ELSEVIER Theoretical Computer Science 217 (1999) 81–98

Theoretical Computer Science

Computation in artificially evolved, non-uniform cellular automata

Moshe Sipper[a,*], Marco Tomassini[b]

[a] *Logic Systems Laboratory, Swiss Federal Institute of Technology, IN-Ecublens, CH-1015 Lausanne, Switzerland*

[b] *Logic Systems Laboratory, Swiss Federal Institute of Technology, and Computer Science Institute, University of Lausanne, CH-1015 Lausanne, Switzerland*

Abstract

Cellular automata are dynamical systems in which space and time are discrete, that operate according to local interaction rules. Designing such systems to exhibit a specific behavior or to perform a particular task is highly complicated, thus severely limiting their applications. We study *non-uniform* cellular automata, focusing on the *evolution* of such systems to perform computational tasks, via a parallel evolutionary algorithm, known as *cellular programming*. We present the algorithm and demonstrate that high-performance systems can be evolved to perform two non-trivial computational tasks, density and random number generation. © 1999 — Elsevier Science B.V. All rights reserved

1. Introduction

Cellular automata (CA) are dynamical systems in which space and time are discrete. A cellular automaton consists of an array of cells, each of which can be in one of a finite number of possible states, updated synchronously in discrete time steps, according to a local, identical interaction rule. The state of a cell at the next time step is determined by the current states of a surrounding neighborhood of cells [31, 35].

CAs exhibit three notable features: massive parallelism, locality of cellular interactions, and simplicity of basic components (cells). They perform computations in a distributed fashion on a spatially extended grid. As such they differ from the standard approach to parallel computation in which a problem is split into independent sub-problems, each solved by a different processor, later to be combined in order to yield the final solution. CAs suggest a new approach in which complex behavior arises in a bottom-up manner from non-linear, spatially extended, local interactions [14, 20]. This is often referred to as *emergent computation*, meaning the appearance of global

* Corresponding author. Tel.: +41-21-693-2658; fax: +41-21-693-3705; e-mail: Moshe.Sipper@di.epfl.ch.

0304-3975/99/$ - see front matter © 1999 — Elsevier Science B.V. All rights reserved
PII: S0304-3975(98)00151-0

information processing capabilities that are not explicitly represented in the system's elementary components or in their local interconnections [5].

A major impediment preventing ubiquitous computing with CAs stems from the difficulty of utilizing their complex behavior to perform useful computations. Designing CAs to exhibit a specific behavior or to perform a particular task is highly complicated, thus severely limiting their applications. This results from the local dynamics of the system, which renders the design of local rules to perform global computational tasks extremely arduous. Automating the design (programming) process would greatly enhance the viability of CAs [20]. One of the prime motivations for studying CAs stems from the observation that they are naturally suited for hardware implementation, with the potential of exhibiting extremely fast and reliable computation that is robust to noisy input data and component failure [6, 8, 20, 28].

The model investigated in this paper, called *non-uniform cellular automaton* [17], is an extension of the basic uniform CA model. Such automata function in the same way as uniform ones, the only difference being in the cellular rules that need not be identical for all cells. Our main focus is on the *evolution* of non-uniform CAs to perform computational tasks, employing a parallel evolutionary algorithm, known as *cellular programming*. The idea of applying the biological principle of natural evolution to artificial systems, introduced more than three decades ago, has seen impressive growth in the past decade. Usually grouped under the term *evolutionary algorithms* or *evolutionary computation*, we find the domains of genetic algorithms, evolution strategies, evolutionary programming, and genetic programming.

The paper is organized as follows: in Section 2 we formally define the CA model, followed by an exposition of genetic algorithms in Section 3. Section 4 presents the cellular programming algorithm, and its application to two non-trivial problems: density and random number generation. Finally, we present our concluding remarks in Section 5.

2. Cellular automata

A d-dimensional CA consists of a finite or infinite d-dimensional grid of cells, each of which can take on a value from a finite, typically small, set of integers. The value of each cell at time step t is a function of the values of a small local neighborhood of cells at time $t-1$. The cells update their states simultaneously according to a given local rule. [1]

Formally, a *cellular automaton* A is a quadruple

$$A = (S, G, d, f),$$

where S is a finite set of states, G is the cellular neighborhood, $d \in Z^+$ is the dimension of A, and f is the local cellular interaction rule, also referred to as the transition function.

[1] Asynchronous CAs can also be considered, though they will not be treated in this paper.

Given the position of a cell $\boldsymbol{i}$, $\boldsymbol{i} \in Z^d$, in a regular d-dimensional uniform lattice, or *grid* (i.e., $\boldsymbol{i}$ is an integer vector in a d-dimensional space), its *neighborhood G* is defined by

$$G_i = \{\boldsymbol{i}, \boldsymbol{i} + \boldsymbol{r}_1, \boldsymbol{i} + \boldsymbol{r}_2, \ldots, \boldsymbol{i} + \boldsymbol{r}_n\},$$

where n is a fixed parameter that determines the neighborhood size, and $\boldsymbol{r}_j$ is a fixed vector in the d-dimensional space.

The *local transition rule f*

$$f : S^n \to S$$

maps the state $s_i \in S$ of a given cell $\boldsymbol{i}$ into another state from the set S, as a function of the states of the cells in the neighborhood G_i. In uniform CAs f is identical for all cells, whereas in non-uniform ones f may differ between different cells, i.e., f depends on $\boldsymbol{i}, f_i$.

For a finite-size CA of size N (such as those treated herein) a *configuration* of the grid at time t is defined as

$$C(t) = (s_0(t), s_1(t), \ldots, s_{N-1}(t)),$$

where $s_i(t) \in S$ is the state of cell $\boldsymbol{i}$ at time t. The progression of the CA in time is then given by the iteration of the *global mapping F*

$$F : C(t) \to C(t+1), \quad t = 0, 1, \ldots$$

through the simultaneous application in each cell of the local transition rule f. The global dynamics of the CA can be described as a directed graph, referred to as the CA's *phase space* [35].

