Proceedings of the 11th International Conference on Applied Informatics
Eger, Hungary, January 29–31, 2020, published at http://ceur-ws.org

Statistical Analysis of Hexagonal and Triangular Game of Life*

Erik Zoltán Hidi, Géza Horváth, Dávid Petrik

University of Debrecen, Faculty of Informatics, H-4028 Debrecen, Kassai Road 26.
hidieric@gmail.com, horvath.geza@inf.unideb.hu, david.petrik01@gmail.com

Abstract

We discuss two different implementations of John Conway's Game of Life. The first variant is based on an environment consisting of hexagon shaped cells. Since 6 neighbors (the actual number of neighbors a hexagonal cell has) does not seem to be enough to support "life", we use 2 additional neighbors. The second variant is based on triangular cells where every cell has 12 neighbors, so the complexity of the rule-systems is greater than in the original square grid implementation. We discuss the properties of several rule-systems which are chosen as the results of a long procedure in which we have run simulations with several possible systems and compared their statistics afterwards. We also present an algorithm which can be used to recognize special patterns called oscillators and gliders.

Keywords: cellular automata, Game of Life, triangular grid, hexagonal grid

MSC: 37B15, 68Q80

CEUR-WS.org/Vol-2650/paper16.pdf

1. Introduction

A cellular automaton [6, 5, 3] is an automaton in which a set of parameters are used to initialize the environment consisting of cells and simple rules used to evolve this particular environment. Simulations created by CA have certain similarities, such as:

– Their environment is a grid of cells, where every cell has a state.

– The environment evolves (changes) over time.

*This work was supported by the construction EFOP-3.6.3-VEKOP-16-2017-00002. The project was supported by the European Union, co-financed by the European Social Fund.
Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

– The change is based on the rules, they describe how cells interact with each other.

The type and form of cells, the number of states and neighbors, the rules and the type of grid differ in a very wide range letting us to play with a huge variety of them. This is the main reason why cellular automata can be used to simulate systems of different branches of science. John Conway's Game of Life might be the most popular cellular automaton [2]. Its environment is a grid of squares, which means every cell has eight neighboring cells. It uses a specific rule-system, a set of simple rules, to simulate a changing environment:

– Any living cell with less than two living neighbors dies.

– Any living cell with four or more living neighbors dies.

– Any living cell with two or three live neighbors lives on to the next generation.

– Any dead cell with three living neighbors becomes alive.

Also, worth to mention, that there are oscillators spawning in these simulations. An oscillator is a structure, which repeats itself after a fixed number of generations (known as it's period). There are two types of special oscillators: still life (static, its period equals to one) and glider (moves through the grid).

2. Preliminaries on hexagonal Game of Life

The hexagonal version of Game of Life has been investigated in the past, but it does not look anything like life [1]. In our research we were looking for answer to the following question: Why does Game of Life work on a square grid, and not on a hexagonal grid? Our conjecture was that the main difference between the two cases is the number of neighbors. Maybe 6 neighbors are too few to have enough variety. We have decided to create Game of Life on a hexagonal grid with 8 "neighbors".

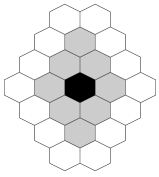

Figure 1: 8 neighbors of the cell in question on hexagonal grid

In the hexagonal implementation of Game of Life originally each cell has six neighboring cells, but in our case, we include two more. (We can suppose that the cell can interact a bit further in two directions.) It means that the cells on the top and bottom (which has two "neighbors", which are "neighbors" to the given cell) are included to the neighborhood as well. To be clear, Figure 1 shows a visual explanation.

