Vol. 9 (2016) *Acta Physica Polonica B Proceedings Supplement* No 1

CELLULAR AUTOMATA AGENTS FORM PATH PATTERNS EFFECTIVELY[*]

Rolf Hoffmann

Technische Universität Darmstadt, FG Rechnerarchitektur
Hochschulstr. 10, 64289 Darmstadt, Germany
hoffmann@rbg.informatik.tu-darmstadt.de

(Received February 29, 2016)

Considered is a 2D cellular automaton with moving agents. Each cell contains a particle with a value = (color, marker), which can be changed by an agent. The objective is to form a specific target pattern belonging to a predefined pattern class. Areas of applications are the alignment of spins, particles, or fibers. The target patterns shall consist of preferably long narrow paths with the same color, they are called "path patterns". The quality of a path pattern is measured by the degree of order that is computed by counting matching 3×3 patterns (templates). The markers act like artificial "pheromones" that improve the solution's quality (effectiveness) and the efficiency of the task. The agents' behavior is controlled by a finite state machine (FSM). The agents used can perform 32 actions, combinations of moving, turning and value setting. They react on the own particle's value, the value in front, and blocking situations. For a given set of $n \times n$ fields near optimal, FSMs were evolved by a genetic algorithm. The evolved agents are capable of forming path patterns with a very high degree of order (90–100%). The whole multi-agent system was described by cellular automata. The CA-w model (cellular automata with write access) was used for the implementation of the system in order to reduce the implementation effort and speed up the simulation.

DOI:10.5506/APhysPolBSupp.9.63

1. Introduction

The original idea for this research was to find artificial patterns by agents which are to a certain extent creative and impressive from the artistic point of view. As artistic patterns are difficult to evaluate, the more modest objective was to find patterns with a certain interesting or valuable structure.

[*] Presented at the Summer Solstice 2015 International Conference on Discrete Models of Complex Systems, Toronto, Ontario, Canada, June 17–19, 2015.

Experiments were conducted to find certain global patterns, like a black box in the center of the field, or with a global symmetry. It turned out, until now, that it is very difficult to find agents that can obey such strict global requirements. Then, the objective was even more relaxed, namely to form global patterns that obey intrinsic local rules (local matching patterns, templates). The 3×3 Moore-neighborhood was used for the templates. In order to form more complex patterns, the neighborhood could be enlarged.

Practical effect can be taken out of this research when nano-structures have to be constructed by nano-robots, or when focused energy is beamed onto certain cells in order to change their physical state [1–5]. Further applications can be seen, like forming mechanical, chemical, biological [6], or computational devices with specific structures.

The task. Given is a field of $N = n \times n$ cells with a border, n assumed to be even. Each cell, except the border cells, contains a particle with a certain *value* $\in \{0, 1, 2, 3\}$. The values $\{0, 1\}$ shall represent the color *white*, and the values $\{2, 3\}$ shall represent the color *black*. Two values are used to represent one color in order to make the task easier to solve. A value can be seen as a tuple of binary values $(\mathrm{c}, \mathrm{s}) \in \{(0,0), (0,1), (1,0), (1,1)\}$, where $\mathrm{c} \in \{0, 1\}$ represents the *color*, and $\mathrm{s} \in \{0, 1\}$ represents an additional information, the *marker*. Markers are data items stored in the whole environment which act as a distributed global memory. Agents can change this data, either to be used later by the same agent (as scratch-pad) or by another agent. In our task, markers enlarge significantly the control space and support communication between agents. *E.g.* an agent may change the value of a marker from 0 to 1 signaling that the current site is already locally ordered. Without using markers, the task can only be solved with a very low degree of order, which is not much better than the order which appears at random. Therefore, markers were introduced.

A given number k of agents is moving around in the field. An agent can change the color and the marker at the site where it is situated on. Initially, the particle values, the agents, and the agent's directions are randomly distributed. The agents' task is to construct a global state where a certain *target pattern* appears that belongs to a predefined pattern class. In this work, we define a *path pattern class* by preferably long paths of width $= 1$, where all the cells of the neighboring paths have the opposite color or belong to the border. The paths may have branches and may form loops (as shown in Fig. 6).

The objective is to find a specific behavior (controlled by a finite state machine (FSM)) for the agents which allows them to solve the task with a high quality. The capabilities of the agents shall be constrained, *e.g.* the number of control states, the action set, and the details of the perceived environment.