In this paper, we focus on one-dimensional CAs with two possible states per cell, i.e., $S = \{0, 1\}$. In this case f is a function $f : \{0,1\}^n \to \{0,1\}$ and the neighborhood size n is usually taken to be $n = 2r + 1$ such that

$$s_i(t+1) = f(s_{i-r}(t), \ldots, s_i(t), \ldots, s_{i+r}(t)),$$

where $r \in Z^+$ is a parameter, known as the *radius*, representing the standard one-dimensional cellular neighborhood. We shall furthermore limit ourselves to the $r = 1$ case, i.e., so-called *elementary* CAs, for which the neighborhood size is $n = 3$:

$$f : \{0,1\}^3 \to \{0,1\}, \qquad s_i(t+1) = f(s_{i-1}(t), s_i(t), s_{i+1}(t)).$$

The domain of f is the set of all 2^3 3-tuples, which gives rise to $2^8 = 256$ distinct elementary rules. We will use Wolfram's decimal numbering convention for describing these rules [35].[2] For two-state CAs a configuration of a size N grid at time t is a binary sequence $C(t)$, $C(t) \in \{0,1\}^N$. For finite-size grids, spatially periodic boundary

[2] For example, $f(111) = 1$, $f(110) = 0$, $f(101) = 1$, $f(100) = 1$, $f(011) = 1$, $f(010) = 0$, $f(001) = 0$, $f(000) = 0$, is denoted rule 184.

conditions are frequently assumed, resulting in a circular grid; formally, this implies that cellular indices are computed modulus N.

3. Genetic algorithms

In the 1950s and the 1960s several researchers independently studied evolutionary systems with the idea that evolution could be used as an optimization tool for engineering problems. Central to all the different methodologies is the notion of solving problems by evolving an initially random population of candidate solutions, through the application of operators inspired by natural genetics and natural selection, such that in time "fitter" (i.e., better) solutions emerge [2, 11, 13]. In this paper, we shall concentrate on one type of evolutionary algorithms, namely, *genetic algorithms*.

A genetic algorithm is an iterative procedure that operates by modifying a constant-size population of individuals, each one represented by a finite string of symbols, known as the *genome*, encoding a possible solution in a given problem space. This space, referred to as the *search space*, comprises all possible solutions to the problem at hand. Generally speaking, the genetic algorithm is applied to spaces which are too large to be exhaustively searched.

The standard genetic algorithm proceeds as follows: an initial population of individuals is generated at random or heuristically. At every evolutionary step, known as a *generation*, the individuals in the current population are *decoded* and *evaluated* according to some predefined quality criterion, referred to as the *fitness*, or *fitness function*. To form a new population (the next generation), individuals are *selected* according to their fitness. Many selection procedures are currently in use, one of the simplest being the so-called *fitness-proportionate selection*, where individuals are selected with a probability proportional to their relative fitness [13]. This ensures that the expected number of times an individual is chosen is approximately proportional to its relative performance in the population. Thus, high-fitness ("good") individuals stand a better chance of "reproducing," while low-fitness ones are more likely to disappear.

Selection alone cannot introduce any new individuals into the population, i.e., it cannot find new points in the search space; these are generated by genetically inspired operators, of which the most well known are *crossover* and *mutation*. Crossover is performed with probability p_x (the "crossover probability" or "crossover rate") between two selected individuals, called *parents*, by exchanging parts of their genomes (i.e., encodings) to form two new individuals, called *offspring*. In its simplest form, substrings are exchanged after a randomly selected crossover point. This operator, known as one-point crossover, tends to enable the evolutionary process to move toward "promising" regions of the search space. The mutation operator is introduced to prevent premature convergence to local optima by randomly sampling new points in the search space. It is carried out by flipping bits at random, with some (small) probability p_m. Genetic algorithms are stochastic iterative processes that are not guaranteed to converge. The termination condition may be specified as some fixed, maximal number of generations

```
begin GA
    g := 0    { generation counter }
    Initialize population $P(g)$
    Evaluate population $P(g)$    { i.e., compute fitness values }
    while not done do
      g := g+1
      Select $P(g)$ from $P(g-1)$
      Crossover $P(g)$
      Mutate $P(g)$
      Evaluate $P(g)$
    end while
end GA
```

Fig. 1. Pseudo-code of the standard genetic algorithm.

or as the attainment of an acceptable fitness level. Fig. 1 presents the standard genetic algorithm in pseudo-code format.

Genetic algorithms are often used to solve hard optimization problems, and can be formally cast within a global optimization framework (other formalizations are also possible, such as, e.g., within a machine-learning setting). Let c be a real cost function to be minimized (the maximization case can be analogously attained):

$$c : Q \to R, \qquad \min\{c(q) \mid q \in Q\},$$

where q is a solution subject to the constraint $q \in Q$. Q is called the *admissible space of solutions* and q is an *admissible solution.*

Consider a finite alphabet Σ, of cardinality $|\Sigma|$. An *individual* I of a finite-size genetic population of size p is a string of symbols of length l from Σ

$$I = \sigma_1 \ldots \sigma_l, \qquad \sigma_i \in \Sigma, \quad i = 1, \ldots, l.$$

The *space of individuals* Γ is defined as the set of all possible individuals of size l; this set is of cardinality $|\Sigma|^l$. The alphabet is often binary, $\{0, 1\}$, so that an individual $I \in \{0, 1\}^l$. Each individual in the population *encodes* a solution to the above optimization problem and a subset $\Lambda \subseteq \Gamma$ encodes solutions belonging to the admissible space Q. We assume the existence of a *decoding function* w such that

$$w : \Lambda \to Q.$$

Given this mapping from individuals (also known as genotypes) to admissible problem solutions (also known as phenotypes), the cost function c can be written as u,

$$u : \Lambda \to R.$$

For technical reasons (see [2]) it is customary to transform the cost function u through composition with a function g, resulting in $h = g(u(I))$, $\forall I \in \Lambda$, where

$$h : \Lambda \rightarrow R^{+}.$$

The transformed cost function is then used to assign fitness values to individuals in the population.

The selection operator s,

$$s : \Lambda^{p} \rightarrow \Lambda^{p}$$

can now be applied to individuals in the population, by referring to their fitness values.

New admissible solutions are generated through the operators of crossover (x) and mutation (m). Crossover is applied with probability p_x, generating two new individuals from the two existing ones:

$$x : \Lambda^{2} \rightarrow \Lambda^{2}.$$

Given two individuals $I = \sigma_1 \ldots \sigma_l$ and $J = \tau_1 \ldots \tau_l$, one-point crossover operates by selecting a random integer number i from a uniform distribution over $[1, l-1]$, after which the symbols $i+1, \ldots, l$ are exchanged between I and J, creating I' and J':

$$I' = \sigma_1 \ldots \sigma_i \tau_{i+1} \ldots \tau_l,$$

$$J' = \tau_1 \ldots \tau_i \sigma_{i+1} \ldots \sigma_l.$$

Note that other forms of crossover are also used, e.g., multi-point and uniform [2, 13].

