Cellular Automaton-Based Pseudorandom Number Generator

Zakarya Zarezadeh
Department of Philosophy, Literature and Art
University of Tor Vergata, 18-00173, Rome, Italy

This paper is concerned with the study of pseudorandom number generation by an extension of the original cellular automaton, termed nonuniform cellular automata. In order to demonstrate the efficacy of a proposed random number generator, it is usually subject to a battery of empirical and theoretical tests. By using a standard software package for statistically evaluating the quality of random number sequences known as the Diehard battery test suite and TestU01, the results of the proposed model are validated, and we demonstrate that cellular automata can be used to rapidly produce purely random temporal bit sequences to an arbitrary precision.

1. Introduction

Random numbers are very important for a variety of purposes, such as Monte Carlo techniques, simulated annealing and Brown dynamics. These simulation methods depend critically on the quality of the random numbers, as measured by appropriate statistical tests. When a large amount of random numbers has to be generated, computational efficiency is also very important. Built-in devices for very large-scale integration (VLSI) circuits are another application area of random numbers. In this case, as well as for fine-grained massively parallel computers and for on-board applications, it is essential that the random number generator also be amenable to hardware implementation in terms of area, number of gates and complexity of the interconnections. Several deterministic algorithms for producing random numbers have been proposed to date. In Section 2, we will review the principal pseudorandom number generators (PRNGs) methods. In Section 3, we will concentrate on generating pseudorandom sequences by using cellular automata (CAs). The proposed cellular automaton (CA) model is described in Section 3.2. Our results in Section 5 show that CA-based PRNGs can yield long-period, high-quality random number sequences. We conclude that CA-based random number generators offer a realistic alternative to other methods; this is especially true in the case of VLSI implementation,

https://doi.org/10.25088/ComplexSystems.26.4.373

fine-grained massively parallel machines for statistical physics simulation and built-in self-test circuits.

2. Random Number Generators

2.1 An Overview of Random Number Generators

A PRNG is an algorithm for generating a sequence of numbers that approximates the properties of random numbers. The sequence is not truly random, in that it is completely determined by a relatively small set of initial values, called the state of the PRNG. Although sequences that are closer to truly random can be generated using hardware random number generators, pseudorandom numbers are important in practice for simulations (e.g., of physical systems with the Monte Carlo method) and are central in the practice of cryptography. The generator is exercised by steps, and two things occur concurrently during each step: there is a transformation of the state information, and the generator outputs a fixed-size bit string. The generator seed is simply the initial state information. With any pseudorandom generator, after a sufficient number of steps, the generator comes back to some sequence of states that was already visited. Then, the period of the generator is the number of steps required to do one full cycle through the visited states. Careful mathematical analysis is required to have any confidence that a PRNG generates numbers that are sufficiently "random" to suit the intended use. All practical methods for obtaining random numbers are based on deterministic algorithms, which is why such numbers are more appropriately called pseudorandom, as distinguished from true random numbers resulting from some natural physical process. In the following we will limit ourselves to uniformly distributed sequences of pseudorandom numbers; however, there are well-known ways for obtaining sequences distributed according to a different distribution, starting from a uniformly distributed one. In practice, the output from many common PRNGs exhibits artifacts that cause it to fail statistical pattern-detection tests. These include, but are certainly not limited to:

1. Shorter-than-expected periods for some seed states; such seed states may be called "weak" in this context.
2. Lack of uniformity of distribution.
3. Correlation of successive values.
4. Poor dimensional distribution of the output sequence.
5. The distances between where certain values occur are distributed differently from those in a random sequence distribution.

Random number generators must possess a number of properties if they are to be useful in lengthy stochastic simulations such as those used in computational physics. One of the most popular methods for generating random numbers is the linear congruential generator. It uses a method similar to the folding schemes in chaotic maps. The general formula is:

$$X_{n+1} = (a\,X_n + c) \bmod m \qquad n \geq 0,$$
$$m > 0, \quad 0 < a < m. \tag{1}$$

The value $m > 0$ is called the modulus, a is the multiplier, and c is an additive constant. If $c = 0$, the generator is a multiplicative congruential generator. Linear congruential generators are periodic and tend to give a lower quality of randomness, especially when a large number of random values are needed. If reals are generated directly from the congruence relation, the period is less than or equal to m. The period of a multiplicative congruential generator is bounded above by the number of positive integers less than or equal to the modulus that are relatively prime to the modulus. This upper bound is Euler's totient function of the modulus. Another method, the so-called lagged-Fibonacci generator, is also widely used. It has the form:

$$X_n = \left(X_{n-r}\ \mathrm{op}\ X_{n-p}\right) \bmod m. \tag{2}$$

The numbers r and p are called *lags* and there are methods for choosing them appropriately. The operator op can be one of the following binary operators: addition, subtraction, multiplication or exclusive or. However, it should be noted that from the point of view of hardware implementation, both congruential and lagged-Fibonacci random number generators are not very suitable: they are inefficient in terms of silicon area and time when applied to fine-grained massively parallel machines, for built-in self-test, or for other on-board applications. A third widespread type of generator is the so-called linear feedback shift register (LFSR). A pseudorandom sequence is generated by the linear recursion equation:

$$X_n = (c_1\,X_{n-1} + c_2\,X_{n-2} + \cdots + c_k\,X_{n-k}) \bmod 2. \tag{3}$$

Linear feedback shift registers are popular generators among physicists and computer engineers. There exist forms of LFSR that are suitable for hardware implementation. However, it turns out that when compared with equivalent CA-based generators, they are of lesser quality; furthermore, they are less favorable in terms of connectivity and delay, although the area needed by a CA cell is slightly larger than that of an LFSR cell. This is so because an LFSR with a large number of memory elements and feedback has an irregular interconnection structure, which makes it more difficult to modularize in VLSI. Moreover, different sequences generated by the same CA are much less correlated than the analogous sequences generated by an

https://doi.org/10.25088/ComplexSystems.26.4.373

LFSR. This means that CA-generated bit sequences can be used in parallel, which offers clear advantages in applications.

2.2 Cellular Automata for Random Number Generation

Cellular automaton-based random number generators evolve a state vector of zeros and ones according to a deterministic rule. For a given CA, an element (or cell) at a given position in the new state vector is determined by certain neighboring cells of that cell in the old state vector. A subset of cells in the state vectors is then output as random bits from which the pseudorandom numbers are generated. 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 [1], 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 [1]. The initially nonzero cell is the site at which the random temporal sequence is generated. Wolfram studied this particular rule extensively, demonstrating its suitability as a high-performance randomizer, which can be efficiently implemented in parallel; indeed, this CA is one of the standard generators of the massively parallel Connection Machine CM2 [2].

A nonuniform CA randomizer was presented in [3] (based on the work in [4]), 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 example application of such CA randomizers was demonstrated in [5], which presented the design of a low-cost, high-capacity associative memory.

An evolutionary approach for obtaining random number generators was taken in [6], which applied genetic programming to the evolution of a symbolic LISP expression that acts as a rule for a uniform CA (i.e., the expression is inserted into each CA cell, thereby comprising the function according to which the cell's next state is computed). It demonstrated evolved expressions that are equivalent to Wolfram's rule 30. The work in [1–6] leads us to ask whether good CA randomizers can be coevolved using CAs; the results reported in Section 5 suggest that.

3. Cellular Automata

3.1 An Informal Introduction

Cellular automata were originally conceived in the 1940s to provide a formal framework for investigating the behavior of complex extended systems [7]. Cellular automata are dynamical systems in which space and time are discrete. Cellular automaton systems are composed of adjacent cells or sites arranged as a regular lattice, which evolve in discrete time steps. Each cell is characterized by an internal state whose value belongs to a finite set. The updating of these states is made simultaneously, according to a common local transition rule involving a neighborhood of each cell. The state of a cell at the next time step is determined by the current states of cells in a surrounding neighborhood. The cellular array (grid) is d-dimensional, where $d = 1, 2, 3$ is used in practice; in this paper we concentrate on $d = 1$, that is, one-dimensional grids. The identical rule contained in each cell is essentially a finite-state machine, usually specified in the form of a rule table (also known as the transition function), with an entry for every possible neighborhood configuration of states. The cellular neighborhood of a cell consists of itself and the surrounding (adjacent) cells. For one-dimensional CAs, a cell is connected to r local neighbors (cells) on either side, where r is referred to as the radius (thus, each cell has $2r + 1$ neighbors). A common method of examining the behavior of one-dimensional CAs is to display a two-dimensional spacetime diagram, where the horizontal axis depicts the configuration at a certain time t and the vertical axis depicts successive time steps (e.g., Figure 1).

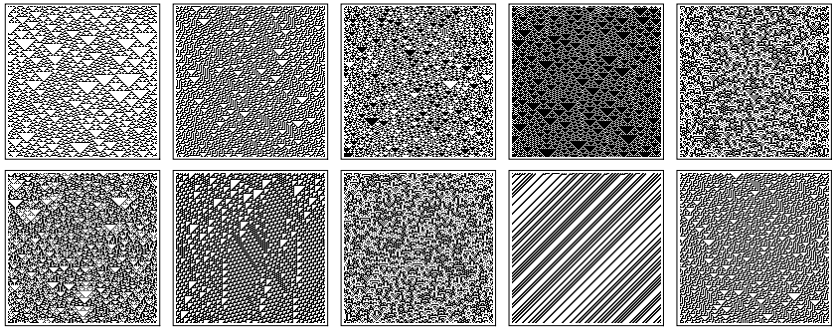


Figure 1. A spacetime diagram of patterns generated by simple one-dimensional CAs. The CAs consist of a row of about 200 sites whose values evolve with time down the page according to simple logical rules. The value 0 or 1 of each site (represented by white or black) is determined from its own value and the values of its two nearest neighbors on the step before. Patterns generated by 10 different rules are shown. In each case, the pattern is obtained with a random initial state. Despite the simplicity of these CAs, the patterns generated show considerable complexity.

