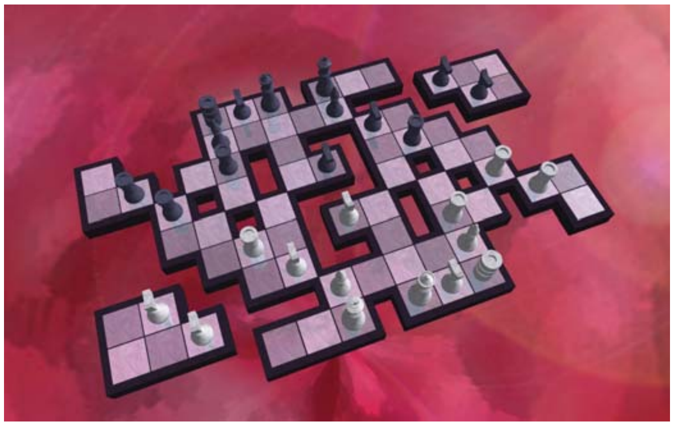
Figure 4: Index 137

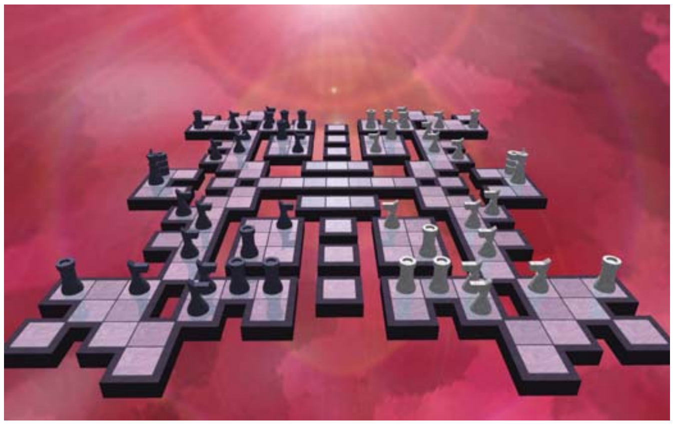
Figure 5: Index 4478

Here, the rules governing the pieces are essentially the same as in chess. In addition, a new piece is introduced called the "pioneer", with the ability to add new squares to the board and thereby bridge "void squares" or gaps. The pioneer is able to move only one step at a time in the directions forward, back, right and left and may be captured like any other chess piece.

The goal of the game is for the player to checkmate the opponent and since the gaps in the board prevents any piece to make a move that includes passing or entering a gap, thus only the pioneers and the knights are able to pass gaps. A knight is able to make any regular jump over gaps, as long as the destination square is either solid or not occupied by a piece of the same color.

II. Software System

The game generation system was developed in C++ using Apple Xcode [12]. By setting a preprocessor flag to true, the source code of the application itself can be modified by the generation of the chess variant boards. To generate the end product the flag is set to false before compilation. This proved to be a well working solution for the development, modification and the final selection of the chess variants, since automatic play by the AI itself could give a preliminary assessment on the basic playability of each chess variant.

To optimize the execution speed and simplify program structure a straightforward class hierarchy was adopted consisting of five classes: Common, Logic, GFX, NetX and GUI. The Common class is a static class that only contains constants and static member variables and methods that can be accessed by all classes. The class Logic is where the essential logic operations are performed, such as alpha-beta pruning and evolutionary generation of chess variant boards.

An object each of Logic, GFX and NetX are created within the GUI object by allocation. The classes Logic, GFX, NetX and GUI are subclasses of Common. Thus any call to the classes Logic, GFX or NetX from GUI has to begin with the corresponding object name plus a pointer denotation, which in practice results in a straightforward code syntax. The GFX class is an OpenGL-based graphics class that produces most of the drawings in the application. The network class NetX is used for peer-to-peer online gaming, and also to hook up with a central web server to find friends by username instead of an IP address.

While the Common class is the communication hub of the system, the GUI class functions as the execution agent and integrates all the functionalities of the classes and thus most high-level methods were located in GUI.