Why agents are used? What is the advantage to solve this task by agents? Generally speaking, agents can behave in a very flexible, powerful and coordinated way because of their intelligence and their specific sensors and actuators. Important properties that can be achieved by agents are:

Scalability. The problem can usually be solved with a variable number of agents, and faster with more agents in a certain range.

Tuneability. The problem can be solved faster or with a higher quality by increasing the agent's intelligence.

Flexibility. Similar problems can be solved by the same agents, *e.g.* by changing the shape or size of the environment.

Fault-tolerance. When obstacles are introduced or not all agents work correctly, the problem can still be solved in a gracefully degraded way.

Updating-tolerance. Often the time-evoluted global state depends only weakly on the updating-scheme (synchronous, asynchronous). This is important if no global clock is available.

Because agents have such valuable properties, they can be employed to design, model, analyze, simulate, and solve problems in the areas of complex systems, real and artificial worlds, games, distributed algorithms or mathematical issues.

Related work.

(i) *Pattern formation:* Agent-based pattern formations in nature and physics are studied in [7,8]. A programming language is presented in [9] for pattern-formation of locally-interacting, identically-programmed agents; as example, the layout of a CMOS inverter is formed by agents. In [10], a general framework is proposed to discover rules that produce special spatial patterns based on a combination of Machine Learning strategies including Genetic Algorithms and Artificial Neural Networks.

(ii) *Modeling moving agents/particles:* In Sect. 3, the agent system is described by classical cellular automata (CA) and then implemented by cellular automata with write access (CA-w) [11]. Other modeling concepts related to CA are lattice-gas cellular automata, block substitutions [12], or partitioned CA as used in [13].

(iii) *FSM-controlled agents:* In former investigations, we have tried to find the best algorithms for the *Creature's Exploration Problem* [14], in which the agents have the task to visit all empty cells in the shortest time, and for the *All-to-All Communication Task* [13], in which each agent has to distribute its information to all the others, and for the *Target Searching Task* [15]. The FSMs for these tasks have been evolved

using, *i.e.*, genetic algorithms, genetic programming [16], and sophisticated enumeration methods. Other related works are a multi-agent system modeled in CA for image processing [17], and modeling the agent's behavior by an FSM with a restricted number of states [18]. An important pioneering work about FSM-controlled agents is [19]. FSM-controlled robots are also well-known.

This work develops further the issues presented in [20] where four colors were asked for, and where it occurred that only patterns with less than four colors and a low degree of order could be found. Now, the goal is to generate patterns with two colors only and a very high degree of order by means of additional markers.

2. Target patterns and degree of order

How can the class of target patterns be defined? The idea is to use a set of small local matching patterns as building blocks, also called *templates*, that can successfully tile the field (with overlaps). In the color structures arranged by the agents, such templates are expected with a high frequency. The templates used here are depicted in Fig. 1 (a). They define the target *path patterns*. *E.g.* the *plank* template means that there are 3 consecutive cells of the same color (depicted in black), enclosed by $3+3$ cells in another color (depicted in gray). The plank can be rotated by 90°, giving the second form of this type. Altogether, under reflection and rotation, there are 41 distinct templates.

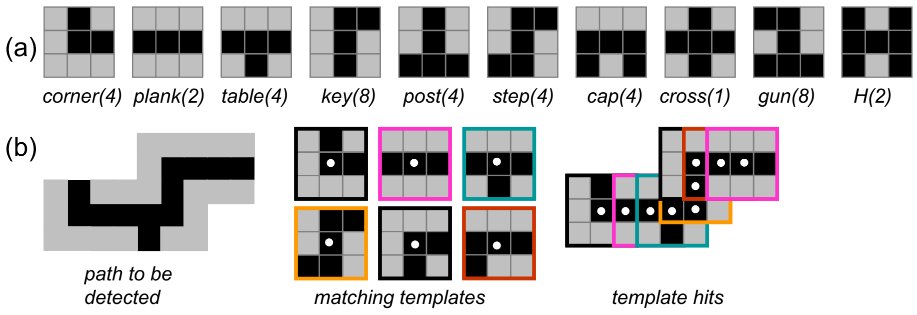

Fig. 1. (a) Templates are small building blocks which are expected to appear in a target pattern. A path cell in the template is shown in "black" (representing either black or white). "Grey" represents another color (either white or black, or border). The number of symmetric templates by rotation and reflection is given in the brackets. Note that the templates describe equally black and white paths. (b) An example for a path (part of the target pattern) that can be tiled (with overlaps) by matching templates. Each matching template produces a hit (dot). All hits are summed up and give the degree of order.