https://doi.org/10.25088/ComplexSystems.26.4.373

The term "configuration" refers to an assignment of ones and zeros at a given time step (i.e., a horizontal line in the diagram). When the same rule is applied to update cells of CAs, such CAs are called uniform CAs, in contrast with nonuniform CAs, when different rules are assigned to cells and used to update them. Using one-dimensional, two-state CAs as a source of random bit sequences was first suggested by Wolfram. He used uniform, one-dimensional CAs with $r = 1$. In particular, he extensively studied rule 30. (Rule numbers are given in accordance with Wolfram's convention.) Nonuniform CAs with two rules, 90 and 150, were used in [3, 8] and it was found that the quality of generated pseudorandom number sequences (PNSs) was better than the quality of the Wolfram system. Proposed in [9] was the use of nonuniform, one-dimensional CAs with $r = 1$ and four rules, 90, 105, 150 and 165, which provide high-quality PNSs and a huge space of possible secret keys that are difficult for cryptanalysis. Instead, in order to design rules for CAs, an evolutionary technique called cellular programming (CP) was used to search for them. In this paper we continue this line of research. We will use finite, one-dimensional, nonuniform CAs. However, we extend the potential space of rules by consideration of one size of rule neighborhood, namely a neighborhood of radius $r = 1$. For example, in the case of rule 30 CA, in Boolean form it can be written as:

$$f(i,\ t+1) = f(i-1,\ t) \veebar \left(f(i,\ t) \vee f(i+1,\ t)\right) \tag{4}$$

where $f(i, t)$ is the value of cell i at time t. The formula gives the state of cell i at time step $t + 1$ as a Boolean function of the states of the neighboring cells at time t. Random bit sequences are obtained by sampling the values that a particular cell (usually the central one) attains as a function of time. In order to further decorrelate bit sequences, time spacing and site spacing are used. Time spacing means that not all the bits generated are considered as part of the random sequence. For instance, maybe only one bit is kept out of two, referred to as a time space value of 1, which means that sequences will be generated at half the maximal rate. In site spacing, only certain sites in a row are considered, where an integer number indicates how many sites are to be ignored between two successive cells. In practice, a site spacing of one or two is common, which means that half or two-thirds of the output bits are lost. Figure 2 demonstrates the workings of a rule 30 CA, both with and without time and site spacing. Figure 3 shows a sequence of totalistic CAs with three possible colors for each cell.

Over the years, CAs have been applied to the study of general phenomenological aspects of the world, including communication, computation, construction, growth, reproduction, competition and evolution (see, e.g., [10–15]). One of the most well-known CA rules,

the Game of Life, was conceived by Conway in the late 1960s [16, 17] and was shown by him to be computation universal [18]. For a review of computation-theoretic results, refer to [19]. Cellular automata also provide a useful model for a branch of dynamical systems theory that studies the emergence of well-characterized collective phenomena, such as ordering, turbulence, chaos, symmetry-breaking and fractality [20, 21].

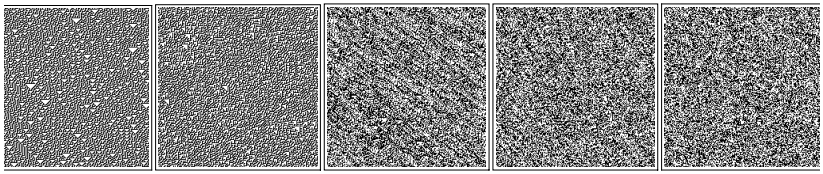


Figure 2. A spacetime diagram of CA rule 30. Grid size is $N = 200$; radius is $r = 1$. White squares represent cells in state 0; black squares represent cells in state 1. The pattern of configurations is shown through time (which increases down the page). The initial configurations were generated by randomly setting the state of each grid cell to 0 or 1 with uniform probability. (a) No time/site spacing; (b) site spacing = 1; (c) time spacing = 1; (d) time/site spacing = 1; (e) time/site spacing = 4.