The game generation system consists of three principal AI subsystems. The first subsystem consists of a mathematical pattern generator that creates 2D matrices of integer numbers combined with a subsequent mapping and piece placement function that converts these integers to an actual chess variant board. The second AI system tests the produced boards to make sure that basic standards are met, such as the originality of a board compared to previously approved boards, the placement of pieces and finally playability, using the third AI system as a subsystem. The third AI system is a computer chess core that is the only AI component that is included in the final product. This component enables the player to play against the computer or watch a game between two computer players.

III. Cellular Automata

In this work, five cellular automata, each consisting of a new version of Conway's Game of Life were implemented and tested. Only rule 1 (see below) was implemented in the game, which showed to be sufficient for the generation of the chess variants. The rules are formulated according to below:

1) A cell is set to 1 if it has 2-3 neighbors, else to 0
2) A cell is set to 1 if it has 2-4 neighbors, else to 0
3) A cell is set to 1 if it has 2-5 neighbors, else to 0
4) A cell is set to 1 if it has 2-6 neighbors, else to 0
5) A cell is set to 1 if it has 2-3 or 5 neighbors, else to 0

In a second step, the evolution space for these automata is constrained by small modular integer lattices. Figs. 7-8 display the evolution of two cellular automata based on rule 1 using identical kernels (start configurations, also called start positions), but of different grid size. Since the same pattern must sooner or later emerge in any finite binary grid, such system will eventually either converge into a static configuration (where all cells are for instance equal to zero) or a cyclic loop where a number of patterns, by definition equal or larger than two, are endlessly repeated (as from a point of view a static configuration could be interpreted as a cyclic loop with a period of a single generation).

In a third step, a variety of kernels can be used to initiate the cellular automata. Only a single cross-shaped kernel was used for the generation of Figs. 7-14 except for Figs. 9 and 13, which were based on an asymmetric kernel. In the game, over 50 kernels were designed of which 24 were finally selected for the generation of the chess variant boards.

In a fourth step, to be able to map the generated patterns more or less directly into chess variant boards, an accumulation grid is evaluated for each generation as the sum of all 20 past generations, resulting in a grid of cells with integer numbers within the interval of 0-20. The present generation is then used as a stencil layer with respect to the location of its void squares to create the void square configuration of the final board. By mapping these numbers, see Table I, a basic template of a chess variant is proposed. A procedural algorithm is then applied to place kings, queens and pioneers.

In a fifth and final step, the game opening is tested by the strategic chess variant AI system, making sure that the game is playable. According to the produced logs, about 12% of the proposed chess variants failed the final verification tests and were excluded from the final product.

As previously mentioned, depending on grid size and kernel, a cellular automaton produces as a rule a number of unique patterns before it reaches an end point, in practice defined by generation of either zero matrices or repetition of specific patterns in an infinite loop. From this perspective it is important to monitor the pattern generation process to stop the generation at an end point. This process can progressively decrease generation speed. In one experiment (generating large sized grids, compared to average sized) a kernel produced more than 100,000 unique patterns before the generation was stopped due to incrementally longer execution time for each generation.

The generation of a few thousand patterns of moderate size showed however to be of no issue. According to a few experiments, it took typically 0.8 seconds on a 2.66 GHz Intel Core i7 processor to generate the game boards in the final product, including the application of the strategic AI component.

IV. Conclusion

A new type of cellular automaton is presented in this paper, characterized by (1) a 2D cellular automaton using a relatively high reproduction rate compared with the Game of Life, in combination with (2) a relatively small-sized modular lattice for the continuous transformation of the patterns. The kernel seems in addition to be as important in this context for the pattern generation process as the system itself.

Acknowledgment

This work was supported by Almi Gotland. A great thanks to Ellinor Dahl at Almi for the encouragement along the way from idea to the final product.