In the paper [4] the authors have already performed several analyses of the given implementation of Game of Life. To investigate the automaton, we had a vital need for a data source, which can generate data continuously. For this purpose, we developed an algorithm, which is able to execute simulations based on the set of parameters given as input. The mentioned algorithm describes the behavior of Game of Life with hexagon shaped cells. It requires the following parameters:

- *Size:* two integer values, representing the width and height of the environment.
- *Radius of interaction:* an integer, which is representing the length of the radius of a circle. The center of this circle equals to the cell in question. Every cell inside of this circle should interact with the cell in question, therefore these cells are the neighbors of the cell in question. In specific cases (like hexagonal grid with 8 neighbors) there is an alternative parameter to the radius of interaction: a set of relative coordinates of the neighbors. Thus, one can define the neighborhood in a very specific way: giving the relative coordinates (relative to the cell in question) of the chosen neighbors.
- *Rule-system:* a set of rules describing when should a cell switch to live state, keep the current stata or when should it switch to dead state. The representation should follow this pattern:
  $Bx_1, x_2, \ldots, x_n/Sy_1, y_2, \ldots, y_m/Dz_1, z_2, \ldots, z_k$, where the sets of integers are the number of living neighbors required for a cell to be born (B = born), remain dead/alive, (S = stable), to die (D = die).
- *Initial AR (alive-ratio):* an integer number between 0 and 100, which represents the percentage of living cells in generation 0.
  $\mathrm{AR} = \frac{\text{number of living cells}}{\text{number of cells}} 100\%$ rounded.
- *Last generation:* an integer used for identifying when the algorithm should stop the simulation. When the current generation number equals to this the simulation is stopped.

The algorithm was implemented in Java. By default, it gathers and saves data about the number of cells alive and dead but can be configured to save the whole environment as a matrix. For human observation we have developed a graphical interface, which displays the simulation.

As we realized, that the hexagonal version of GoL with 8 neighbors also has the potential to provoke oscillators and gliders, we decided to develop some kind of method, which can recognize them through the simulations.

In the paper [4] authors have discussed the following rule-systems:

- **B3/S2/D0145678:** original Game of Life rule-system, supports oscillators (Figure 2 for reference). By applying this system, we get a simulation, which has a stabilizing population. This means, that it slowly becomes a still life. The duration of the mentioned process depends on the size of the environment (how many cells are involved).

– **B36/S2/D014578:** similar to the original version, supports oscillators as well (Figure 2 for reference). The main difference between this and the original system is that the population is not stabilizing. It does have an interval of AR ($\approx 25-35\%$), but it never stops changing. May support other oscillators, than the original rule-system.

– **B3678/S4/D0125:** interesting behavior, supports oscillators (Figure 3 for reference), has a dying population. At the beginning the population may grow (up to 10% of growth), in this phase the cells are creating groups. These groups are merging till there are no direct interactions with other groups. Then every group starts to shrink (number of living cells are decreasing). At the end the number of living cells equals to 0 (if we do not count the oscillators remaining).

Applying these rules the authors have found several oscillators just by visualizing the simulation. The mentioned oscillators and gliders can be seen on Figures 2 and 3.

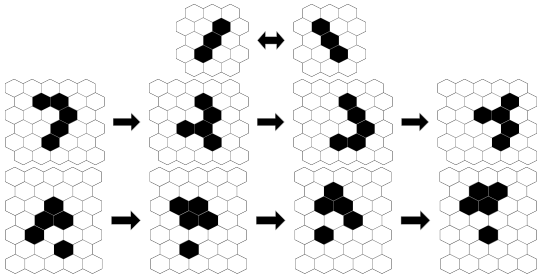

Figure 2: oscillators B3/S2/D0145678 and B36/S2/D014578

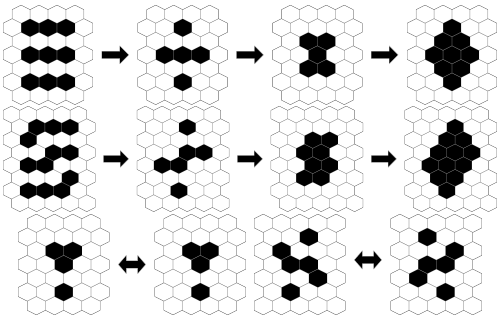

Figure 3: oscillators B3678/S4/D0125

Further information and statistical data are provided in the paper [4].

