Cellular Automaton as Sorting Network Generator Using Instruction-Based Development

Michal Bidlo and Zdenek Vasicek

Brno University of Technology, Faculty of Information Technology
IT4Innovations Centre of Excellence
Božetěchova 2, 61266 Brno, Czech republic
{bidlom,vasicek}@fit.vutbr.cz

Abstract. A new cellular automaton-based approach allowing to generate sorting networks is presented. Since the traditional table-based transition function in this case involves excessive number of rules, a program-based representation of the transition function is applied. The sorting networks are encoded by the cell states and generated during the cellular automaton development. The obtained results are compared with our previous approaches utilizing cellular automata.

Keywords: cellular automaton, sorting network, instruction-based development, evolutionary design

1 Introduction

In nature, development is a biological process of ontogeny representing the formation of a multicellular organism from a zygote. It is influenced by the genetic information of the organism and the environment in which the development is carried out. In the area of computer science and evolutionary algorithms in particular, the artificial (or computational) development has been inspired by this biological phenomena. Computational development is usually considered as a non-trivial and indirect mapping from genotypes to phenotypes in an evolutionary algorithm to provide a more flexibility in the construction process of a candidate solution than that is achievable by direct mappings. In such case the genotype has to contain a prescription for the construction of target object. While the genetic operators work with the genotypes, the fitness calculation (evaluation of the candidate solutions) is applied on phenotypes created by means of the development. The principles of the computational development together with a brief biological background and selected application of this bio-inspired approach are summarized in [13]. There are several approaches to perform the development, for example Miller's developmental cartesian genetic programming [11], Koza's developmental genetic programming [8], instruction-based development [5] or cellular automata [12].

This paper proposes a method for the evolutionary design of cellular automata allowing to generate functional structures encoded by the states produced during the cellular automaton development. In order to overcome the problem of

scale when evolving CAs with higher number of cell states, the instruction-based approach to represent the local transition function of the CA will be applied. The concept of instruction-based CA was introduced in [3] and demonstrated on non-trivial problems such as replication and pattern development problem. An encoding will be introduced that allows to represent parts of a sorting network by means of cell states in a given development step of the CA. The hypothesis we will work on in this paper is that if a suitable development algorithm (local transition function) is found, then various patterns of cell states may be generated in a series of CA steps that encode a working sorting network. The obtained results will be compared to the sorting networks developed using our previous methods [4].

2 Related Work

In this paper, one-dimensional uniform cellular automata (CA) will be applied whose concept was introduced in [12]. To calculate the next cell states, the cellular neighborhood will involve the investigated cell and its left and right immediate neighbors. Since a finite CA dimension will be considered, zero boundary conditions will be considered to determine the states of boundary cells.

Cellular automata have been applied to solve many complex problems in different areas. For example, Miller investigated the problem of evolving a developmental program inside a cell to create multicellular organism of an arbitrary size and characteristic [10]. Tufte and Haddow utilized a FPGA-based platform of Sblocks [7] for the online evolution of digital circuits. The evolutionary algorithm was utilized to design the rules for the development of the CA [14].

Cellular automata have also been successfully applied as a developmental method for generating digital circuits. Although both the uniform and non-uniform CA demonstrated the ability to generate combinational circuits, the non-uniform approach requires several times higher chromosome length which in many cases exceeds the amount of information needed to encode the candidate solution directly (e.g. using CGP) [1, 2]. This issue is caused by the need of encoding different local transition function for each cell of the CA. Therefore, our next research has mainly been devoted to uniform CAs and advanced techniques of encoding of the local transition function allowing to reduce the search space for the increasing number of cell states [3]. A method for generating sorting networks by means of cellular automata was introduced in [4].

3 Sorting Networks and Their Design

A sorting network [9] is defined as a sequence of compare–swap operations (comparators) that depends only on the number of elements to be sorted, not on the values of the elements. A compare–swap of two elements (a, b) compares and exchanges a and b so that we obtain $a \leq b$ after the operation (a comparator possesses 2 inputs and 2 outputs). Sorting networks represent a class of digital circuits consisting of a finite sequence of comparators. Therefore, a pair (a, b),

$a < b$ represents a comparator whose first input is connected to wire of index a and the second input to wire of index b of a sorting network. Figure 1 shows an example of a 3-input sorting network. For the purposes of this paper we will define the "width" of a comparator as the difference of the indices of wires the comparator is connected to. As evident, all comparators of the width 0 or the value exceeding the number of wires (inputs) of the sorting network are meaningless according to the previous specification.