References

[1] R. Alonso-Sanz, “Cellular Automata with Memory”, Computational Complexity, Springer, pp. 382-406, 2012.
[2] C. Ariza, “Automata Bending: Applications of Dynamic Mutation and Dynamic Rules in Modular One-Dimensional Cellular Automata”, Computer Music Journal, vol. 31, no. 1, pp. 29-49, Spring 2007.
[3] M. Gardner, “Mathematical Games: The Fantastic Combinations of John Conway’s New Solitaire Game ’Life”’, Scientific American, vol. 223, no. 4, pp. 120-123, October 1970.
[4] J. Kari, “Reversibility of 2D Cellular Automata is Undecidable”, Physica D 45, pp. 379-385, 1990.
[5] G. J. Martinez, A. Adamatzky, K. Morita, and M. Margenstern, “Computation with Competing Patterns in Life-Like Automaton”, Game of Life Cellular Automata, Springer, pp. 547-572, 2010.
[6] E. Miranda, “Cellular Automata Music: From Sound Synthesis to Musical Forms”, Evolutionary Computer Music, pp. 170-193, 2007.
[7] F. Peper, S. Adachi, and J. Lee, “Variations on the Game of Life”, Game of Life Cellular Automata, Springer, pp. 235-255, 2010.
[8] S. Phon-Amnuaisuk, “Investigating Music Pattern Formations from Heterogeneous Cellular Automata”, Journal of New Music Research, vol. 39, no. 3, pp. 253-267, 2010.
[9] A. R. Smith, “Real-Time Language Recognition by One-Dimensional Cellular Automata”, J. ACM, vol. 6, pp. 233-253, 1972.
[10] G. Tempesti, D. Mange, A. Stauffer, “Self-Replication and Cellular Automata”, Computational Complexity, Springer, pp. 2792-2809, 2012.
[11] S. Wolfram, A New Kind of Science, Wolfram Media, Inc., May 14, 2002.
[12] Xcode, Apple Inc., May 14, 2013 <https://developer.apple.com/xcode/>.

```
#define For(i,N)  for (int (i) = 0; (i) < (N); (i)++)
...
void Logic::CA_Rule1(int idx){

    const int NX[][2] = {
        {1,0},{1,1},{0,1},{-1,1},{-1,0},{-1,-1},{0,-1},{1,-1}};
    For (x,mX) For (y,mY){
        int neighbors = 0;
        For (i,8){
            int x1 = x + NX[i][0], y1 = y + NX[i][1];
            x1 = (mX + x1) % mX; y1 = (mY + y1) % mY;
            if (mCA[idx-1][x1][y1] != 0) neighbors++;
        }
        mCA[idx][x][y] = int(neighbors == 2 || neighbors == 3);
    }
}
```

Figure 6: An example of the implementation of rule 1 in C++ for idx (grid index number) with respect to previous generation, $idx - 1$, using a rectangular modular grid of size mX $\times$ mY.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$\times$	$\times$	♙	$\times$	$\times$	♗	♖	♘	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
$\times$	$\times$	$\times$	$\times$	♙	$\times$	$\times$	♗	♖	♘	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	♖	♘	♗	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$

Table I: Example of three heuristic mapping schemes for the initial placement of chess pieces in the computer game generation system. By summation of the last 20 generations, where a void square is equal to an integer value of zero and a solid a value of one, it is possible to map numbers directly to chess pieces such as pawns, bishops, knights and rooks. In this mapping scheme the token $\times$ denotes an empty square (or a void square if current generation contains a void in this specific location of the board).

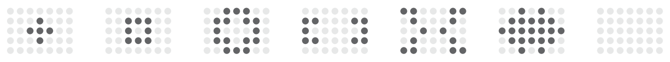

Figure 7: Example of a 7×5 grid that ends after 6 generations.

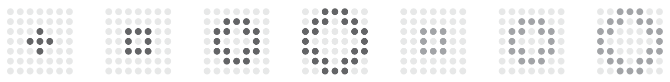

Figure 8: Example of a 7×7 grid that falls into a cycle after 4 generations.