3. The oscillator finder algorithm

3.1. Ideas, pros and cons

The main idea of the algorithm is to detect every separate cell cluster then compare it to every other cluster. If there is a match, then we observe the lifecycle of the given structure. The lifecycle can be periodic, if so then it is either an oscillator or a glider. Since the gliders are moving the program should compare a cluster in question to every other clusters. If we compare the cluster with clusters on the same location, then this method loses the capability of detecting gliders. As we can suspect the given method requires enormous computational power, but it is capable of detecting oscillators and gliders in any size. As we already know Game of Life simulations are full of cell clusters, which results in a relatively slow operation. We could boost the speed by turning off the glider detecting function (as we described earlier), but we would not like to lose this essential ability. Involving a database could improve the algorithm. The database should contain every pattern, which was already met during simulations categorized by type (still life, oscillator, glider, none). The algorithm detects a cluster, then compares it to the ones in the database. If there is a match, then there is no need to observe the detected pattern any further. If no match is detected, then it should be further observed and should also be put in the database.

The next idea of the algorithm is to generate every possible pattern of living cells of given size. Put this pattern in a simulation (every other cell should not be in living state), then run the simulation. Then the initial pattern should be compared to every evolved pattern (from the following generations). If there is a match, then we should observe the lifecycle. If it is periodic, then we have either an oscillator or a glider. It is a simpler method than the previous one and requires less computational power. The issue with this one is that it can detect oscillators effectively with small dimensions. If we use this method on an average PC, we should not look for oscillators bigger than 6x6.

3.2. Implementation

We chose the second algorithm for implementation. Our goal is to find every oscillator not bigger than 5x5 in size. For the given size the second algorithm is the optimal solution.

The implemented algorithm requires the following parameter list:

- *Size, radius of interaction, rule-system, last generation:* same as in the previous program.
- *Size of patterns:* width and height of the initial patterns.
- *Input file:* a text file which contains the initial patterns. Patterns are represented in binary form, where '1' means that the cell is alive and '0' means that it is dead.

– *Output file:* a text file which will contain the potential oscillators. The representation of the patterns is the same as in the input file.

The number of permutations (the number of possible patterns) can be calculated by applying the formula

$$P = s^{w*h},$$

where P stands for permutations, s for the number of possible states (in our case it is 2: dead or alive), w for width (the number of cells in a row) and h for height (the number of rows). So, to achieve our goal we should observe 2^{25} patterns for every rule system. To prevent the gliders of getting out of the environment and keeping the size at minimum at the same time the size of the environment should be calculated by applying the following formulas:

$$W = w + 2g,$$

$$H = h + 2g,$$

where W is the width of the environment, while w is the width of the pattern and g is the number of generations. The same scheme goes for the height. The gliders can move only one cell in one direction in every generation. This is the reason we are using these formulas to calculate the ideal size of the environment. To boost the speed of the comparison, we implemented the algorithm in the following way:

– *Generation 0:* no need for comparing.

– *Generation 1:* let us say that the first cell of the pattern has the (x, y) coordinates; then the comparing starts at $(x - 1, y - 1)$.

– *Generation 2:* comparing starts at $(x - 2, y - 2)$.

For better understanding, see Figure 4.

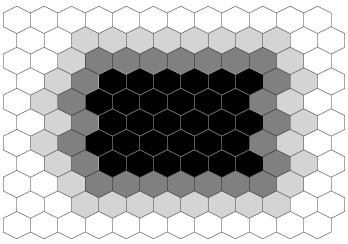

Figure 4: The black cells are representing the initial area of the pattern. The dark grey cells are for the area where we are looking for the pattern in the next generation, and the light grey is for the second generation.

In the future we would like to implement the first algorithm as well. Using the first solution, we would be able to detect bigger oscillators. However, for performance boost we are planning to implement it in C++.

4. Results of the oscilator finder

By applying our implementation in practice, we were able to detect more oscillators and gliders than before. Some of the oscillators found with the new algorithm can be seen on Figure 5 and 6, using rule systems B36/S2/D014578 and B3678/S4/D0125. Because of the limited size, this paper does not contain all detected oscillators.