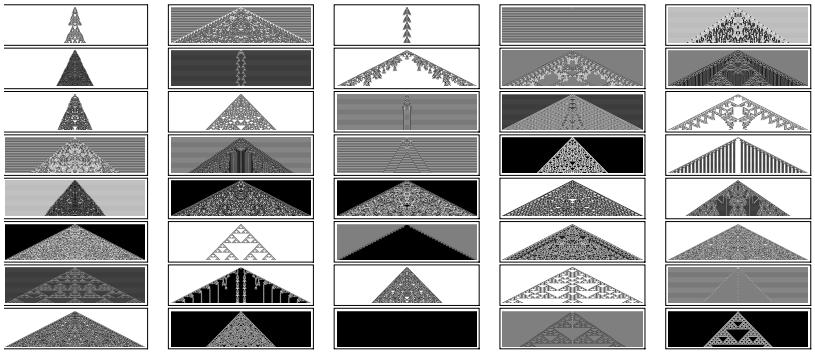


Figure 3. A sequence of totalistic CAs with three possible colors for each cell. Although their basic rules are more complicated, the CAs shown here do not seem to have fundamentally more complicated behavior than the two-color CAs shown in Figure 2. The symmetry of all the patterns is a consequence of the basic structure of totalistic rules.

The systematic study of CAs in this context was pioneered by Wolfram and studied extensively by him [22–24]. He investigated CAs and their relationships to dynamical systems, identifying the following four qualitative classes of behavior, with analogs in the field

https://doi.org/10.25088/ComplexSystems.26.4.373

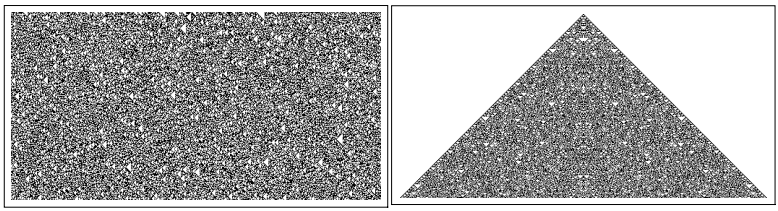
of dynamical systems (the latter are shown in parenthesis; see also [25, 26]:

- Class I relaxes to a homogeneous state (limit points).
- Class II converges to simple separated periodic structures (limit cycles).
- Class III yields chaotic aperiodic patterns (chaotic behavior of the kind associated with strange attractors).
- Class IV yields complex patterns of localized structures, including propagating structures (very long transients with no apparent analog in continuous dynamical systems).

3.2 Basic Structure of Nonuniform Cellular Automata

The basic model we employ in this paper is an extension of the original CA model, termed nonuniform CAs (Figure 4). 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. Note that nonuniform CAs share the basic "attractive" properties of uniform ones (massive parallelism, locality of cellular interactions, simplicity of cells). Nonuniform CAs were investigated in [27], which discusses a one-dimensional CA in which a cell probabilistically selects one of two rules at each time step. It showed that complex patterns appear characteristic of class IV behavior [27]. Two generalizations of CAs are presented in [28], namely, discrete neural networks and automata networks. These are compared to the original model from a computational point of view that considers the classes of problems such models can solve. In this section, we describe a prototype version of our model, in which the following feature is added to the elementary cellular automaton (ECA) presented in Section 3.1:

- A cell may contain a small number of different rules. At a given moment only one rule is active and determines the cell's function. An inactive rule may be activated or copied into a neighboring cell.



(a) (b)

Figure 4. The behavior of nonuniform CAs with two colors. In each case, 400 steps of evolution are shown. (a) Nonuniform CA with random initial configurations. (b) Shifted nonuniform CA, starting with a single black cell.

This feature could serve as a possible future enhancement in the evolutionary studies as well. At this point, we present a system involving the growth and replication of complex structures that are created from grid cells and behave as multicellular organisms once formed. The system consists initially of two cell types, builders and replicators, floating around on the grid.

- Historic memory can be embedded in the CA dynamics by endowing memory in cells without altering the mappings ϕ.

Conventional CAs are ahistoric (memoryless); that is, the new state of a cell depends on the neighborhood configuration solely at the preceding time step. Thus, if $\delta_i^{(T)}$ is taken to denote the value of cell i at time step T, the site values evolve by iteration of the mapping:

$$\delta^{(T+1)} = \phi\left(\left\{\delta_j^{(T)}\right\} \in \mathcal{N}_i\right) \tag{5}$$

where ϕ is an arbitrary function that specifies the CA rule operating on the cells in the neighborhood $\mathcal{N}$ of the cell i. The standard framework of CAs can be extended by implementing memory capabilities in cells:

$$\delta^{(T+1)} = \phi\left(\left\{S_j^{(T)}\right\} \in \mathcal{N}_i\right) \tag{6}$$

with $S_j^{(T)}$ being a state function of the series of states of the cell j up to time step T:

$$S_j^{(T)} = S\left(\delta_j^1, \ldots, \delta_j^{(T+1)}, \delta_j^{(T)}\right). \tag{7}$$

Thus in CAs with memory, while the mapping ϕ remains unaltered, the historic memory of all past iterations is retained by featuring each cell by a summary of its past states. That is to say, cells canalize memory to the map ϕ. The dynamics of ECA rules is dramatically altered when endowing cells with memory of the last steps, compared to the conventional CA paradigm that merely takes into account the last configuration. Particularly interesting is the effect of the parity rule acting as memory on rule 30 and on the linear rules 90 and 150, as it generates a seemingly random dynamic, even if it causes the system to fail in most of the randomness tests. Cellular automata with memory in cells can be considered as a natural and promising extension of the basic paradigm.