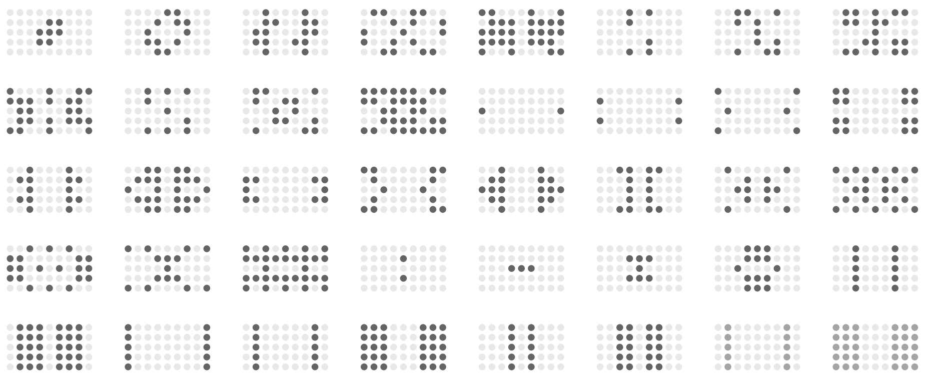
Figure 9: Example of a 9×5 grid that becomes symmetric after 12 generations and falls into a cycle after 38 generations.

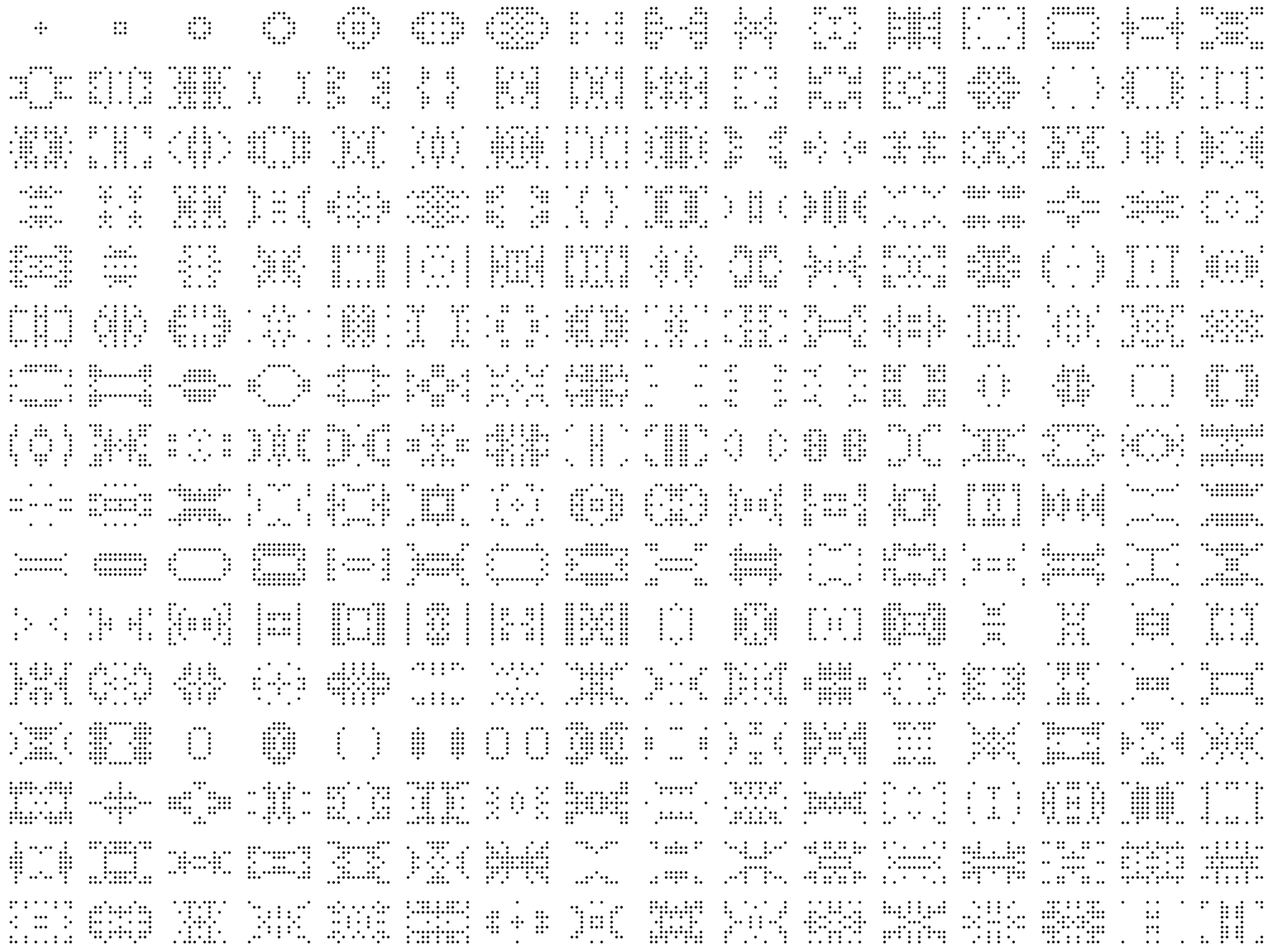
Figure 10: A 13×9 grid evolved from generation 0 to 255.