The patterns created by the agents have to be evaluated, how well they fit into the defined path pattern class. In order to evaluate a given pattern, all templates are applied (tested) on each cell. If a match (hit) is found, a dot is used to mark this cell. Then, all dots are summed up which gives the total number of hits h. This number is also called *degree of order*. For the path shown in Fig. 1 (b), there are 6 templates that match, leading to 9 hits. If several templates match at the same site, the number of hits remains one.

By this rating method, the terminal cells of a path-tree are not counted. *E.g.* a linear path of k black or k white cells without branches has two terminal cells which are not counted, therefore the length count yields $k-2$. Not counting the terminal cells of each path can be seen as a penalty reflected in the fitness function which is used during the optimization, thereby the searching for loops (loops without dendrites have no terminals) and long paths are favored. Note that terminal cells are no border cells, they are the endings or leafs of the tree-like structures. Loops are favored because they have no leafs and they can be seen, for example, as ideal stiff physical structures. The theoretically maximum order is $h_{\max} = n \times n - 3$ (for even n). The relative order is $h/h_{\max}$. The maximum can be reached by special patterns *e.g.* for $n = 4, 8$ as shown in Fig. 2 (a), (b). Patterns are called "balanced" if the number of black and white cells are equal. In order to simplify the optimization task (Sects. 4, 5), unbalanced patterns were allowed for #black/#white in the range of $1/2$ to 2.

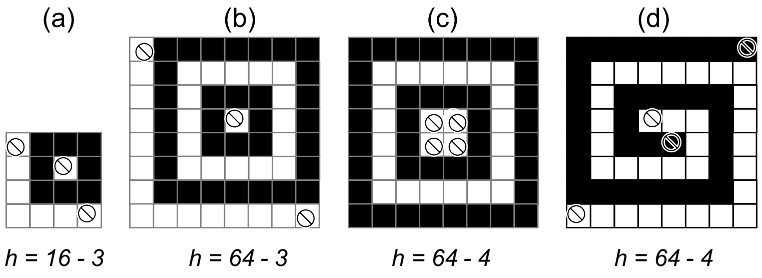

Fig. 2. Optimal balanced target patterns for (a) field size 4×4, (b) field size 8×8. Near optimal balanced target patterns for size 8×8 ((c), (d)). Terminal cells are marked by a crossed circle.

It can be noticed that the testing for template hits can be performed synchronously in a CA-like fashion. From the formal language point of view, the templates can be seen as the generators of a 2D pattern language. After having defined and used theses path patterns, the author noticed that related patterns have been known as "Square Kufic" in Arabic art for long.

3. Describing the system by cellular automata

The cell's rule has to react on non-uniform situations, *e.g.* whether there is an agent situated on a cell or not. Therefore, the cell's state is modeled by a record $(Type, (Color, Marker) = Value, Agent)$ comprising a type tag, where

$$Type \in \{Border, Particle, ParticleAndAgent\}, \quad \text{and}$$
$$Agent = (Identifier, Direction, ControlState).$$

In fluid dynamics, there are two representations of agents (fluid particles), Lagrangian and Eulerian. In the Eulerian representation, agents are observed at each grid location. In the Lagrangian representation, the position of each agent is updated and followed by the observer. We used the Eulerian representation, which makes the CA description and implementation simpler although more computer space is needed for systems with a low density of agents. In addition, some computer time overhead arises when the active agents have to be identified. The Eulerian representation allows also to create and delete agents in a simple way, although for our problem this feature was not needed. The implementation of the Lagrangian representation requires to maintain a separate list of agents that is connected to the grid. If the agents density is low and the agents carry a lot of individual information, then the Lagrangian representation may be a better choice.