In this paper we study one-dimensional, 2-state, $r = 1$ nonuniform CAs, in which each cell may contain a different rule. Spatially periodic boundary conditions are used, resulting in a circular grid. Rather than employ a genetic algorithm or cellular programming approach, our algorithm (Figure 5) involves a single nonuniform CA of size $\mathbb{N}$. After initializing the states of each cell, the CA starts to evolve according to the assigned rule during a predefined number of time steps.

https://doi.org/10.25088/ComplexSystems.26.4.373

$\mathcal{V}1,\mathcal{V}2,\mathcal{V}3,\mathcal{V}4${null vector}
generate a random initial configuration for vector $\mathcal{V}1$ and $\mathcal{V}2$
$c = 0$ {initial configuration counter}
run CA on initial configuration for M time steps
while $c <$ M:
    for each cell i in $\mathcal{V}1$ and $\mathcal{V}2$ do
        compute weighted mean value m of previous states =
        $m(\sigma^{(T)}_{\mathcal{V}3_i}, \sigma^{(T)}_{\mathcal{V}4_i})$
        construct transition function = $f(m)$
        $\sigma^{(T+1)}_{\mathcal{V}3i} = \sigma^{(T)}_{\mathcal{V}1i}$
        $\sigma^{(T+1)}_{\mathcal{V}4i} = \sigma^{(T)}_{\mathcal{V}2i}$
        $\sigma^{(T+1)}_{\mathcal{V}1i} = \phi\left(f\left(\mathcal{S}^{(T)}_{i-1}\right), \mathcal{S}^{(T)}_{i}, f\left(\mathcal{S}^{(T)}_{i+1}\right)\right)$
        $\sigma^{(T+1)}_{\mathcal{V}2i} = \phi\left(\mathcal{S}^{(T)}_{i-1}, \mathcal{S}^{(T)}_{i}, \mathcal{S}^{(T)}_{i+1}\right)$
    $c = c + 1$

Figure 5. Pseudocode of the nonuniform CA algorithm.

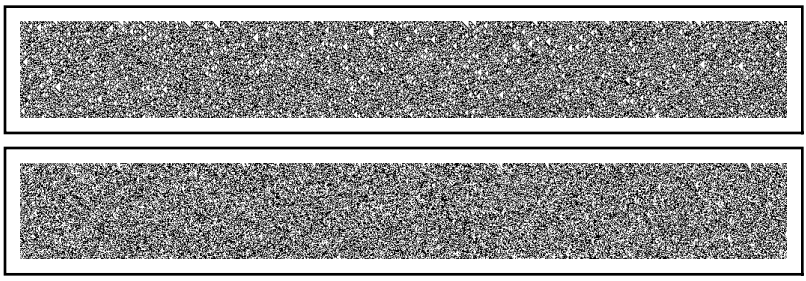


Figure 6. A spacetime diagram of nonuniform CAs; grid size is $\aleph = 128$, radius is $r = 1$. White squares represent cells in state 0; black squares represent cells in state 1. The initial configurations were generated by randomly setting the state of each grid cell to 0 or 1 with uniform probability (upper). No time/site spacing (lower). Site spacing of 1.

During the CA evolution, the preceding time steps are calculated and serve as a transition function applied locally to a given rule. The size and pattern of this transition function may differ from the neighborhood associated with types of rules. Once a state and transition vector is computed by evolving the CA by the specified neighbor rule, bits are selected for random numbers from bits {start,start+skip(site spacing),...}. In practice, using every second cell in each state vector proves to be sufficient to pass all stringent randomness tests. For even faster random number generation, a skip setting of 0 could be used, but the quality of the random numbers will then decline. The skip

option tied to a large state vector size is useful for setting up a family of independent generators that can be used in parallel computations. A typical result of a single run of this process starting with a random initial state (seed) is shown in Figure 6.

4. Statistical Testing of Random Number Generators and Historical Development

Over the years many statistical tests for testing random number generators have been proposed. One of the first collections was found in earlier editions of [29]. These tests, plus a few others designed for testing parallel generators, were implemented in SPRNG, a scalable library for pseudorandom number generation in [30]. New and more stringent tests, compared to the ones from [29], were introduced in [31]. Most of these tests were later implemented in Diehard, a battery of tests of randomness in [32], probably the best-known software package for testing random number generators. The National Institute of Standards and Technology (NIST), developed the NIST Statistical Test Suite [33] for the evaluation of the Advanced Encryption Standard (AES) candidate algorithms. The state-of-the-art library for testing random number generators today is TestU01, a C library for empirical testing of random number generators introduced in [34]. It implements:

1. A large variety of different random number generators proposed in the literature and/or used in software packages or operating systems.
2. Most of the statistical tests from Diehard, the NIST package, the Knuth collection, other tests found in the literature and some original ones.
3. Predefined test batteries.
4. Tools for investigating dependence of the period length of a generator within a whole family of random number generators and the length of a sequence when this generator begins to fail a given test systematically.