The number of compare–swap components and the circuit delay are two crucial parameters of any sorting network. By delay we mean the minimal number of groups of compare–swap components that can be executed sequentially. Designers try to minimize the number of comparators, delay or both parameters.

4 Sorting Network Development from Cell States

In [4], two different techniques were introduced for the development of sorting networks by means of cellular automata: (1) an absolute encoding and (2) a relative encoding. Each method is based on a suitable enhancement of the local transition function of the CA. The fundamental principle of this enhancement is based on including an additional information to the local transition function (together with the new cell state) that represents a prescription for generating a compare–swap component. In summary, this information includes connection of a compare–swap element to the specific wires of the target sorting network. During the process of the CA development, each cell determines its next state according to a specific rule of the local transition function. The additional information associated with this rule specifies a comparator to be generated by a cell in a development step. Whilst the absolute encoding directly specifies connection of a comparator generated by a given cell after calculating its next state, the relative encoding involves position of the cell and the additional information in the transition function is used to calculate the comparator connection with respect to the cell position. The initial state of the CA together with the enhanced local transition function is a subject of the evolutionary design process. Those experiments have shown the possibility of involving the cellular automata development to generate working sorting networks.

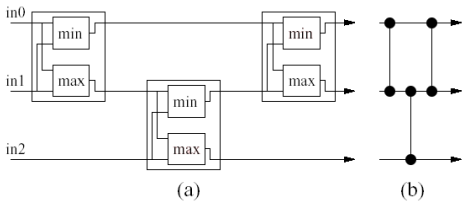

Fig. 1. (a) A three-input sorting network consists of three comparators connected in an appropriate way to three wires. (b) Alternative symbol. This network can be described using the sequence of integer pairs (0,1)(1,2)(0,1).

This section introduces a new approach to developing sorting networks by means of cellular automata that encodes the compare–swap operations by the cell states rather than the additional information of the local transition function. However, if the number of inputs of the sorting network increases, the number of cell states increases too in order to be able to encode the connection of the comparators to all the wires of the sorting network (which is needed to generate a working solution). For example, the development of an 8-input sorting network requires 8 cell states. If 3-cell neighborhood is considered, there are $8^3 = 512$ rules of the transition function in total and in this case the search space (the space of all the local transition functions) contains $8^{512} = 2.41 \times 10^{462}$ candidate solutions and the exploration of so huge search space using an evolutionary algorithm becomes extremely difficult. In order to overcome this issue, the concept of instruction-based development was adopted to cellular automata [3]. The key idea of this approach is to evolve a program (a sequence of application-specific instructions) for calculating the transition function rather than the complete sequence of transition rules. Since the length of the chromosome encoding the program is shorter, the size of the search space can be reduced substantially as demonstrated in [3].

For the purposes of this paper the following rules will be considered for generating comparators from the cell states. Let p denote the position (index) of a cell in the CA. Then p corresponds to the index of a wire of target sorting network the first input of a comparator will be connected to. Let s denote the state of a cell. Then s corresponds to the width of the comparator, thus its second input will be connected to the wire of index $p+s$. A comparator $(p, p+s)$ is generated by the cell in a development step of the CA if (1) s is different from state 0, (2) $p+s$ does not exceed the index of the last wire of the target sorting network and (3) neither wire p nor wire $p+s$ is occupied by other comparator generated in the same development step (The comparators generated in a development step are independent each other and thus can be executed in parallel. It means that the delay of the target sorting network may be reduced). The order of comparators generated in a given development step is determined by the increasing cell position which ensures the development process is deterministic.

For example, consider a 4-cell cellular automaton whose cells can possess one of the states 0, 1, 2, 3. Assume that the CA performs three development steps and the goal is to generate a 4-input sorting network as shown in Figure

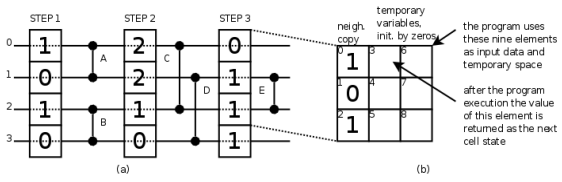

Fig. 2. An example of generating 4-input sorting network by means of a cellular automaton.