The capabilities of the agents have to be defined before designing or searching for the agents' behavior. The main capabilities are: the perceivable inputs from the environment, the actions an agent can perform, and the size of its memory (number of possible control states, optionally additional data states). In our system, an agent shall react on the following inputs in a certain combination:

— the **own value** $V = (color, marker)$ of the cell the agent is situated on,
— the **value in front** V_{F} (in moving/viewing direction),
— a **border cell** in front,
— the **blocked** situation/condition, caused either by a border, another agent in front, or when another prior agent can move to the front cell in the case of a conflict. The inverse condition is called *free*.

An agent has a moving/viewing direction $D \in \{0, 1, 2, 3\} \equiv$ {*toN, toE, toS, toW*}. Note that in the used model an agent cannot observe the direction and control state of another agent in the neighborhood, only its presence. The actions that an agent shall be able to perform are:

— **move**: $move \in \{0, 1\} \equiv \{wait, go\}$,
— **turn**: $turn \in \{0, 1, 2, 3\}$. The new direction is $D(t+1) \leftarrow (D(t) + turn)\ mod\ 4$,
— **setvalue**: The new value of (color, marker) is $V(t+1) \leftarrow setvalue \in \{0, 1, 2, 3\} \equiv \{(0,0), (0,1), (1,0), (1,1)\}$.

The move, turn and set value actions can be performed simultaneously (32 combinations). There is only one constraint: when the agent's action is *go* and the situation is *blocked*, then the agent cannot move and has to wait, but still it can turn and change the cell's particle value. In the case of a moving conflict, the agent with the lowest identifier (ID $= 0 \ldots k-1$) gets priority. Instead of using the identifier for prioritization, it would be possible to use other schemes, *e.g.* random priority, or a cyclic priority with a fixed or space-dependent base.

How can an agent move from A to B in cellular automata (CA)? Two rules have to be performed, a *delete-rule* that deletes the agent on A, and a *copy-rule* that copies the agent to B. In CA, both rules have to compute the same blocking condition, this means a redundant computation. In order to avoid this redundancy, a two-phase updating scheme can be used (first compute the moving condition, second use it in cell A and B), or using the C*ellular* A*utomata with* w*rite-access model* (CA-w) [11]. When using the CA-w model, the moving condition needs to be computed only once, either by cell A or B. If A is the actor, it computes the condition and if it is true, it copies the agent to B (like "beaming") and deletes it. If B is the actor, it computes the condition and if it is true, it copies the agent from A to B and deletes it on A. The simulation program was implemented by the CA-w model, although it is possible to implement the system in standard CA with redundant computation.

The CA-w model was introduced in order to describe moving agents, moving particles or dynamically changing activities. This model allows to write information onto a neighbor. This feature has the advantage that a neighbor can directly be activated or deactivated, or data can be sent to it by an agent.

The behavior of an agent shall be determined by an embedded finite state control automaton (FSM) (Fig. 3). The FSM corresponds to the "brain" or algorithm that controls the agent. Each CA cell is supplied with an FSM which is active when an agent is situated on it. An FSM contains a *state table* (also called *next state/output table*). Outputs are the actions (move, turn, setvalue) and the next control state. Inputs are the control state s and the relevant input situations x. The *input mapping* reduces all possible input combinations of (border, blocked, value, front value) to an index $x \in X = \{0, 1, \ldots, |X| - 1\}$ that is used in combination with the control state to select the actual line of the state table.

The following input mapping is used. If the situation is *free*, the index x is the particle's value in front: $x = V_\mathrm{F}$. If the situation is *blocked* by a border cell in front, then $x = 4$. If the blocking is caused by another agent in front, or by a prior agent in the case of a conflict, then the own value is directly mapped to the index with an offset $x = V + 5$. It is possible to choose other

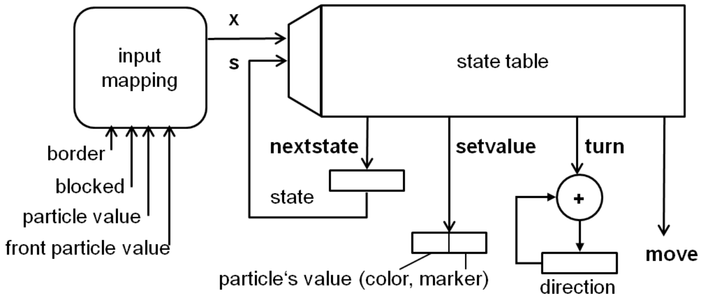

Fig. 3. Finite state machine (FSM). The state table defines the next control state, the setting of the particle's value (color, marker), the agent's new direction, and whether to move or not.

input mappings, with fewer or more x codes, or other assignments, *e.g.* in the case of blocking, the direction of the agent could be used ($x = D + 5$), instead of the own particle's value. Note that agent's view is very limited, it can react either on the value in front or its own value. Therefore, the agent's task is really difficult.