Mutation is applied independently to every symbol of an individual string, with (usually low) probability p_m. Mutation is a monadic operator that maps an individual into another individual:

$$m : \Lambda \rightarrow \Lambda.$$

Given an individual I, mutation creates I'

$$I' = \sigma_1 \ldots \sigma_{i-1} \sigma'_i \sigma_{i+1} \ldots \sigma_l,$$

where i was randomly chosen with probability p_m and σ'_i was randomly chosen from the set Σ (for simplicity we have illustrated above only one mutated symbol). Note that we have assumed above that the crossover and mutation operators are *syntactically closed*, meaning that their range is Λ; this need not always be the case, i.e., the range can be Γ, entailing a "filtering" process to weed out the non-admissible individuals. For more comprehensive formal accounts of genetic algorithms see [2, 30].

Genetic algorithms are ubiquitous nowadays, having been successfully applied to numerous problems from different domains, including optimization, automatic programming, machine learning, economics, operations research, immune systems, ecology, population genetics, studies of evolution and learning, and social systems. For a recent review of the current state of the art, refer to [33].

The implementation of an evolutionary algorithm, an issue which usually remains in the background, is quite costly in many cases, since populations of solutions are involved, coupled with computation-intensive fitness evaluations. One possible solution is to parallelize the process, an idea which has been explored to some extent in recent years (see reviews [3, 33]). While posing no major problems in principle, this may require judicious modifications of existing algorithms or the introduction of new ones in order to meet the constraints of a given parallel machine. The cellular programming algorithm, described in the next section, is inherently parallel and local, lending itself more readily to implementation.

4. Cellular programming

4.1. The algorithm

We study two-state, one-dimensional, non-uniform CAs, in which each cell may contain a different rule. A cell's rule table is encoded as a bit string (the genome), containing the next-state (output) bits for all possible neighborhood configurations, listed in lexicographic order (as explained in Section 2). Rather than employ a *population* of evolving, uniform CAs, as with the standard genetic algorithm, our algorithm involves a *single*, non-uniform CA of size N, with cell rules initialized at random. Initial configurations are then generated at random, in accordance with the task at hand, and for each one the CA is run for M time steps. Each cell's *fitness* is accumulated over $C = 300$ initial configurations, where a single run's score is 1 if the cell is in the correct state after M time steps, and 0 otherwise. After every C configurations evolution of rules occurs by applying crossover and mutation. This evolutionary process is performed in a completely *local* manner, where genetic operators are applied only between directly connected cells. It is driven by $nf_i(c)$, the number of fitter neighbors of cell i after c configurations. The pseudo-code of our algorithm is delineated in Fig. 2.

The cellular programming algorithm can be considered as a probabilistic cellular automaton [32, 34], under the following interpretation: referring to the formal CA definition (Section 2) as a quadruple $A = (S, G, d, f)$, each cell can be in one of 256 rules (the number of possible 8-bit genomes). The neighborhood G and the grid dimensionality d are the same as defined above ($r = 1$, $d = 1$). The only element that calls for a different interpretation is the local transition rule f. Up to now, we have only considered *deterministic* rules, however, *stochastic* ones are also possible, giving rise to probabilistic cellular automata [28]. In cellular programming, f, $f : \{0, \ldots, 255\}^3 \to \{0, \ldots, 255\}$, maps the state of a given cell to a new state, through deterministic and stochastic transformations (Fig. 2). Representing f as $f = s \circ x \circ m$, where s, x, and m are the selection, crossover, and mutation operators, respectively (Section 3), then s can be either deterministic or stochastic, but x and m are always stochastic, thus rendering f a stochastic function.

```
for each cell $i$ in CA do in parallel
  initialize rule table of cell $i$
  $f_i = 0$ { fitness value }
end parallel for
$c = 0$ { initial configurations counter }
while not done do
  generate a random initial configuration
  run CA on initial configuration for $M$ time steps
  for each cell $i$ do in parallel
    if cell $i$ is in the correct final state then
      $f_i = f_i + 1$
    end if
  end parallel for
  $c = c + 1$
  if $c$ mod $C = 0$ then { evolve every $C$ configurations}
    for each cell $i$ do in parallel
      compute $nf_i(c)$ { number of fitter neighbors }
      if $nf_i(c) = 0$ then rule $i$ is left unchanged
      else if $nf_i(c) = 1$ then replace rule $i$ with the fitter neighboring rule,
                                 followed by mutation
      else if $nf_i(c) = 2$ then replace rule $i$ with the crossover of the two fitter
                                 neighboring rules, followed by mutation
      else if $nf_i(c) > 2$ then replace rule $i$ with the crossover of two randomly
                                 chosen fitter neighboring rules, followed by mutation
                                 (this case can occur if $r > 1$)
      end if
      $f_i = 0$
    end parallel for
  end if
end while
```

Fig. 2. Pseudo-code of the cellular programming algorithm.

There are two main differences between our algorithm and the standard genetic algorithm (Section 3): (1) The latter involves a population of evolving, uniform CAs, with all individuals *ranked* according to fitness, and crossover occurring between *any* two individuals in the population. Thus, while the CA runs in accordance with a local rule, evolution proceeds in a *global* manner. In contrast, our algorithm proceeds *locally* in the sense that each cell has access only to its locale, not only during the run but also during the evolutionary phase, and no global fitness ranking is performed. (2) The standard genetic algorithm involves a population of *independent* problem solutions, meaning that the individuals in the population are assigned fitness values independent of one another, and interact only through the genetic operators in order to produce the next generation. In contrast, our CA *coevolves* since each cell's fitness depends upon its evolving neighbors.

The cellular programming approach has been applied to several problems, employed to construct evolving hardware, and extended to include other generalizations of the classical CA model (the first generalization being non-uniformity of rules), such as non-standard connectivity architectures and non-deterministic state updating [8, 19–26, 28, 29]. A thorough examination of cellular programming is provided in the recent book by Sipper [20]; for reviews the reader is referred to Sipper [21, 22]. In what follows, we concentrate on two specific, non-trivial problems, namely, density and random number generation.

4.2. The density problem

In the density problem, the one-dimensional, two-state CA is presented with an arbitrary initial configuration, and should converge in time to a state of all 1's if the initial configuration contains a density of 1's >0.5, and to all 0's if this density <0.5; for an initial density of 0.5, the CA's behavior is undefined. Spatially periodic boundary conditions are used, resulting in a circular grid. Formally, let $C(t)=(s_0(t),\dots,s_{N-1}(t))$ be the grid configuration at time step t (Section 2), and let $D(s_i(t),\dots,s_{i+k-1}(t))$ be the density of 1s at time t over a block of k cells at positions $i,\dots,i+k-1$ (with cellular indices computed modulo N, due to the grid's circularity), then the CA's final density (output) is:

$$D(C(M))=\begin{cases} 1 & \text{if } D(C(0))>0.5,\\ 0 & \text{if } D(C(0))<0.5,\\ \text{undefined} & \text{if } D(C(0))=0.5,\end{cases}$$

where M is a given number of time steps to convergence.