2a. After the first development step, the state of the CA is 1010. Therefore, comparator A (0, 1) will be generated by the cell at position 0 because this cell possesses non-zero state. According to the position of this cell, the first input of the comparator will be connected to wire 0 and the cell state corresponds to the width of the comparator (1), i.e. its second input will be connected to wire of index $0 + 1 = 1$. At the same time, comparator B is generated by the cell at position 2. Similarly, the cell position 2 corresponds to the wire of connecting the first comparator input and the cell state 1 determines width of the comarator, i.e. the second input will be connected to wire 3. No other comparator is generated in this development step because all the remaining cells possess state 0. After the second development step the CA exhibits state 2210. Comparator C (0, 2) is generated by the cell at position 0 – it is a comparator of width 2 because it is generated by the cell possessing state 2. Similarly, comparator D (1, 3) is generated by the cell at position 1. In the second development step of the CA, the cell at position 2 does not generate any comparator because the appropriate wires have already been occupied by the comparators generated from cells 0 and 1. The last development step takes the CA into state 0111. In this step only the cell at position 1 generates the comparator E (1, 2). The cell at position 2 does not generate any comparator because the wire 2 has already been occupied and the cell at position 3 also can not generate its comparator since it would exceed the last wire of the sorting network. As evident, the 4-input sorting network can be fully generated in 3 development steps. If more than one comparator is generated in a single step, such comparators can be executed in parallel which reduces the delay of the resulting network. It is a case of step 1 in which comparators $(0, 1)$ and $(2, 3)$ were generated and step 2 that produced comparators $(0, 2)$ and $(1, 3)$.

The cell states are calculated by the program that represents the local transition function and is a subject of evolution. In order to determine the next state of a cell, the states of cells in the cellular neighborhood are copied into the first three elements of a temporary program memory (see Fig. 2b). Then the program is executed and its instructions modify the program memory. Then the value of the memory cell 3 is returned as the next state.

5 Evolutionary System Setup

The simple genetic algorithm (GA) will be utilized for the evolutionary design of the cellular automaton that generates a target sorting network. The CA is represented by a local transition function in the form of a program consisting of a sequence of instructions [3]. Several sets of experiments will be presented regarding the development of this kind of circuits using different setup of the CA (the initial state, the number of instructions of the program and the number of development steps of the CA).

The selection of proper instructions for a given problem represent a challenging task. Experiments were performed with a wide range of arithmetic, logic and conditional instructions. However, one of the most promising instruction set for generating sorting networks showed to be a modulo addition operation with

Table 1. The set of instructions utilized for the development. $N[a1], N[a2]$ denote the cell states from the neighborhood positions determined by the instruction arguments $a1, a2$. $S[a1], S[a2], S[a3]$ represent state values specified directly by the arguments $a1, a2$ and $a3$. $|S|$ represents the number of cell states. W, C and E specifies the cell state in the West, Central and East position in the cellular neighborhood respectively. *mod* represents the modulo division.

Instruction	**Operation**
IFW $a1$ $a2$ $a3$	$if\ (N[a1] == W)\ N[a1] = N[a2]\ else\ N[a1] = N[a3]$
IFC $a1$ $a2$ $a3$	$if\ (N[a1] == C)\ N[a1] = N[a2]\ else\ N[a1] = N[a3]$
IFE $a1$ $a2$ $a3$	$if\ (N[a1] == E)\ N[a1] = N[a2]\ else\ N[a1] = N[a3]$
IFSW $a1$ $a2$ $a3$	$if\ (S[a1] == W)\ W = S[a2]\ else\ W = S[a3]$
IFSC $a1$ $a2$ $a3$	$if\ (S[a1] == C)\ C = S[a2]\ else\ C = S[a3]$
IFSE $a1$ $a2$ $a3$	$if\ (S[a1] == E)\ E = S[a2]\ else\ E = S[a3]$
IFG $a1$ $a2$ $a3$	$if\ (N[a1] == N[a2])N[a1] = N[a3]$
IFGS $a1$ $a2$ $a3$	$if\ (N[a1] == S[a2])N[a1] = S[a3]$
IFNG $a1$ $a2$ $a3$	$if\ (N[a1] != N[a2])N[a1] = N[a3]$
IFNGS $a1$ $a2$ $a3$	$if\ (N[a1] != S[a2])N[a1] = S[a3]$
ADDM $a1$ $a2$ $a3$	$N[a1] = (N[a2] + N[a3])\ mod\ \|S\|$
NOP	empty operation

some conditional instructions for modifying the cell states. Each instruction is encoded as a 4-tuple $[op, a1, a2, a3]$, where op denotes the operation code of the instruction and $a1, a2$ and $a3$ represent its arguments. The description of the complete instruction set considered for the experiments is shown in Table 1.