The used updating scheme is *synchronous*; exemplarily simulation experiments showed that the results with asynchronous updating are quite similar.

4. Evolving the agent's behavior by a genetic algorithm

The ultimate goal is to find an FSM (algorithm controlling the behavior) which is optimal on average for all possible initial configurations. As we cannot optimize for all possible initial configurations within a limited amount of computation time, we used: *(i)* a fixed field size of $N = n \times n$ and optimized separately for $n = 4, 8, 16$, *(ii)* a fixed density of agents ($\delta = N/k = 25\%$), and *(iii)* 100 initial configurations (for training and evaluation). This means that we searched for specialists and not for all-rounders.

As the search space for different FSMs is very large, we are not able to check all possible behaviors by enumeration. The number of FSMs which can be coded by a state table is $Z = (|s||y|)^{(|s||x|)}$, where $|s|$ is the number of control states, $|x|$ is the number of inputs and $|y|$ is the number of outputs. As the search space increases exponentially, we use a genetic algorithm in order to find the best FSM with reasonable computational cost. Even with a genetic approach, the number of states, inputs and outputs have to be kept low in order to find a good solution in acceptable time. A possible solution (genome of an individual in the genetic) corresponds to the contents of the FSM's state table. For each input combination $(x, state) = j$, a set of actions is assigned: $actions(j) = (nextstate(j), setvalue(j), move(j), turn(j))$, *e.g.* actions((0, 0)) = (4, 2, 1, 1) given by the table in Fig. 4 (a).

```
          /x=0 \ /x=1 \ /x=2 \ /x=3 \ /x=4 \ /x=5 \ /x=6 \ /x=7 \ /x=8 \
state     012345 012345 012345 012345 012345 012345 012345 012345 012345
(a)
nextstate 411245 335143 434520 241251 140202 331211 020325 435332 105104
setvalue  221202 113010 220102 031022 233321 020310 131112 202031 321011
move      100001 000111 101011 110100 100100 011011 010100 000000 111100
turn      110330 320103 101200 303200 103121 330023 302231 201000 212032
(b)
nextstate 245030 205135 043043 033000 000253 233243 223215 115225 025231
setvalue  012012 333310 120101 233313 312312 100010 323113 331200 210002
move      010001 001000 100100 111010 110001 001101 011011 101110 001101
turn      010303 310321 312103 203113 220030 113033 313133 213211 011013
(c)
nextstate 245040 205145 043103 033000 000214 244243 223210 120321 030241
setvalue  010312 233310 120102 233323 313312 102111 333113 302312 210113
move      000011 001000 100100 111010 101010 011110 011011 001100 001101
turn      110300 310331 312100 203123 220000 110003 323130 213223 021023
```

Fig. 4. (a) Top FSM evolved for 4×4 fields. (b) Top FSM evolved for 8×8 fields. (c) Top FSM evolved for 16×16 fields.

A relatively simple genetic algorithm was used. We experimented also with the classical crossover/mutation method. Then, we found that mutation only gave us similar good results. Therefore, we are using here only mutation. It is subject to further research which heuristic is best to evolve state machines.

The population of M individuals is stored in two lists (A, B) with $M/2$ individuals each. During each iteration, the following steps are performed:

(Step 1) $[A' \leftarrow mutate(A)]$ — $M/2$ offspring are produced from list A by mutation.

(Step 2) $[(A, B) \leftarrow deleteDuplicates(sort(A', A, B))]$ — The union of the $M/2$ offspring and the old M individuals are sorted according to their fitness (high fitness — low time first), duplicates are deleted and the number of individuals is then reduced to the limit of M in the pool.

(Step 3) $[(A, B) \leftarrow exchange(b, A, B)]$ — In order not to get stuck in local minima and to allow a certain diversity in the gene pool, the first b individuals from B are exchanged with the last b individuals from A.

