International Journal of Modern Physics C, Vol. 7, No. 2 (1996) 181–190

GENERATING PARALLEL RANDOM NUMBER GENERATORS BY CELLULAR PROGRAMMING

MOSHE SIPPER
Logic Systems Laboratory, Swiss Federal Institute of Technology, IN-Ecublens, CH-1015 Lausanne, Switzerland. E-mail: Moshe.Sipper@di.epfl.ch

and

MARCO TOMASSINI
Logic Systems Laboratory and Swiss Scientific Computing Center, Manno. E-mail: Marco.Tomassini@di.epfl.ch

September 25, 1996

Random numbers are needed in a variety of applications, yet finding good random number generators is a difficult task. In this paper non-uniform cellular automata (CA) are studied, presenting the *cellular programming* algorithm for co-evolving such CAs to perform computations. The algorithm is applied to the evolution of random number generators; our results suggest that evolved generators are at least as good as previously described CAs, with notable advantages arising from the existence of a "tunable" algorithm for obtaining random number generators.

1. Introduction

Random numbers are needed in a variety of applications, yet finding good random number generators, or randomizers, is a difficult task [1]. 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 efficiency of a proposed generator, it is usually subjected to a battery of empirical and theoretical tests, among which the most well known are those described by Ref. [2].

In the last decade cellular automata (CA) have been used to generate random numbers. CAs are dynamical systems in which space and time are discrete; they consist 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 is determined by the previous states of a surrounding neighborhood of cells [3,4]. CAs exhibit three notable features, namely massive parallelism, locality of cellular interactions, and simplicity of basic components (cells), thus lending themselves naturally to fast, efficient hardware implementation. It has been shown that certain CAs can produce good random

sequences, i.e., act as random number generators (see Section 2).

The model investigated by us is an extension of the CA model, termed *non-uniform cellular automata* [5,6,7]; such automata function in the same way as the original uniform model, 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 local, co-evolutionary algorithm, an approach referred to as *cellular programming* [8]. In this paper we apply the cellular programming algorithm to the evolution of random number generators. Our results suggest that good randomizers can be evolved; these exhibit behavior at least as good as that of previously described CA generators, with notable advantages arising from the existence of a "tunable" algorithm for the generation of randomizers.

2. Previous work

In this section we survey previous work on randomizers and on evolving CAs. The first work examining the application of CAs to random number generation is that of Wolfram [9], in which rule 30 is extensively studied, demonstrating its ability to produce highly random, temporal bit sequences.[a] Such sequences are obtained by sampling the values that a particular cell (usually the central one) attains as a function of time. The cellular space under question is one-dimensional with $k = 2$ and $r = 1$, where k denotes the number of possible states per cell and r denotes the radius of a cell, i.e., the number of neighbors on either side (thus each cell has $2r + 1$ neighbors, including itself). A common method of examining the behavior of one-dimensional CAs is to display a two-dimensional space-time diagram, where the horizontal axis depicts the configuration at a certain time t and the vertical axis depicts successive time steps (e.g., Figure 2). The term 'configuration' refers to an assignment of 1 states to several cells, and 0s otherwise.

In Ref. [9], 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; the initially non-zero 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 [10]. A non-uniform CA randomizer was presented in Refs. [11,12], 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 has recently been demonstrated in Ref. [13], which presents the design of a low-cost, high-capacity associative memory.

The application of genetic algorithms [14,15,16] to the *evolution* of *uniform* cellular automata was initially studied in Ref. [17] and recently undertaken by the EVCA

[a] Rule numbers are given in accordance with Wolfram's convention, see Ref. [3]

(evolving CA) group [18,19,20,21,22], demonstrating that CAs can be evolved to perform non-trivial computational tasks, such as density and synchronization. The genetic algorithm employs a randomly generated initial population of uniform CAs with $k = 2$, $r = 3$. Each CA is represented by a bit string, delineating its rule table, containing the output bits for all possible neighborhood configurations (i.e., the bit at position 0 is the state to which neighborhood configuration 0000000 is mapped to and so on until bit 127 corresponding to neighborhood configuration 1111111). Each CA in the population is run for a maximum number of M time steps, after which its fitness is evaluated, defined as the fraction of cells in the correct state at the last time step.[b] Using the genetic algorithm highly successful CA rules were found for both the density and the synchronization tasks.

An evolutionary approach for obtaining random number generators was taken by Koza [23], who 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). 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 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 $E_h = h$. Koza evolved LISP expressions which act as rules for uniform, one-dimensional CAs. The CAs were run for 4096 time steps and the entropy of the resulting temporal sequence of a designated cell (usually the central one) was taken as the fitness of the particular rule (i.e., LISP expression). In his experiments Koza used a subsequence length of $h = 4$, obtaining rules with an entropy of 3.996. The best rule of each run was re-tested over 65536 time steps, some of which exhibited the maximal entropy value of 4.0.