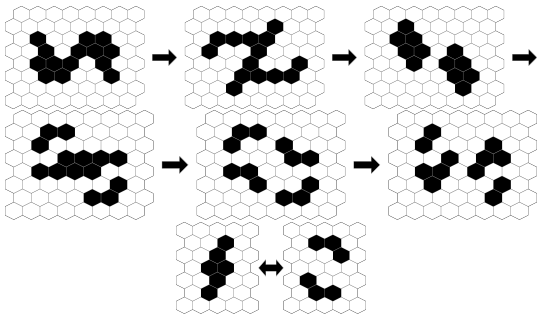

Figure 5: oscillators of the new algorithm (B36/S2/D014578)

5. Results of Game of Life on triangular grid

In this implementation we use the same rule-system format as in the hexagonal implementation. Unlike in the previous implementation of Game of Life, every cell has 12 neighbors (two triangles are considered as neighbors, when the corners or the sides are connected). There are two different types of cells. Both types of cells (the black cells) represented on Figure 7 have 12 neighbors. The relatively big number of the neighbors makes this implementation more complex. We wanted to test every possible rule-system that can be applied in this environment. However, due to computing capacities we reduced the set of rules from 3^{12} rulesystems to 3^8 rule-systems by applying a constant subrule (can be expanded): D0,1,10,11,12. The results were obtained from the analyzes of simulations, which have an environment of a 50x50 triangular grid. This means that the total number of cells is 2500.

5.1. Case 1

By applying the rule-system B2,3,4/S6,9/D0,1,5,7,8,10,11,12 we have seen what we expected: the rule-system doesn't stabilize. In most cases the number of living cells varied greatly between two direct generations. Obviously this phenomenon occurred due to the nature of the rule-system: if a dead cell has a low number of living neighbors (2, 3, 4), then it will be alive, thus if there is underpopulation in the current generation there will be a relatively big population growth and vice versa.

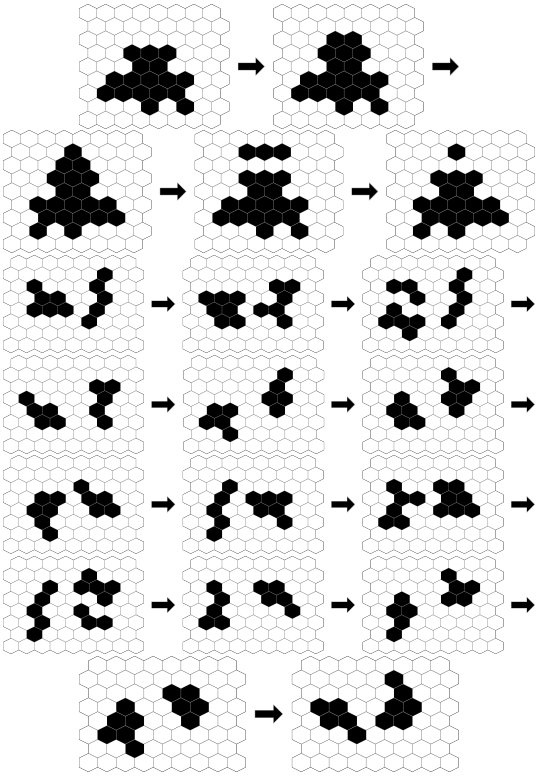

Figure 6: oscillators of the new algorithm (B3678/S4/D0125)