An offspring is produced by modifying separately with a certain probability p each $action \in \{nextstate, setvalue, move, turn\}$: $action \leftarrow (action+1)$ $mod\ N_{\text{action}}$. We restricted the number of actions to $N_{\text{states}} = 6$, $N_{\text{setvalue}} = 4$, $N_{\text{move}} = 2$, and $N_{\text{turn}} = 4$.

The fitness of our multi-agent system is defined as the number t of time steps which is necessary to emerge successfully a target pattern with a given degree h_{target} of order, averaged over all given randomized initial configurations (color and marker distribution, position and directions of the agents). As the behavior of the whole system depends on the behavior of the agents, we search for an agents' FSM that can solve the problem successfully with a minimum number of steps for a large number of initial configurations. Successfully means that a target pattern with $h \geq h_{\text{target}}$ was found. The fitness function F is evaluated by simulating the agent system with a tentative FSM on a given initial configuration.

F(FSM, initial config.) $=$ TimeSteps *if successful* within TimeLimit,
F(FSM, initial config.) $=$ HighConstant *otherwise.*

Then the mean fitness $\overline{F}$(FSM) is computed by averaging over all given initial configurations. The mean fitness $\overline{F}$ is then used to rank and sort the FSMs. The parameters used were TimeLimit $= 5,000$ and HighConstant $= 100,000$. The genetic procedure starts with $M = 20$ random FSMs. Usually, there is no FSM in the initial population that is successful. After some generations, some successful FSMs are found. Then, after further generations, FSMs are expected to be evolved that are completely successful on all or most of the given set of initial configurations (the *training set*). It turned out that it is very difficult and time consuming to find good solutions straightforward. Therefore, the genetic procedure was divided into several phases with increasing difficulty by increasing stepwise the degree of order h_{target} and the cardinality of the training set. In addition, several runs were performed in parallel with different initial random seeds. The total computation time on a processor Intel Xeon QuadCore 2 GHz was around two weeks to find optimal FSMs for the cases described in the next section.

5. Evolved finite state machines

First case. This case was investigated in order to find the most efficient optimization parameters. The field size was 4×4, and 4 agents were used. The number of individuals in the whole list (A, B) was set to $M = 20$. The number of b individuals from the second list B (with smaller fitness) to be exchanged with list A was varied, the most efficient value was $b = 3$. The mutation probability p was varied from $1/20$ to $1/60$, $p = 1/40$ turned out to be the most efficient (good solutions were found fastest).

The best evolved FSM (Fig. 4 (a)) was able to order totally (100%, $h_{\max} = 13$), Fig. 1 (a) only 75 random fields out of 100, and taken into account only them, the mean number of time steps was $t_{100\%} = 467$ (min. 14, max. 1960). This result shows that it is very difficult, even for small fields and using markers, to find an FSM that is able to order totally all possible initial configurations.

Second case. The field size was 8×8, and 16 agents were used. The chosen density of the agents was again $\delta = 1/4$, because as a result taken from the former research [20], this density turned out to be most cost-effective in terms of *time* $\times$ *agents*. The degree of order to be reached was incremented from lower levels until $h_{\text{target}} = 58$, the theoretical maximum is $h_{\max} = 61$. The relative degree of order is $h_{\text{rel}} = h/h_{\max}$, *e.g.* $58/61 = 95\%$. FSMs were found which are successful on all 100 configurations of the training set. The total number of optimization generations was 8,000 (each with 10 offspring FSMs), and the TimeLimit was 3,000 simulation steps. The best found FSM for $h_{\text{rel}} \geq 58$ (95%) is given in (Fig. 4 (b)), and the mean number of time-steps was $t_{95\%} = 738$, averaged over 100 random initial configurations. For comparison with the following third case, the requested degree of order was lowered to $h_{\text{rel}} = 90\%$ which yielded a mean time of only $t_{90\%} = 412$.

Third case. The field size was 16×16, and 64 agents ($\delta = 1/4$) were used. The degree of order to be reached was incremented from lower levels to $h_{\text{target}} = 228$ ($h_{\text{rel}} = 90\%$), the theoretical maximum is $h_{\max} = 253$. Compared to the second case, h_{rel} is 5% lower, because it was not possible to find FSMs which were successful on all 100 configurations of the training set. The total number of optimization generations was 8,000 (each with 10 new FSMs tested), and the TimeLimit was 5,000. The best found FSM for $h_{\text{rel}} \geq 90\%$ is given in (Fig. 4 (c)). The mean number of time-steps was $t_{90\%} = 599$, averaged over 100 random initial configurations. Depending on the initial configuration, $t_{90\%}$ varied between 199 and 1846. Comparing to the second case (field size ratio is 4:1), the time ratio is $t_{\text{ratio}} = t_{90\%,16\times16}/t_{90\%,4\times4} = 1.45$.