It has been noted by Mitchell et al. [14] that the density task comprises a non-trivial computation for a small-radius CA ($r \ll N$, where N is the grid size). Density is a global property of a configuration, whereas a small-radius CA relies solely on local interactions. Since the 1s can be distributed throughout the grid, propagation of information must occur over large distances (i.e., $O(N)$). The minimum amount of memory required for the task is $O(\log N)$ using a serial-scan algorithm, thus the computation involved corresponds to recognition of a non-regular language. It has been shown that for a uniform one-dimensional grid of fixed size N, and for a fixed radius $r \geqslant 1$, there exists no two-state CA rule which correctly classifies all possible initial configurations [12]. This says nothing, however, about how well an imperfect CA might perform, one possible approach for obtaining successful CAs being artificial evolution.

The application of a standard genetic algorithm (Section 3) to the evolution of *uniform*, one-dimensional, $r=3$ CAs that solve this task was studied in [14, 15]. More recently, Andre et al. [1] used genetic programming [11], an evolutionary computation

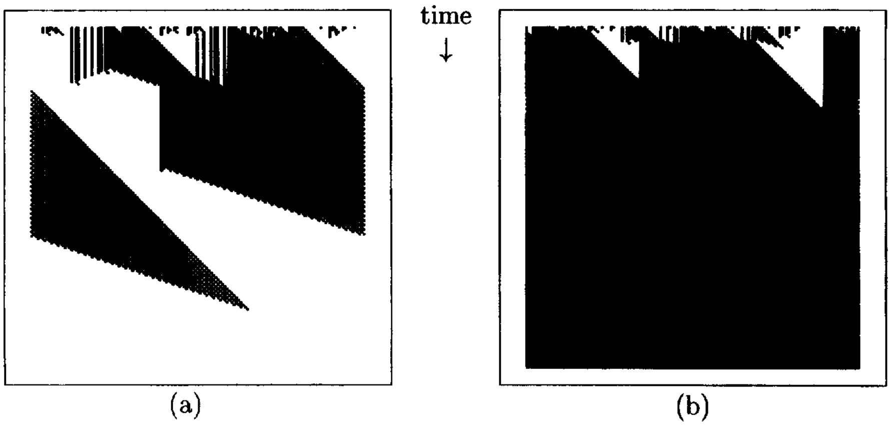

Fig. 3. The density task: operation of the GKL rule. CA is one-dimensional, uniform, 2-state, with connectivity radius $r = 3$. Grid size is $N = 149$. White squares represent cells in state 0, black squares represent cells in state 1. The pattern of configurations is shown through time (which increases down the page). Initial configurations were generated at random. (a) Initial density of 1's is 0.47. (b) Initial density of 1's is 0.53. The CA relaxes in both cases to a fixed pattern of all 0's or all 1's, correctly classifying the initial configuration.

paradigm in which an individual in the population is represented by a LISP expression, to evolve uniform CAs to perform the density task. It was shown that high-performance CAs can indeed be evolved.[3] One of the best known uniform, $r = 3$ CAs for this task is in fact a human-designed one, known as GKL, defined as follows [7]:

$$s_i(t+1) = \begin{cases} \text{majority}[s_i(t), s_{i-1}(t), s_{i-3}(t)] & \text{if } s_i(t) = 0, \\ \text{majority}[s_i(t), s_{i+1}(t), s_{i+3}(t)] & \text{if } s_i(t) = 1. \end{cases}$$

Its operation is demonstrated in Fig. 3.

We have studied the density task, as defined above, using non-uniform, one-dimensional, minimal radius $r = 1$ CAs of size $N = 149$. The search space involved is extremely large; since each cell contains one of 2^8 possible rules this space is of size $(2^8)^{149} = 2^{1192}$. In contrast, the size of *uniform*, $r = 1$ CA rule space is small, consisting of only $2^8 = 256$ rules. This enabled us to test each and every one of these rules on the density task, a feat not possible for larger values of r. One of our major results is that evolved, non-uniform, $r = 1$ CAs outperform any possible uniform, $r = 1$ CA [19, 20].[4]

For the cellular programming algorithm we used randomly generated initial configurations, uniformly distributed over densities in the range [0, 1], with the CA being run for $M = 150$ time steps (thus, the CA's computation time is linear with grid size). We found that non-uniform CAs had coevolved that exhibit high performance on this

[3] For the precise performance measures used refer to the aforementioned references.

[4] For details on the performance comparison see [19, 20].

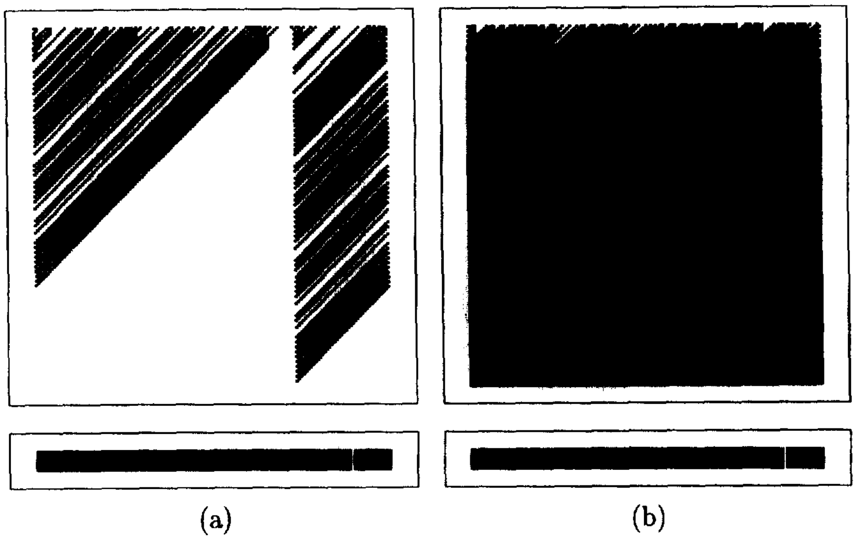

Fig. 4. One-dimensional density task: operation of a coevolved, non-uniform, $r = 1$ CA. Grid size is $N = 149$. Top figures depict space–time diagrams, bottom figures depict rule maps. (a) Initial density of 1's is 0.40, final density is 0. (b) Initial density of 1's is 0.60, final density is 1.

task. Furthermore, these consist of a grid in which one rule dominates, a situation referred to as *quasi-uniformity* [18, 20]. Fig. 4 demonstrates the operation of one such coevolved CA along with a rules map, depicting the distribution of rules by assigning a unique gray level to each distinct rule. In this example the grid consists of 146 cells containing rule 226, 2 cells containing rule 224, and 1 cell containing rule 234.