In essence, statistical testing of random number generators is nothing but a particular kind of Monte Carlo simulation. Conversely, when testing a random number generator for suitability with respect to a particular Monte Carlo problem, running the simulation with a related but simplified model, that is, one where the distribution of the result can be attained theoretically, may serve as a test. Even if the distribution is not known, the results of the designated random number generator can still be compared to the ones produced by a few other "good" generators of quite different designs.

Most statistical tests for random number generators utilize the concept of a p-value. P-values of single tests should not only be in the proper range (not too close to 0 or 1), but should also be uniformly

https://doi.org/10.25088/ComplexSystems.26.4.373

distributed on $[0, 1)$. Therefore, it might be useful to run the same test many times independently, that is, on different parts of the original sequence. Very often random number generators are tested against whole batteries of tests, and therefore p-values close to 0 or 1 are not too uncommon even for good (including perfect) generators (Table 1). If the final p-value of a test is really close to 0 or 1, the random number generator is said to fail the test. If the p-value is suspicious, the test is repeated and/or the sample size is increased, and often things will then clarify. Otherwise, the random number generator is said to have passed the test.

P-Value	Interpretation
$0.01 < p < 0.99$	Clear passed
p or $(1 - p) < 10^{-10}$	Clear failure

Table 1. Interpretation of p-values.

5. Analysis of the Test Results

In order to demonstrate the efficacy of a proposed random number generator, it is usually subject to a battery of empirical and theoretical tests. For the tests, we have used Diehard and TestU01. In order to apply the tests, we generated sequences of length L random bits using this procedure: the CA of radius $r = 1$, size $\mathbb{N} = 256$ and 1024, with site spacing $\mathrm{ss} = 1$ is run for t time steps, thus generating $t * (\mathbb{N} / (\mathrm{ss} + 1))$ random temporal bit sequences of length L. Table 2 shows the results of applying the test battery Crush from the random number generator software package TestU01 and the Diehard battery of tests to our novel CA-based model. We note that the nonuniform CAs attain very good results on all tests. Our results are somewhat better than rule 30, 90 and 105 CAs as PRNGs and also markedly improved in comparison to rule 30, which attains lower scores on the statistical tests. All these CAs failed the bitstream and OPSO tests. With respect to the OQSO test, rule 30 had always failed, while the other rules sometimes produced good (passing) strings. We conclude that on the whole, uniform CAs comprise fairly good generators, but they do not compare well with standard classical PRNG. In our simulations (using grids of sizes $\mathbb{N} = 256$ and 1024), we observed that high-performance architecture is attained as the grid size increases (computation time is linear with grid size). We have shown that the nonuniform CA algorithm can be applied to the difficult problem of generating random numbers. While a more extensive suite of tests is in order, it seems safe to say at this point these results are comparable

to the entropy values obtained in [6], as well as to those of the rule-30 CA of Wolfram [1] and the nonuniform CA rules {90, 150} of [3, 8]. Our results are also better than the CA rules {90, 150}, and markedly improved in comparison to the nonuniform CA randomizer in [9]. Furthermore, there is a notable advantage arising from the existence of a "tunable" algorithm for the generation of randomizers.

Generator Parameters	Test Package	Number of Statistics	Result
$\mathbb{N} = 256$, skip = 1	Diehard	126	All tests were passed
$\mathbb{N} = 1024$, skip = 1	Crush-TestU01	144	All tests were passed

Table 2. Results of the Diehard and test battery Crush (software package TestU01 (1.2.3)). (Note: skip or site spacing means an integer number indicates how many sites are to be ignored between two successive cells.)

6. Conclusion

This section highlights the contributions of this paper and some features by which it differs from the original cellular automaton (CA) model, organized in chronological order.

- Whereas the CA model consists of uniform cells, each containing the same rule, we consider the nonuniform case where different cells may contain different rules. A possible extension is the addition of restrictions to the nonuniform cellular automata (CAs), which have proven in the past more powerful than uniform ones, at no extra cost in terms of "software" or "hardware," while being faster to evolve and restricted to the scope of computation. In fact, it is easy to see that a uniform CA can simulate a nonuniform one by encoding all the different rules as one (huge) rule, employing a large number of states. One feature of our model, namely, the "active" nature of rules, whereby they may effect changes upon neighboring cells, may also be obviated by using "static" rules with larger neighborhoods, performing the equivalent operations. While these arguments hold true in principle, we argue that this is not so in practice. The power offered by our model cannot strictly be reduced to the question of computational power. Nonetheless, our investigations reported in the following sections do indeed show that our model holds potential for the exploration of CA phenomena.
- A novel developmental process of our system was presented. In this system, evolution takes place not only in state space as in the CA model, but also in rule space; rules may change (evolve) over time. A cell may contain a small number of different rules. At a given moment, only one rule is active and determines the cell's function. An inactive rule may be

activated or copied into a neighboring cell. At this point, we present a system involving the growth and replication of complex structures that are created from grid cells and behave as multicellular organisms once formed.