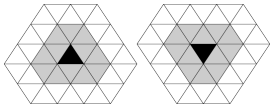

Figure 7: neighbors

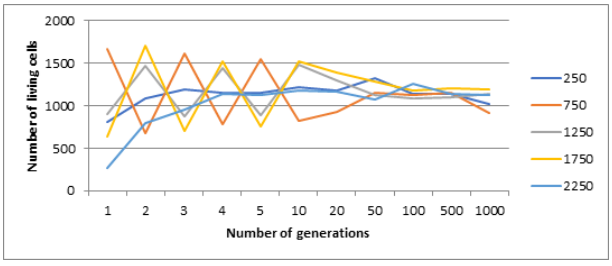

Figure 8: statistics B2,3,4/S6,9/D0,1,5,7,8,10,11,12

5.2. Case 2

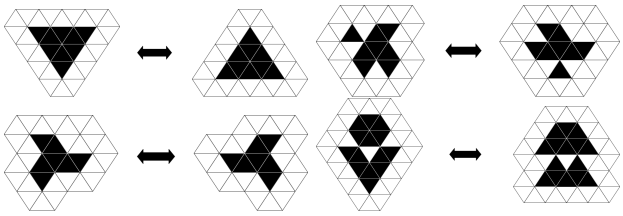

Figure 9: oscillators B5,6,7,8,9/S/D0,1,2,3,4,10,11,12

The rule-system B5,6,7,8,9/S/D0,1,2,3,4,10,11,12 does not stabilize either and there are no great differences in the number of living cells between generations after 100th generation. However, if the generation 0 has been initialized with a low AR, then the population dies out almost instantly. Only the still lives and oscillators could have been found in these simulations (Figure 9).

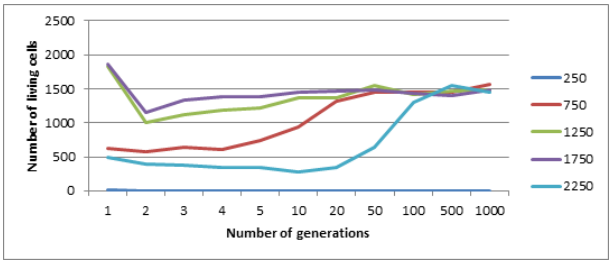

Figure 10: statistics B5,6,7,8,9/S/D0,1,2,3,4,10,11,12

5.3. Case 3

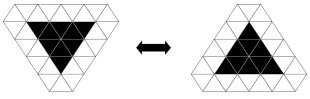

Figure 11: oscillator B4,5,6,7,9/S8/D0,1,2,3,10,11,12

The rule-system B4,5,6,7,9/S8/D0,1,2,3,10,11,12 is not different from the rest, after a few generations it populates the environment. We have detected only 1 oscillator, which has already been found earlier in the B5,6,7,8,9/S/0,1,2,3,4,10,11,12 system (Figure 11).

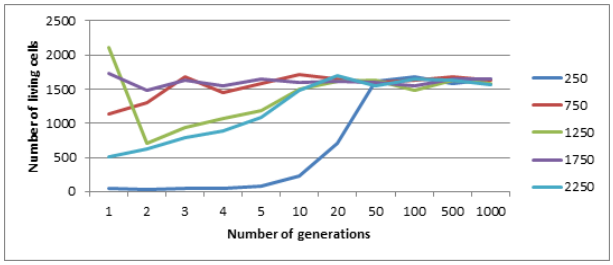

Figure 12: statistics B4,5,6,7,9/S8/D0,1,2,3,10,11,12

6. Conclusions

We have developed two algorithms for oscillator detection on hexagonal grid and implemented one of them. This detector algorithm successfully detected several oscillators and gliders. Additionally, we performed statistical analyses on Game of Life with triangular environment. We concluded that the triangular implementation is viable and functional. The fact, that oscillators are spawning and the population of cells changes through time just like in the original Game of Life proves that it supports "life". In the future we plan to run simulations on hexagonal grid with 12 neighbors as well, since it would restore the symmetry, also it could be compared to the triangular grid. Furthermore, we would like to compare the behavior of the hexagonal grid with 8 neighbors to behavior of the square grid.

References

[1] Bays, C., A Note on the Game of Life in Hexagonal and Pentagonal Tessellations, *Complex Systems* Vol. 15 (2005), 245–252.

[2] Gardner, M., The Fantastic Combinations of John Conway's New Solitaire Game 'Life', *Scientific American* Vol. 223 (1970), 120–123.

[3] Herendi, T., Nagy, B., Parallel Approach of Algorithms, *Typotex Budapest* (2014).

[4] Hidi, E., Z., Horváth, G., Hexagonal Game of Life with 8 Neighbors, *Eleventh Workshop on Non-Classical Models of Automata and Applications (NCMA 2019)*, (2019), 23–29.

[5] Preston, K., Modern Cellular Automata Theory and Applications, *Plenum Press, New York* (1984).

[6] Wolfram, S., Cellular Automata, *Los Alamos Science* Vol. 9 (1983), 2–21.