Most investigations of the density problem to date concentrated on the above statement of the problem, which specifies convergence to one of two fixed-point configurations, that are considered as the output of the computation. Recently, Capcarrere et al. [4] showed that a perfect CA density classifier exists, upon defining a different output specification. Consider the uniform, two-state, $r = 1$ rule-184 CA, defined as follows:

$$s_i(t+1) = \begin{cases} s_{i-1}(t) & \text{if } s_i(t) = 0, \\ s_{i+1}(t) & \text{if } s_i(t) = 1. \end{cases}$$

Upon presentation of an arbitrary initial configuration, the grid relaxes to a limit cycle, within $\lceil N/2 \rceil$ time steps, that provides a classification of the initial configuration's density of 1's: if this density >0.5 (respectively <0.5), then the final configuration consists of one or more blocks of at least two consecutive 1's (0's), interspersed by an alternation of 0's and 1's; for an initial density of exactly 0.5, the final configuration consists of an alternation of 0s and 1s. The computation's output is given by the state of the consecutive block (or blocks) of same-state cells (Fig. 5). Capcarrere et al. [4] proved the following theorem: for a finite-size CA of size N, let $C(t)$ be the grid

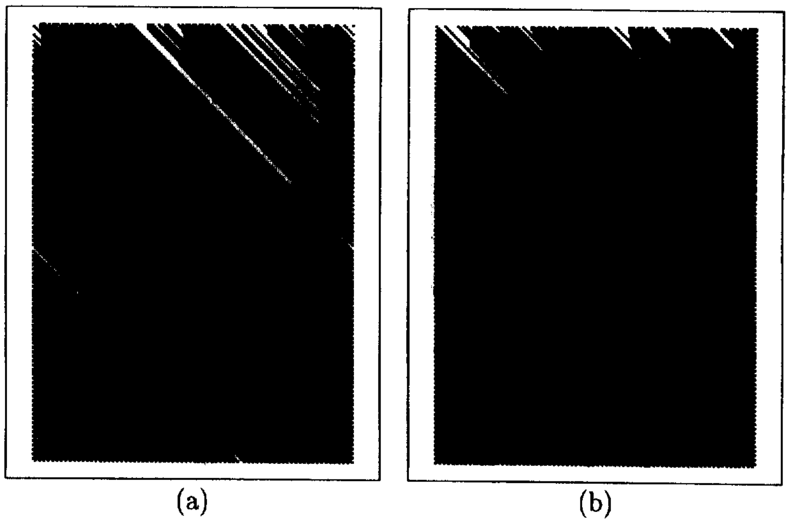

Fig. 5. Density classification: demonstration of the uniform rule-184 CA on two random initial configurations. The pattern of configurations is shown for the first 200 time steps. Grid size is $N = 149$. (a) Initial density is 0.497, i.e., 75 cells are in state 0, and 74 are in state 1. The final configuration consists of an alternation of 0's and 1's with a single block of two cells in state 0. (b) Initial density is 0.537. The final configuration consists of an alternation of 0's and 1's with several blocks of two or more cells in state 1. In both cases the CA correctly classifies the initial configuration.

configuration at time step t, let $D(s_i(t),\dots,s_{i+k-1}(t))$ be the density of 1's at time t over a block of k cells at positions $i,\dots,i+k-1$ (as defined above), and let $T = \lceil N/2 \rceil$. Then

(i) If $D(C(0)) > 0.5$, then (a) there exists a pair of adjacent cells $i, i+1$, such that $s_i(T) = 1$ and $s_{i+1}(T) = 1$; (b) for all i, $s_i(T) = 0 \Rightarrow s_{i+1}(T) = 1$.

(ii) If $D(C(0)) < 0.5$, then (a) there exists a pair of adjacent cells $i, i+1$, such that $s_i(T) = 0$ and $s_{i+1}(T) = 0$; (b) for all i, $s_i(T) = 1 \Rightarrow s_{i+1}(T) = 0$.

(iii) If $D(C(0)) = 0.5$, then for all i, $s_i(T) \neq s_{i+1}(T)$.

Thus, rule 184 performs perfect density classification (including the density $= 0.5$ case). We note in passing that the reflection-symmetric rule 226 holds the same properties of rule 184.

As noted above, the computational complexity of the input is that of a non-regular language, whereas the fixed-point output of the original problem involves a simple regular language (all 0's or all 1's); we note that the novel output specification also involves a regular language (a block of two state-0 or state-1 cells). Capcarrere et al. [4] thus concluded that their newly proposed density classifier is as viable as the original one with respect to computational complexity, while surpassing the latter in terms of performance. As a final remark we mention our recent experiments in which we set out to find whether our local, coevolutionary cellular programming algorithm can discover this optimal CA [27]. We found that starting from a random non-uniform CA, and using an appropriate fitness function, a large percentage of the evolutionary runs ended with the uniform, rule-184 CA – the provenly optimal solution. Thus, a global optimum can be found through a purely local evolutionary process.

4.3. Random number generation

Random numbers are needed in a variety of applications, yet finding good random number generators, or randomizers, is a difficult task [16]. To generate a random sequence on a digital computer, one starts with a fixed-length seed, then iteratively applies some transformation to it, progressively extracting as long as possible a random sequence. Such numbers are usually referred to as *pseudo*-random, as distinguished from true random numbers, resulting from some natural physical process. In order to demonstrate the efficacy of a proposed generator, it is usually subject to a battery of empirical and theoretical tests, such as those described by Knuth [10].

In the last decade, CAs have been used to generate "good" random numbers. The first work examining the application of CAs to random number generation is that of Wolfram [35], in which the uniform, 2-state, $r=1$ rule-30 CA was extensively studied, demonstrating its ability to produce highly random, temporal bit sequences. Such sequences are obtained by sampling the values that a particular cell (usually the central one) attains as a function of time. In Wolfram's work, the uniform rule-30 CA is initialized with a configuration consisting of a single cell in state 1, with all other cells being in state 0 [35]. The initially non-zero cell is the site at which the random temporal sequence is generated.

A non-uniform CA randomizer was presented by Hortensius et al. [9], consisting of two rules, 90 and 150, arranged in a specific order in the grid. The performance of this CA in terms of random number generation was found to be at least as good as that of rule 30, with the added benefit of less costly hardware implementation. It is interesting in that it combines two rules, both of which are simple linear rules, that do not comprise good randomizers, to form an efficient, high-performance generator.