In a single evolutionary experiment, a chromosome of the genetic algorithm contains a fixed number of instructions that undergo changes during evolution in order to create a suitable program for calculating the local transition function. A gene of the chromosome is considered as a single element of the instruction. In all the experiments, the population consists of 100 chromosomes which are initialized randomly at the beginning of evolution. The chromosomes are selected by means of the tournament operator with the base 4. Experiments have shown that the crossover operator is not very suitable to evolve programs using linear encoding, however, in this case a larger change in the chromosomes allows to increase the convergence of the GA, therefore we use one-point crossover operator with the probability 5%. Mutation represents a basic genetic operation to evolve the programs and is performed by generating a new random value for a given gene. In this paper four genes of the chromosome are selected randomly, each of which is mutated with the probability 80%.

Each chromosome is evaluated as the complete test of the sorting network generated by the corresponding CA. The fitness is calculated as the number of correct output bits of the sorting network using all the binary input test vectors. For example, there are 2^{16} test vectors in case of 16-input sorting network. Therefore, the fitness value of a perfect solution is $F_{max} = 16 \cdot 2^{16} = 1048576$. If the maximum fitness is not reached until 100k generations, then the evolutionary run is terminated and considered as unsuccessful.

6 Experimental Results and Discussion

The experiments were focused of the evolutionary development of 16-input sorting networks by means of one-dimensional uniform cellular automata. 16-input networks were chosen as a benchmark problem for the proposed developmental encoding. This section presents the obtained results and discusses their properties as well as the features of the proposed approach. Note that the analysis of results (i.e. the number of generations and properties of obtained sorting networks) is performed for successful runs only (in which the fitness reached the maximal value F_{max}).

Table 2. Statistical results of the CA-based sorting network development using the homogeneous 0-valued initial state. The success rate is calculated as the number of successful experiments out of 100 independent evolutionary runs.

parameters			comparators			generations [$\times 10^3$]			program length			success
initial st.	prog. len.	steps	average	max	min	average	max	min	average	max	min	rate
0000	10	13	–	–	–	–	–	–	–	–	–	0
0000	16	13	–	–	–	–	–	–	–	–	–	0
0000	20	13	99.2 $\pm$2.8	104	97	39.2 $\pm$21.1	63.9	7.6	17.2 $\pm$0.8	18	16	4
0000	26	13	99.7 $\pm$0.9	101	99	68.1 $\pm$21.9	94.9	41.2	25.0 $\pm$0.8	26	24	3
0000	10	14	104.0 $\pm$0.0	104	104	32.3 $\pm$29.1	72.3	3.8	9.7 $\pm$0.5	10	9	3
0000	16	14	106.2 $\pm$4.4	112	99	43.9 $\pm$24.8	84.7	7.6	15.2 $\pm$0.9	16	13	13
0000	20	14	109.1 $\pm$2.8	112	104	42.1 $\pm$30.4	98.3	3.7	18.8 $\pm$1.0	20	17	12
0000	26	14	106.8 $\pm$4.2	112	99	27.5 $\pm$23.1	82.7	2.4	24.3 $\pm$0.8	25	23	20

The experiments have shown that the initial CA state represents a crucial parameter to achieve working results with a reasonable success rate. Table 2 shows the statistical results for a set of experiments considering a homogeneous initial state consisting of only 0-state cells. In some cases the evolution has succeeded and found programs that generate sorting networks using this initial state. We have determined that the zero boundary conditions of the CA play an important role in this set of experiments. If the CA state changes to an other homogeneous state after the first development step, the zero boundary conditions provide different state values in the cellular neighborhood allowing the cell states to diverse during the subsequent steps and to generate suitable comparator arrangements. As evident from Table 2, the success rate depends especially on the number of development steps and the length of the program to be evolved. We have determined that 20 instruction in a chromosome and 13 development steps are required to develop a working sorting network in this set of experiments. As expectable, more development steps allows to generate sorting networks with a higher success rate because more comparators can be generated. However, no clear dependence has been observed for increasing the program length for a given number of development steps. In some cases the success rate is even lower for larger number of instructions in the chromosome. This may be caused by increasing the cardinality of the search space in which the evolution probably needs more generations to find a working solution.