The cost per cell for ordering is $\text{cpc} = \text{cost}/N = t \times k/N = t \times \delta$. The ratio $\text{cpc}_{90\%,16\times16}/\text{cpc}_{90\%,4\times4}$ is equal to the above time ratio t_{ratio}.

The whole multi-agent system using the evolved top FSMs was simulated (Figs. 5 and 6). The snapshots show how the path patterns are being built. It can be seen that at the end black and white long paths trees appear, or closed loops. At the beginning agents move on any cell, whereas at the end they mainly move on the black paths. The markers (depicted as gray (green) or black (red) small squares) are random at the beginning, and almost all of them are changed to black (red) during the run by the agents. Black (red) markers signal to the agent that the system is already partially ordered, and gray (green) markers signal that the degree of order can be improved. Thus, the agent's FSM can react not only on the local color but also on

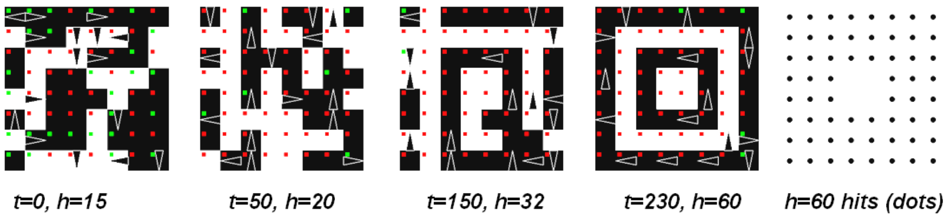

Fig. 5. Snapshots showing how path patterns are formed in a 8×8 field by 16 agents. h = degree of order. Agents are represented by triangles, and the markers are represented by small squares (black (red) = 1, gray (green) = 0). Dots represent template hits.

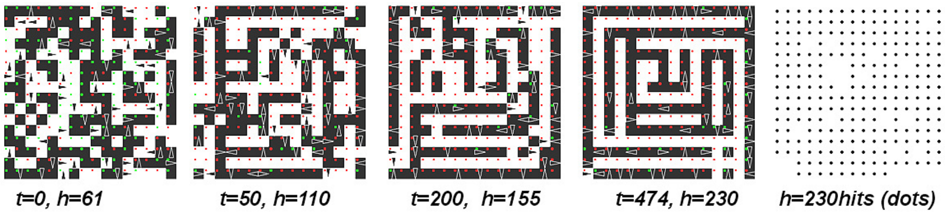

Fig. 6. Snapshots showing how path patterns are formed in a 16 × 16 field by 64 agents. h = degree of order.

the marker's value in order to be more effective and efficient. Note that the reached path patterns are not stable. After having reached the required degree of order, the pattern is changing and the degree of order is fluctuating (*e.g.* for one selected 16 × 16 field: $h_{\text{rel}} = 90\%(\approx +8 \ldots -15\%)$, see Fig. 7).