An evolutionary approach for obtaining random number generators was taken by Koza, using genetic programming [11]. He demonstrated evolved expressions that are equivalent to Wolfram's rule 30. The fitness measure used by Koza is the *entropy* E_h: let k be the number of possible values per sequence position (in our case CA states) and h a subsequence length. E_h (measured in bits) for the set of k^h probabilities p_{h_j} of the k^h possible subsequences of length h is given by

$$E_h = -\sum_{j=1}^{k^h} p_{h_j} \log_2 p_{h_j},$$

where $h_1, h_2, \ldots, h_{k^h}$ are all the possible subsequences of length h (by convention, $\log_2 0 = 0$ when computing entropy). The entropy attains its maximal value when the probabilities of all k^h possible subsequences of length h are equal to $1/k^h$; in our case $k=2$ and the maximal entropy is $E_h = h$. Formally, for a given CA, we can define a *temporal sequence* S_i as the sequence of states $s_i(t)$ that cell i assumes through time:

$$S_i = \{s_i(t)\}_{t=0,1,\ldots}, \quad i = 0, 1, \ldots, N-1.$$

Then a necessary (though not sufficient) condition for a sequence of bits to be random is that their entropy be maximal [35].

The above account led us to ask whether good CA randomizers can be coevolved using cellular programming, our results suggesting that this is indeed the case [25, 26]. The algorithm presented in Section 4.1 is slightly modified, such that the cell's fitness score for a single configuration is defined as the entropy E_h of the temporal sequence, after the CA has been run for M time steps. Cell i's fitness value, f_i, is then updated as follows (refer to Fig. 2):

```
for each cell $i$ do in parallel
    $f_i = f_i +$ entropy $E_h$ of the temporal sequence of cell $i$
end parallel for
```

Initial configurations are randomly generated and for each one the CA is run for $M = 4096$ time steps.[5] Note that we do not restrict ourselves to one designated cell, but consider all grid cells, thus obtaining N random sequences in parallel, rather than a single one.

In our simulations (using grids of sizes $N = 50$ and $N = 150$), we observed that the average cellular entropy taken over all grid cells is initially low (usually in the range [0.2, 0.5]), ultimately evolving to a maximum of 3.997, when using a subsequence size of $h = 4$ (i.e., entropy is computed by considering the occurrence probabilities of 16 possible subsequences, using a "sliding window" of length 4). We performed several such experiments using $h = 4$ and $h = 7$. The evolved, non-uniform CAs attained average fitness values (entropy) of 3.997 and 6.978, respectively. We then re-tested our best CAs over $M = 65\,536$ times steps (as in [11]), obtaining entropy values of 3.9998 and 6.999, respectively. Interestingly, when we performed this test with $h = 7$ for CAs which were evolved using $h = 4$, high entropy was displayed, as for CAs which were originally evolved with $h = 7$. These results are comparable to the entropy values obtained by Koza [11], as well as to those of the rule-30 CA of Wolfram [35] and the non-uniform, rules $\{90, 150\}$ CA of Hortensius et al. [9]. Note that while our fitness measure is local, the evolved entropy results reported above represent the average of *all* grid cells. Thus, we obtain N random sequences in parallel rather than a single one. Fig. 6 demonstrates the operation of three CAs discussed above: rule 30, rules $\{90, 150\}$, and a coevolved CA.

We next subjected our evolved CAs to a number of additional tests, including chi-square (χ^2), serial correlation coefficient, and a Monte Carlo simulation for calculating the value of π; these are well-known tests described in detail by Knuth [10]. In order to apply the tests we generated sequences of 100 000 random bytes using two different procedures: (a) The CA of size $N = 50$ is run for 500 time steps, thus generating 50 random temporal bit sequences of length 500. These are concatenated to form one long

[5] A standard, 48-bit, linear congruential algorithm proved sufficient for the generation of random initial configurations.

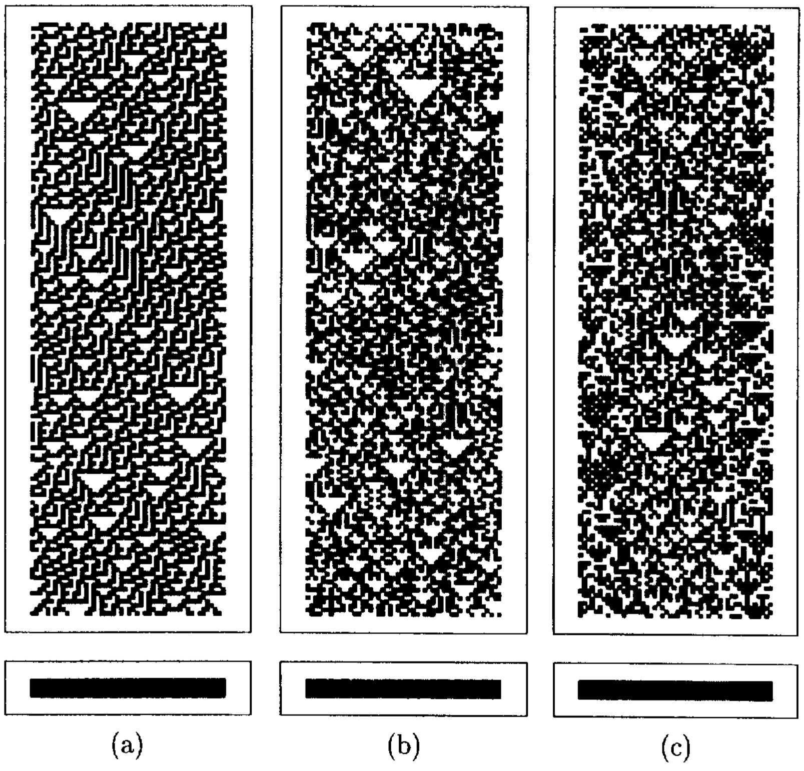
Fig. 6. One-dimensional random number generators: operation of three CAs. Grid size is $N = 50$, radius is $r = 1$. Initial configurations were generated by randomly setting the state of each grid cell to 0 or 1 with uniform probability. Top figures depict space–time diagrams, bottom figures depict rule maps. (a) Rule-30 CA. (b) Rules {90, 150} CA. (c) A coevolved, non-uniform CA, consisting of three rules: rule 165 (22 cells), rule 90 (22 cells), and rule 150 (6 cells).

sequence of length 25 000 bits. This process is then repeated 32 times, thus obtaining a sequence of 800 000 bits, which are grouped into 100 000 bytes. (b) The CA of size $N = 50$ is run for 400 time steps. Every 8 time steps, 50 8-bit sequences (bytes) are produced, which are concatenated, resulting in 2500 bytes after 400 time steps. This process is then repeated 40 times, thus obtaining the 100 000 byte sequence.