Table 3. Statistical results of the CA-based sorting network development using the alternating initial state consisting of values 2 and 0. The success rate is calculated as the number of successful experiments out of 100 independent evolutionary runs.

parameters			comparators			generations [$\times 10^3$]			program length			success
initial st.	prog. len.	steps	average	max	min	average	max	min	average	max	min	rate
2020	10	13	94.0 $\pm$0.0	94	94	95.4 $\pm$0.0	95.4	95.4	10.0 $\pm$0.0	10	10	1
2020	16	13	96.4 $\pm$3.1	100	90	28.2 $\pm$27.2	89.4	5.9	14.4 $\pm$0.7	16	14	7
2020	20	13	96.3 $\pm$2.4	99	94	40.7 $\pm$13.6	57.2	19.2	19.3 $\pm$0.9	20	18	6
2020	26	13	95.5 $\pm$1.5	97	94	48.7 $\pm$35.3	84.1	13.3	24.5 $\pm$0.5	25	24	2
2020	10	14	102.6 $\pm$2.9	108	90	34.8 $\pm$27.2	93.9	0.5	9.3 $\pm$1.0	10	7	31
2020	16	14	102.5 $\pm$2.8	106	90	35.0 $\pm$27.2	89.4	1.6	15.0 $\pm$1.0	16	13	45
2020	20	14	103.8 $\pm$4.0	112	90	38.7 $\pm$31.5	98.8	0.4	18.8 $\pm$1.2	20	15	43
2020	26	14	103.1 $\pm$3.1	110	90	28.3 $\pm$25.4	83.6	0.8	24.1 $\pm$1.2	26	21	36

Table 3 shows the statistical results for the experiments considering an initial state with alternating values 2 and 0 (an alternating initial state). It is interesting to observe that the success rate has increased in most cases just by changing the initial CA state. Moreover, the evolution was able to find a solution even for 10 or 16 instructions in the chromosomes. Although the success rate is not very high for 13 development steps, it has increased substantially for 14 steps (Table 3). This fact indicates that the conditions for generating working sorting networks are highly dependent on the initial state of the CA. However, it is very difficult to identify the optimal initial state for 16-cell CA in which each cell may possess one of 16 different values. The proposed initial states were determined experimentally and the evolution has not optimized them in this stage of research.

Another interesting feature can be observed in the number of comparators of the resulting sorting networks which exhibits lower values for the alternating initial state (Table 3) in comparison with the experiments considering the homogeneous state (Table 2). This fact supports the hypothesis that the initial state does not only influence the success rate but also the properties of the target sorting networks. Note that the delay of the sorting networks are determined by the number of development steps of the CA because the proposed encoding ensures that the comparators generated in a single step can be performed in parallel.

The approach proposed in this paper exhibits the following features in comparison with other techniques. In [4], two techniques were applied: (1) the absolute encoding provided a sorting network with 75 comparators whose delay is 16 and (2) the relative encoding provided a sorting network with 92 comparators whose delay is 14. In this paper, the best resulting sorting network consists of 78 comparators and its delay is 13. This network has been obtained by removing redundant comparators from a SN consisting of 104 comparators. The resulting sorting network and the corresponding program according to which the CA generated that network is shown in Figure 3. However, the average delay achieved for different developmental setups in [4] was greater than 20 whilst in this paper the delay is limited by the number of development steps of the CA (i.e. for 13 development steps the delay can not be larger). The currently best-known

16-input sorting network consists of 60 comparators and its delay is 10 (e.g. see [6]). Note that the significant difference in the proposed approach is that we have used a developmental encoding whilst the currently best known result was obtained using a direct representation with an explicit area/delay optimization mechanism.