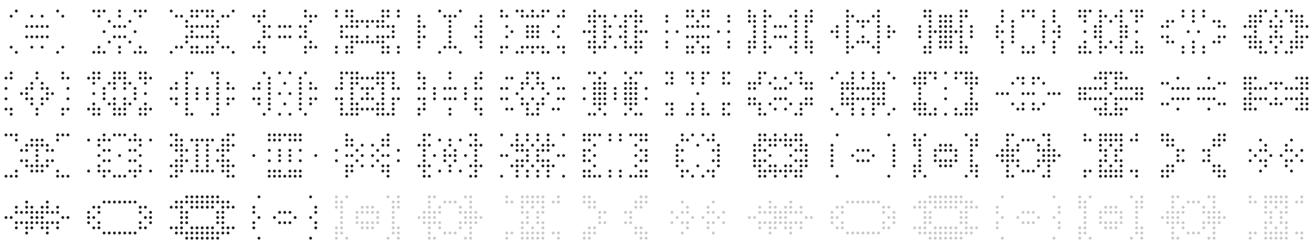
Figure 11: The 13×9 grid from previous figure, evolved from generation 800 to 843, where it falls into a cycle.

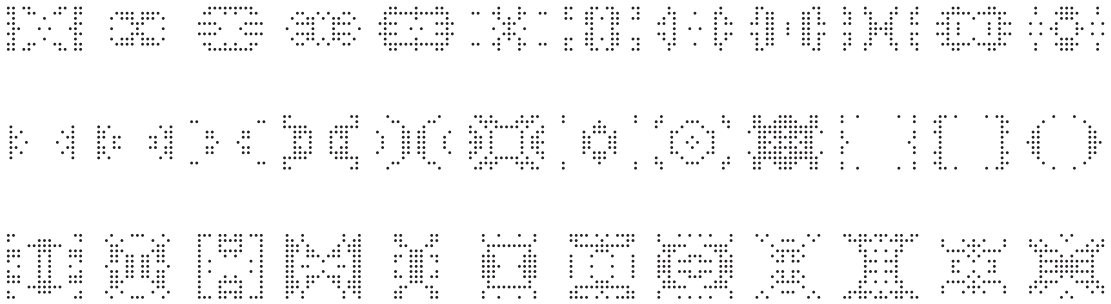
Figure 12: A 15×9 grid (above), 15×11 grid (middle), and 15×13 grid (below), evolved from generation 100 to 111.