- A useful additional component is internal, finite memory. The dynamics of elementary rules is dramatically altered when endowing cells with memory of the previous time steps, compared to the conventional CA paradigm that merely takes into account the last configuration. Perhaps, as a result of a further full rigorous study of CAs with memory, it will be possible to paraphrase [35] in presenting CAs with memory as an alternative to (rather than an approximation of) integral equations in modeling, in particular, to Volterra integral equations that appear in the study of many phenomena incorporating memory, which are important in applied sciences such as population dynamics, diffusion, neural networks and so on.
- An analysis of pseudocode of the nonuniform CA algorithm. In this analysis, we showed a number of features of such a method, demonstrating the traps into which this algorithm may fall.
- In this paper, we have reported results of the study on using CAs as a basic form of a high-speed massively parallel computation engine. The main assumption of our approach was to consider nonuniform one-dimensional CAs. After we constructed a novel CA—demonstrated in Section 3.2—we verified the performance of a CA-based random number generator with the Diehard and TestU01 suites of statistical tests. Table 2 summarizes our findings, ranking all tested results according to the quality of the random numbers. Finally, an extensive suite of statistical tests and the results of experiments have shown that our nonuniform CA algorithm can be applied to the difficult problem of generating random numbers. Such CAs can be efficiently implemented in hardware and can be applied in the field of parallel computation. However, the main contribution is thus to have paved the path for future developments of similar systems.

References

[1] S. Wolfram, "Random Sequence Generation by Cellular Automata," *Advances in Applied Mathematics*, 7(2), 1986 pp. 123–169. doi:10.1016/0196-8858(86)90028-X.

[2] Connection, *The Connection Machine: CM-200 Series Technical Summary*, Cambridge, MA: Thinking Machines Corporation, 1991.

[3] P. D. Hortensius, R. D. McLeod and H. C. Card, "Parallel Random Number Generation for VLSI Systems Using Cellular Automata," *IEEE Transactions on Computers*, 38(10), 1989 pp. 1466–1473. doi:10.1109/12.35843.

[4] P. D. Hortensius, R. D. McLeod, W. Pries, D. M. Miller and H. C. Card. "Cellular Automata-Based Pseudorandom Number Generators for Built-In Self-Test," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **8**(8), 1989 pp. 842–859. doi:10.1109/43.31545.

[5] D. R. Chowdhury, I. S. Gupta and P. P. Chaudhuri, "A Low-Cost High-Capacity Associative Memory Design Using Cellular Automata," *IEEE Transactions on Computers*, **44**(10), 1995 pp. 1260–1264. doi:10.1109/12.467703.

[6] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge, MA: The MIT Press, 1992.

[7] J. von Neumann, *Theory of Self-Reproducing Automata* (A. W. Burks, ed.), Urbana, IL: University of Illinois Press, 1966.

[8] S. Nandi, B. K. Kar and P. P. Chaudhuri, "Theory and Applications of Cellular Automata in Cryptography," *IEEE Transactions on Computers*, **43**(12), 1994 pp. 1346–1357. doi:10.1109/12.338094.

[9] M. Tomassini and M. Perrenoud, "Stream Ciphers with One- and Two-Dimensional Cellular Automata," in *Parallel Problem Solving from Nature PPSN VI, LNCS VI (PPSN 2000)*, (M. Schoenauer et al., eds.), Berlin, Heidelberg: Springer, 2000 pp. 722–731 doi:10.1007/3-540-45356-3_ 71.

[10] A. W. Burks, ed., *Essays on Cellular Automata*, Urbana, IL: University of Illinois Press, 1970.

[11] A. R. Smith, *Cellular Automata Theory*, Technical Report 2, Digital Systems Laboratory, Stanford Electronics Laboratories, Stanford University, 1969.

[12] A. R. Smith, "Simple Computation-Universal Cellular Spaces," *Journal of the ACM*, **18**(3), 1971 pp. 339–353. doi:10.1145/321650.321652.

[13] A. R. Smith, "Simple Nontrivial Self-Reproducing Machines," in *Artificial Life II, Vol. X of SFI Studies in the Sciences of Complexity* (C. G. Langton, C. Taylor, J. D. Farmer and S. Rasmussen, eds.), Redwood City, CA: Addison-Wesley, 1992 pp. 709–725.