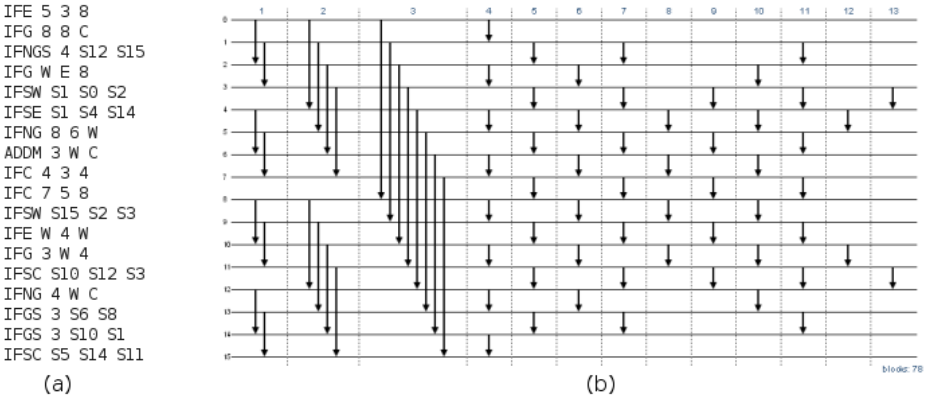

Fig. 3. (a) Evolved program for calculating the transition function of cellular automaton, (b) a sorting network developed by the CA (redundant comparators were removed).

7 Conclusions

A generative approach was introduced for the development of sorting networks by means of cellular automata. The encoding of sorting network comparators is based on the positions of cell in the cellular automaton and the cell states. The transition function of the cellular automaton is represented by a program (consisting of simple application-specific instructions) that is a subject of evolutionary design process. It was shown that the proposed method is able to generate sorting networks whose properties are significantly influenced by the initial state of the cellular automaton. Since the identification of a proper initial state represents a difficult task, it was performed experimentally for the presented case studies. In order to increase the success rate and the quality of the resulting solutions, more research is needed. Therefore, the design of initial states by means of evolution as well as the utilization of advanced evolutionary techniques for the design of the transition function represent areas in which the future experiments will be performed.

Acknowledgment

This work was supported by the Czech science foundation project P103/10/1517.

References

1. Bidlo, M., Vasicek, Z.: Gate-level evolutionary development using cellular automata. In: Proc. of The 3nd NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2008. pp. 11–18. IEEE Computer Society (2008)
2. Bidlo, M., Vasicek, Z.: Comparison of the uniform and non-uniform cellular automata-based approach to the development of combinational circuits. In: Proc. of The 4th NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2009. pp. 423–430. IEEE Computer Society (2009)
3. Bidlo, M., Vasicek, Z.: Instruction-based development of cellular automata. In: Proc. of The 2012 IEEE Congress on Evolutionary Computatio, CEC 2012. IEEE Computer Society (2012)
4. Bidlo, M., Vasicek, Z., Slany, K.: Sorting network development using cellular automata. In: Proc. of the 9th International Conference on Evolvable Systems: From Biology to Hardware (ICES 2010), Lecture Notes in Computer Science, vol. 6274. pp. 85–96. Springer, Berlin Heidelberg New York (2010)
5. Bidlo, M., Škarvada, J.: Instruction-based development: From evolution to generic structures of digital circuits. International Journal of Knowledge-Based and Intelligent Engineering Systems 12(3), 221–236 (2008)
6. Choi, S.S., Moon, B.R.: Isomorphism, normalization, and a genetic algorithm for sorting network optimization. In: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 327–334. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2002)
7. Haddow, P.C., Tufte, G.: Bridging the genotype–phenotype mapping for digital FPGAs. In: Proc. of the 3rd NASA/DoD Workshop on Evolvable Hardware. pp. 109–115. IEEE Computer Society, Los Alamitos, CA, US (2001)
8. J. R. Koza and M. A. Keane and M. J. Streeter and W. Mydlowec and J. Yu and G. Lanza: Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers (2003)
9. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching (2nd ed.). Addison Wesley (1998)
10. Miller, J.F.: Evolving developmental programs for adaptation, morphogenesis and self-repair. In: Advances in Artificial Life. 7th European Conference on Artificial Life, Lecture Notes in Artificial Intelligence, volume 2801. pp. 256–265. Springer, Dortmund DE (2003)
11. Miller, J.F., Thomson, P.: A developmental method for growing graphs and circuits. In: Proc. of the 5th Conf. on Evolvable Systems: From Biology to Hardware (ICES 2003), Lecture Notes in Computer Science, vol. 2606. pp. 93–104. Springer-Verlag, Berlin DE (2003)
12. von Neumann, J.: The Theory of Self-Reproducing Automata. A. W. Burks (ed.), University of Illinois Press (1966)
13. S. Kumar and P. J. Bentley (eds.): On Growth, Form and Computers. Elsevier Academic Press (2003)
14. Tufte, G., Haddow, P.C.: Towards development on a silicon-based cellular computing machine. Natural Computing 4(4), 387–416 (2005)