Table 1 shows the test results of four random number generators: two coevolved CAs, rule-30 CA, and the rules {90, 150} CA. We note that the two coevolved CAs attain good results on all tests, most notably chi-square which is one of the most significant ones [10]. Our results are somewhat better than the rules {90, 150} CA, and markedly improved in comparison to the rule-30 CA, which attains lower scores on the chi-square test (procedure (a)), and on the serial correlation test (procedure (b)). It is noteworthy that our CAs attain good results on a number of tests, while the fitness measure used during evolution is entropy alone. The relatively low results obtained by the rule-30 CA may be due to the fact that we considered N random sequences

Table 1
Results of tests. Each entry represents the test result for a sequence of 100,000 bytes, generated by the corresponding randomizer. 20 sequences were generated by each randomizer, 10 by procedure (a) and 10 by procedure (b) (see text). The table lists the chi-square test results for all 10 sequences and the first 5 results for the other tests. CA Grid size is $N = 50$. Coevolved CA (1) consists of three rules: rule 165 (22 cells), rule 90 (22 cells), and rule 150 (6 cells). Coevolved CA (2) consists of two rules: rule 165 (45 cells) and rule 225 (5 cells). Interpretation of the listed values is as follows (for a full explanation see Knuth [10]): (i) Chi-square test: "good" results are between 10%–90%, with extremities on both sides (i.e., <10% and >90%) representing non-satisfactory random sequences. The total percentage of sequences passing the chi-square test is listed below the 10 individual test results. Knuth suggested that at least three sequences from a generator be subject to the chi-square test and if a majority pass then the generator is considered to have passed (with respect to chi-square). (ii) Serial correlation coefficient: this value should be close to zero. (iii) Entropy test: this value should be close to 8. (iv) Monte Carlo π: the random number sequence is used in a Monte Carlo computation of the value of π, and the error percentage from the actual value is shown

	Coevolved CA (1)		Coevolved CA (2)		Rule 30 CA		Rules {90,150} CA	
Test	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
	50.00%	75.00%	50.00%	50.00%	90.00%	90.00%	50.00%	25.00%
	50.00%	50.00%	75.00%	50.00%	10.00%	50.00%	5.00%	50.00%
	90.00%	50.00%	95.00%	5.00%	97.50%	0.50%	10.00%	50.00%
	25.00%	75.00%	50.00%	50.00%	0.01%	50.00%	75.00%	25.00%
(i)	50.00%	25.00%	75.00%	50.00%	95.00%	75.00%	97.50%	25.00%
	25.00%	10.00%	75.00%	25.00%	97.50%	50.00%	25.00%	50.00%
	75.00%	50.00%	75.00%	75.00%	50.00%	50.00%	25.00%	50.00%
	10.00%	50.00%	25.00%	50.00%	5.00%	50.00%	25.00%	50.00%
	50.00%	25.00%	50.00%	75.00%	25.00%	50.00%	95.00%	75.00%
	90.00%	75.00%	90.00%	10.00%	25.00%	50.00%	75.00%	75.00%
	100%	100%	90%	90%	50%	90%	70%	100%
	0.00185	−0.00085	−0.00390	0.01952	0.00052	−0.24685	0.00646	0.00036
	−0.00386	−0.00228	0.00228	0.02144	−0.00175	−0.24838	−0.00071	−0.00194
(ii)	0.00192	−0.00297	0.00048	0.01970	0.00156	−0.24291	0.00205	−0.00322
	−0.00011	−0.00248	−0.00237	0.02192	0.00478	−0.23735	0.00177	0.00094
	−0.00060	−0.00762	0.00194	0.01937	0.00214	−0.24149	−0.00075	0.00378
	7.99819	7.99828	7.99807	7.99827	7.99841	7.99842	7.99821	7.99797
	7.99821	7.99817	7.99835	7.99810	7.99789	7.99820	7.99788	7.99807
(iii)	7.99838	7.99810	7.99845	7.99786	7.99849	7.99770	7.99793	7.99809
	7.99800	7.99831	7.99806	7.99808	7.99733	7.99807	7.99832	7.99804
	7.99808	7.99801	7.99829	7.99808	7.99844	7.99835	7.99851	7.99800
	0.54%	0.19%	0.42%	0.16%	0.21%	0.90%	0.52%	0.20%
	0.03%	0.12%	0.33%	0.35%	0.21%	0.13%	0.05%	0.07%
(iv)	0.18%	0.68%	0.62%	0.65%	0.32%	0.13%	0.27%	0.07%
	0.45%	0.73%	0.48%	0.33%	0.37%	0.38%	0.07%	0.17%
	0.16%	0.09%	0.12%	0.13%	0.40%	0.08%	0.78%	0.01%

generated in parallel, rather than the single one considered by Wolfram. We note in passing that the rules {90, 150} CA results may probably be somewhat improved (as perhaps our own results) by using "site spacing" and "time spacing" [9].

5. Concluding remarks

A major impediment preventing ubiquitous computing with CAs stems from the difficulty of utilizing their complex behavior to perform useful computations. We presented the cellular programming algorithm for coevolving computation in non-uniform CAs, demonstrating that high-performance systems can be evolved for non-trivial computational tasks.

We applied the algorithm to the density problem, showing that non-uniform CAs can be evolved to solve it. Restating the problem, we discussed a uniform CA that can perfectly perform the task, and remarked that our algorithm can be used to evolve the optimal solution.

We showed that the cellular programming algorithm can be applied to the difficult problem of generating random number generators. While a more extensive suite of tests is in order, it seems safe to say at this point that our coevolved generators are at least as good as the best available CA randomizers. Furthermore, there is a notable advantage arising from the existence of a "tunable" algorithm for the generation of randomizers. As some rules lend themselves more easily to hardware implementation, our algorithm may be used to find good randomizers which can also be efficiently implemented. A possible extension is the addition of restrictions to the evolutionary process, e.g., by prespecifying rules for some cells, in order to accommodate hardware constraints. Another possible modification of the evolutionary process is the incorporation of statistical measures of randomness into the fitness function (and not just as an aftermath benchmark). These possible extensions could lead to the automatic generation of high-performance, random number generators, meeting specific user demands.

Evolving non-uniform CAs present many interesting questions for future research, involving global computation in locally interconnected systems, as well as how to attain them by means of artificial evolution. We hope this work has shed some light on these issues.