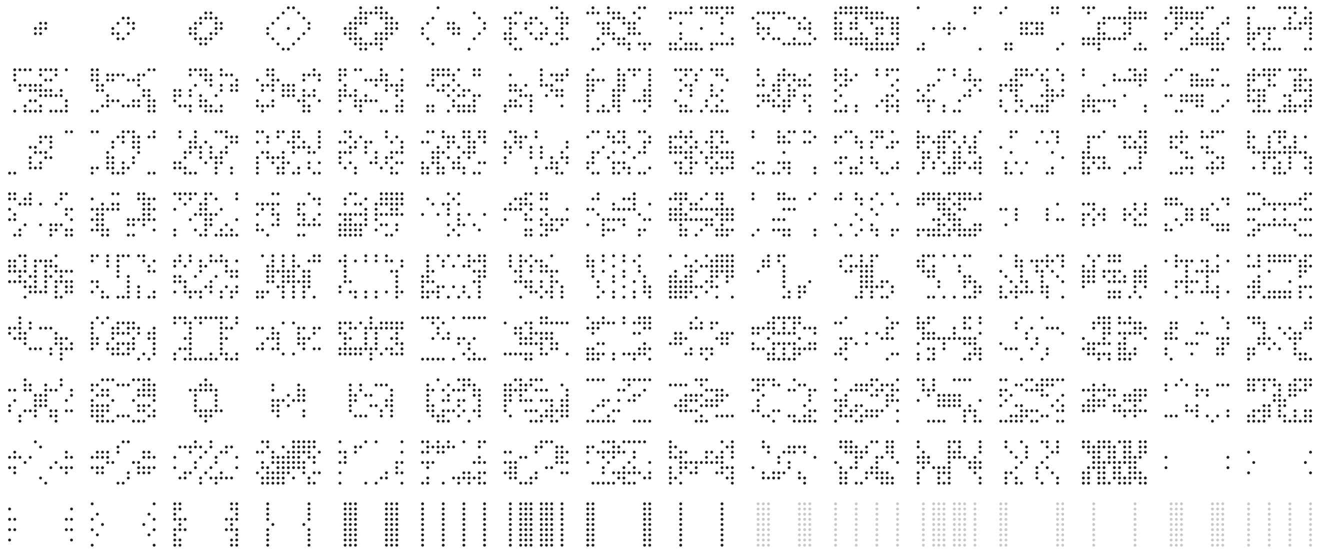
Figure 13: A 13×9 grid using an asymmetric kernel evolved from generation 0 to 136, where it falls into a cycle.

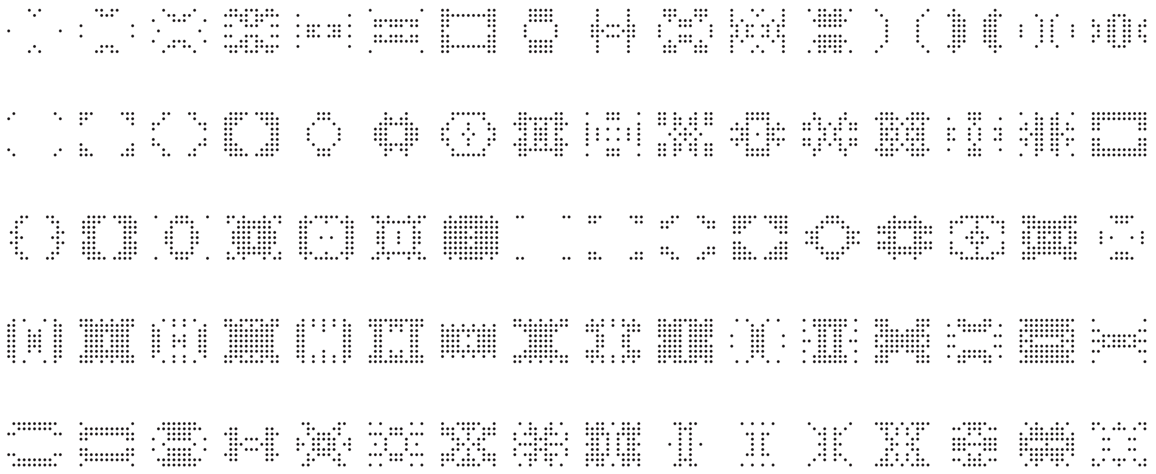

Figure 14: The figures in this paper are exclusively based on rule 1, except for this figure which represents an 11×9 grid based on rules 1-5 (from above to below), evolved from generation 48 to 63.