[14] T. Toffoli and N. Margolus, *Cellular Automata Machines: A New Environment for Modeling*, Cambridge, MA: MIT Press, 1987.

[15] [15] J.-Y. Perrier, M. Sipper and J. Zahnd, 1996. "Toward a Viable, Self-Reproducing Universal Computer," *Physica D: Nonlinear Phenomena*, **97**(4), 1996 pp. 335–352. doi:10.1016/0167-2789(96)00091-7.

[16] M. Gardner, "Mathematical Games: The Fantastic Combinations of John H. Conway's New Solitaire Game 'Life'," *Scientific American*, **223**, 1970 pp. 120–123.

[17] M. Gardner, "On Cellular Automata, Self-Replication, the Garden of Eden and the Game 'Life'," *Scientific American*, **224**(2), 1971 pp. 112–117.

[18] E. R. Berlekamp, J. H. Conway and R. K. Guy, *Winning Ways for Your Mathematical Plays,* New York: Academic Press, 1982 pp. 817–850.

[19] K. Culik II, L. P. Hurd and S. Yu, "Computation Theoretic Aspects of Cellular Automata," *Physica D*, **45**(1–3), 1990 pp. 357–378. doi:10.1016/0167-2789(90)90194-T.

[20] G. Y. Vichniac, "Simulating Physics with Cellular Automata," *Physica D: Nonlinear Phenomena*, **10**(1–2), 1984 pp. 96–116. doi:10.1016/0167-2789(84)90253-7.

[21] C. H. Bennett and G. Grinstein, "Role of Irreversibility in Stabilizing Complex and Nonergodic Behavior in Locally Interacting Discrete Systems," *Physical Review Letters*, **55**(7), 1985 pp. 657–660. doi:10.1103/PhysRevLett.55.657.

[22] S. Wolfram, "Statistical Mechanics of Cellular Automata," *Reviews of Modern Physics*, **55**(3), 1983 pp. 601–644. doi:10.1103/RevModPhys.55.601.

[23] S. Wolfram, "Cellular Automata as Models of Complexity," *Nature*, **311**, 1984 pp. 419–424. doi:10.1038/311419a0.

[24] S. Wolfram, "Universality and Complexity in Cellular Automata," *Physica D: Nonlinear Phenomena*, **10**(1–2), 1984 pp. 1–35. doi:10.1016/0167-2789(84)90245-8.

[25] C. G. Langton, "Studying Artificial Life with Cellular Automata," *Physica D: Nonlinear Phenomena*, **22**(1–3), 1986 pp. 120–149. doi:10.1016/0167-2789(86)90237-X.

[26] C. G. Langton, "Life at the Edge of Chaos," in *Artificial Life II, Vol. X of SFI Studies in the Sciences of Complexity* (C. G. Langton, C. Taylor, J. D. Farmer and S. Rasmussen, eds.), Redwood City, CA: Addison-Wesley, 1992 pp. 41–91.

[27] G. Y. Vichniac, P. Tamayo and H. Hartman, "Annealed and Quenched Inhomogeneous Cellular Automata," *Journal of Statistical Physics*, **45**(5–6), 1986 pp. 875–883. doi:10.1007/BF01020578.

[28] M. Garzon, "Cellular Automata and Discrete Neural Networks," *Physica D: Nonlinear Phenomena*, **45**(1–3), 1990 pp. 431–440. doi:10.1016/0167-2789(90)90200-9.

[29] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 2nd ed., Reading, MA: Addison-Wesley, 1981.

[30] M. Mascagni and A. Srinivasan, "Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation," *ACM Transactions on Mathematical Software*, **26**(3), 2000 pp. 436–461. doi:10.1145/358407.358427.

[31] G. Marsaglia, "A Current View of Random Number Generators," in *Computational Science and Statistics: Proceedings of the Sixteenth Symposium on the Interface*, Atlanta, Georgia, 1984 (L. Billard, ed.), New York: North-Holland, 1985 pp. 3–10.

[32] G. Marsaglia, "DIEHARD: Battery of Tests of Randomness," 1996.

[33] L. Bassham, A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, et al., "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications," *NIST Special Publication 800-22 Rev. 1a*, Gaithersburg, MD: National Institute of Standards and Technology, 2001. csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final.

[34] P. L'Ecuyer and R. Simard, "TestU01: A C Library for Empirical Testing of Random Number Generators," *ACM Transactions on Mathematical Software*, **33**(4), 2007 Article No. 22. doi:10.1145/1268776.1268777.

[35] T. Toffoli, "Cellular Automata as an Alternative to (Rather Than an Approximation of) Differential Equations in Modeling Physics," *Physica D: Nonlinear Phenomena*, **10**(1–2), 1984 pp. 117–127. doi:10.1016/0167-2789(84)90254-9.