References

[1] D. Andre, F.H. Bennett III, J.R. Koza, Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem, in: J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo (Eds.), Genetic Programming 1996: Proc. 1st Ann. Conf., The MIT Press, Cambridge, MA, 1996, pp. 3–11.

[2] T. Bäck, Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms, Oxford University Press, New York, 1996.

[3] E. Cantú-Paz, A summary of research on parallel genetic algorithms, Tech. Rep. 95007, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL, July 1995.

[4] M.S. Capcarrere, M. Sipper, M. Tomassini, Two-state, $r = 1$ cellular automaton that classifies density, Phys. Rev. Lett. 77 (24) (1996) 4969–4971.

[5] S. Forrest (Ed.), Emergent Computation: Self-organizing, Collective, and Cooperative Phenomena in Natural and Artificial Computing Networks, The MIT Press, Cambridge, MA, 1991.

[6] P. Gacs, Nonergodic one-dimensional media and reliable computation, Contemp. Math. 41 (1985) 125.

[7] P. Gacs, G.L. Kurdyumov, L.A. Levin, One-dimensional uniform arrays that wash out finite islands, Problemy Peredachi Informatsii 14 (1978) 92–98.

[8] M. Goeke, M. Sipper, D. Mange, A. Stauffer, E. Sanchez, M. Tomassini, Online autonomous evolware, in: T. Higuchi, M. Iwata, W. Liu (Eds.), Proc. 1st Internat. Conf. on Evolvable Systems: From Biology to Hardware (ICES96), Lecture Notes in Computer Science, vol. 1259, Springer, Heidelberg, 1997, pp. 96–106.
[9] P.D. Hortensius, R.D. McLeod, H.C. Card, Parallel random number generation for VLSI systems using cellular automata, IEEE Trans. Comput. 38 (10) (1989) 1466–1473.
[10] D.E. Knuth, The Art of Computer Programming: Vol. 2, Seminumerical Algorithms, 2nd ed., Addison-Wesley, Reading, MA, 1981.
[11] J.R. Koza, Genetic Programming, The MIT Press, Cambridge, MA, 1992.
[12] M. Land, R.K. Belew, No perfect two-state cellular automata for density classification exists, Phys. Rev. Lett. 74(25) (1995) 5148–5150.
[13] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, 3rd ed., Springer, Heidelberg, 1996.
[14] M. Mitchell, J.P. Crutchfield, P.T. Hraber, Evolving cellular automata to perform computations: Mechanisms and impediments, Physica D 75 (1994) 361–391.
[15] N.H. Packard, Adaptation toward the edge of chaos, in: J.A.S. Kelso, A.J. Mandell, M.F. Shlesinger (Eds.), Dynamic Patterns in Complex Systems, World Scientific, Singapore, 1988, pp. 293–301.
[16] S.K. Park, K.W. Miller, Random number generators: good ones are hard to find, Comm. ACM 31(10) (1988) 1192–1201.
[17] M. Sipper, Non-uniform cellular automata: evolution in rule space and formation of complex structures, in: R.A. Brooks, P. Maes (Eds.), Artificial Life IV, The MIT Press, Cambridge, MA, 1994, pp. 394–399.
[18] M. Sipper, Quasi-uniform computation-universal cellular automata, in: F. Morán, A. Moreno, J.J. Merelo, P. Chacón (Eds.), ECAL'95: 3rd European Conf. on Artificial Life, Lecture Notes in Computer Science, vol. 929, Springer, Heidelberg, 1995, pp. 544–554.
[19] M. Sipper, Co-evolving non-uniform cellular automata to perform computations, Physica D 92 (1996) 193–208.
[20] M. Sipper, Evolution of Parallel Cellular Machines: The Cellular Programming Approach, Springer, Heidelberg, 1997.
[21] M. Sipper, The evolution of parallel cellular machines: toward evolware, BioSystems 42 (1997) 29–43.
[22] M. Sipper, Evolving uniform and non-uniform cellular automata networks, in: D. Stauffer (Ed.), Annual Reviews of Computational Physics, vol. V, World Scientific, Singapore, 1997, pp. 243–285.
[23] M. Sipper, E. Ruppin, Co-evolving cellular architectures by cellular programming, in: Proc. IEEE 3rd Internat. Conf. on Evolutionary Computation (ICEC'96), 1996, pp. 306–311.
[24] M. Sipper, E. Ruppin, Co-evolving architectures for cellular machines, Physica D 99 (1997) 428–441.
[25] M. Sipper, M. Tomassini, Co-evolving parallel random number generators, in: H.-M. Voigt, W. Ebeling, I. Rechenberg, H.-P. Schwefel (Eds.), Parallel Problem Solving from Nature – PPSN IV, Lecture Notes in Computer Science, vol. 1141, Springer, Heidelberg, 1996, pp. 950–959.
[26] M. Sipper, M. Tomassini, Generating parallel random number generators by cellular programming, Internat. J. Modern Phys. C 7(2) (1996) 181–190.
[27] M. Sipper, M. Tomassini, Convergence to uniformity in a cellular automaton via local coevolution, Internat. J. Modern Phys. C 8(5) (1997) 1013–1024.
[28] M. Sipper, M. Tomassini, O. Beuret, Studying probabilistic faults in evolved non-uniform cellular automata, Internat. J. Modern Phys. C 7(6) (1996) 923–939.
[29] M. Sipper, M. Tomassini, M.S. Capcarrere, Designing cellular automata using a parallel evolutionary algorithm, Nucl. Instr. Meth. A 389(1–2) (1997) 278–283.
[30] A. Tettamanzi, Algoritmi Evolutivi per L'ottimizzazione, Ph.D. Thesis, Computer Science Department, University of Milano, 1995.
[31] T. Toffoli, N. Margolus, Cellular Automata Machines, The MIT Press, Cambridge, MA, 1987.
[32] M. Tomassini, The parallel genetic cellular automata: application to global function optimization, in: R.F. Albrecht, C.R. Reeves, N.C. Steele (Eds.), Proc. Internat. Conf. on Artificial Neural Networks and Genetic Algorithms, Springer, Berlin, 1993, pp. 385–391.
[33] M. Tomassini, Evolutionary algorithms, in: E. Sanchez, M. Tomassini (Eds.), Towards Evolvable Hardware, Lecture Notes in Computer Science, vol. 1062, Springer, Heidelberg, 1996, pp. 19–47.
[34] D. Whitely, Cellular genetic algorithms, in: S. Forrest (Ed.), Proc. 5th Internat. Conf. on Genetic Algorithms, Morgan Kaufmann Publishers, San Mateo, CA, 1993, p. 658.
[35] S. Wolfram, Cellular Automata and Complexity, Addison-Wesley, Reading, MA, 1994.