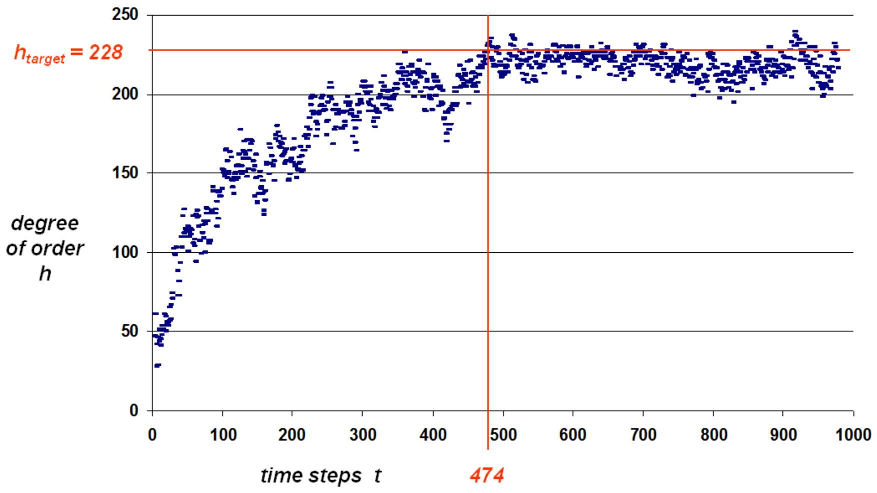

Fig. 7. Degree of order *vs.* time. Initial configuration: 16 × 16 field with random colors, random markers, and with 64 random placed agents. The degree of order to be reached was $h_{\text{target}} = 229 (h_{\text{rel}} = 90\%)$. The best evolved FSM was used.

6. Conclusion

The objective was to find FSM-controlled agents that can form specific path patterns. The class of path patterns was defined by a set of templates, small 3 × 3 local patterns. The number of templates that can be found in a given pattern defines the degree of order. For $n \times n$ fields ($n = 4, 8, 16$) and a density of 25% of agents, near optimal FSMs were evolved by a genetic algorithm. The agents are able to form successfully the aimed path patterns with a 100–90% degree of order by means of markers. The general result is that the generation of specific patterns by CA agents can be designed in a methodical way. Using larger templates, the same method would allow to generate more sophisticated patterns.

REFERENCES

[1] D. Shi *et al.*, *J. Appl. Phys.* **97**, 064312 (2005).

[2] M. Itoh, M. Takahira, T. Yatagai, *Opt. Rev.* **5**, 55 (1998).

[3] Y. Jiang, T. Narushima, H. Okamoto, *Nature Phys.* **6**, 1005 (2010).

[4] G. Roberts Jr, "X-ray Laser Explores How to Write Data with Light", National Accelerator Laboratory, March 19, 2013, `https://www6.slac.stanford.edu/news`

[5] D. Press, T.D. Ladd, B. Zhang, Y. Yamamoto, *Nature* **456**, 218 (2008).

[6] A. Deutsch, S. Dormann, *Cellular Automaton Modeling of Biological Pattern Formation*, Birkäuser, 2005.

[7] E. Bonabeau, "From Classical Models of Morphogenesis to Agent-Based Models of Pattern Formation", Santa Fe Institute Working Paper: 1997-07-063.

[8] H. Hamann, "Pattern Formation as a Transient Phenomenon in the Nonlinear Dynamics of a Multi-Agent System" in: Proc. of MATHMOD 2009.

[9] R. Nagpal, "Programmable Pattern-Formation and Scale-Independence", MIT Artificial Intelligence Lab, 2002.

[10] S. Bandini, L. Vanneschi, A. Wuensche, A.B. Shehata, *Fundam. Inform.* **87**, 207 (2008).

[11] R. Hoffmann, *Acta Phys. Pol. B Proc. Suppl.* **5**, 53 (2012).

[12] S. Achasova, O. Bandman, V. Markova, S. Piskunov, *Parallel Substitution Algorithm — Theory and Application*, World Scientific, 1994.

[13] R. Hoffmann, D. Désérable, *Lect. Notes Comput. Sci.* **7979**, 316 (2013).

[14] M. Halbach, R. Hoffmann, L. Both, *Lect. Notes Comput. Sci.* **4173**, 571 (2006).

[15] P. Ediger, R. Hoffmann, *Electronic Notes Theor. Comput. Sci.* **252**, 41 (2009).

[16] M. Komann, P. Ediger, D. Fey, R. Hoffmann, *Lect. Notes Comput. Sci.* **5481**, 280 (2009).

[17] M. Komann, A. Mainka, D. Fey, *Lect. Notes Comput. Sci.* **4671**, 432 (2007).

[18] B. Mesot, E. Sanchez, C.-A.Peña, A. Perez-Uribe, "SOS++: Finding Smart Behaviors Using Learning and Evolution. Artificial Life VIII", MIT Press, Cambridge, Mass., 2002, pp. 264–273.

[19] M. Blum, W. Sakoda, "On the Capability of Finite Automata in 2- and 3-dimensional Space, 18th IEEE Symp. on Foundations of Computer Science, 1977, pp. 147–161.

[20] R. Hoffmann, *Lect. Notes Comput. Sci.* **8751**, 660 (2014).