The model investigated in this paper is that of *non-uniform* CAs, where cellular rules need not be identical for all cells. We have previously applied this model to the study of artificial life issues, presenting multi-cellular "organisms" that display several interesting behaviors, including reproduction, growth and mobility [7,24,25]. In Refs. [26,27] we demonstrated that universal computation can be attained in non-uniform, two-dimensional, 2-state, 5-neighbor CAs, which are not computation-universal in the uniform case. The universal systems we presented are simpler than

[b] The 'correct' state depends on the particular computational task for which the CA is being evolved.

previous ones and are *quasi*-uniform, meaning that the number of distinct rules is extremely small with respect to rule space size; furthermore, the rules are distributed such that a subset of dominant rules occupies most of the grid. The co-evolution of non-uniform, one-dimensional CAs to perform non-trivial computational tasks, such as density and synchronization, was undertaken in Refs. [8,27], where the cellular programming algorithm was presented; we showed that high performance, non-uniform CAs can be co-evolved not only with radius $r = 3$, as studied by Mitchell *et al.*, but also for smaller radiuses, most notably $r = 1$ which is minimal. It was also found that evolved systems exhibiting high performance are quasi-uniform.

The above account leads us to ask whether good CA randomizers can be co-evolved using cellular programming; the results reported below suggest that indeed this is the case.

3. Cellular programming

We study one-dimensional, 2-state, $r = 1$ non-uniform CAs, in which each cell may contain a different rule; spatially periodic boundary conditions are used, resulting in a circular grid. A cell's rule table is encoded as a bit string, known as the "genome", containing the output bits for all possible neighborhood configurations (see Section 2). Rather than employ a *population* of evolving, uniform CAs, as with genetic algorithm approaches, our algorithm involves a *single*, non-uniform CA of size N, with cell rules initialized at random. Initial configurations are then randomly generated and for each one the CA is run for $M = 4096$ time steps.[c] Each cell's *fitness*, f_i, is accumulated over $C = 300$ initial configurations, where a single run's score equals the entropy E_h of the temporal sequence of cell i. 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. After every C configurations evolution of rules occurs by applying the genetic operators of crossover and mutation in a completely *local* manner, 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 Figure 1. Crossover between two rules is performed by selecting at random (with uniform probability) a single crossover point and creating a new rule by combining the first rule's bit string before the crossover point with the second rule's bit string from this point onward. Mutation is applied to the bit string of a rule with probability 0.001 per bit.

There are two main differences between our evolutionary algorithm and a standard genetic algorithm: (a) A standard genetic algorithm involves a population of evolving, uniform CAs; all CAs are *ranked* according to fitness, with crossover occurring between *any* two CA rules. 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 per-

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

```
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
        f_i = f_i + entropy E_h of the temporal sequence of cell i
    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 the cellular radius, r, > 1)
            end if
            f_i = 0
        end parallel for
    end if
end while
```

Fig. 1. Pseudo-code of the cellular programming algorithm

formed. (b) The standard genetic algorithm involves a population of *independent* problem solutions; each CA is run independently, after which genetic operators are applied to produce a new population. In contrast, our CA *co-evolves* since each cell's fitness depends upon its evolving neighbors.

4. Results

In this section we describe results of applying the cellular programming algorithm to the evolution of random number generators. 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 $M = 65536$ times steps (as in Ref. [23]), 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 in Ref. [23] as well as to those of the rule 30 CA of Ref. [9] and the non-uniform, rules $\{90, 150\}$ CA of Refs. [11,12]. 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. Figure 2 demonstrates the operation of three CAs discussed above: rule 30, rules $\{90, 150\}$, and a co-evolved CA. Note that the latter is quasi-uniform (Section 2), as evident by observing the rules map; this map depicts the distribution of rules by assigning a unique color to each distinct rule.

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 in Ref. [2]. 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 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:[d] two co-evolved CAs, rule 30 CA, and the rules $\{90, 150\}$ CA. We note that the two co-evolved CAs attain good results on all tests, most notably chi-square which is one of the most significant ones [2]. 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 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" [11,12]. Our final experiments involve the implementation of a simple scaling scheme in which a $N = 500$ CA is created from the evolved $N = 50$ one. This is done in a straightforward manner by duplicating the evolved CA 10 times, i.e., concatenating 10 identical copies of the 50-cell rules grid. Table 2 shows that good randomizers can thus be obtained, demonstrating possible scalability of our system.

[d] The tests were conducted using a public domain software written by J. Walker, available at `http://www.fourmilab.ch/random/`

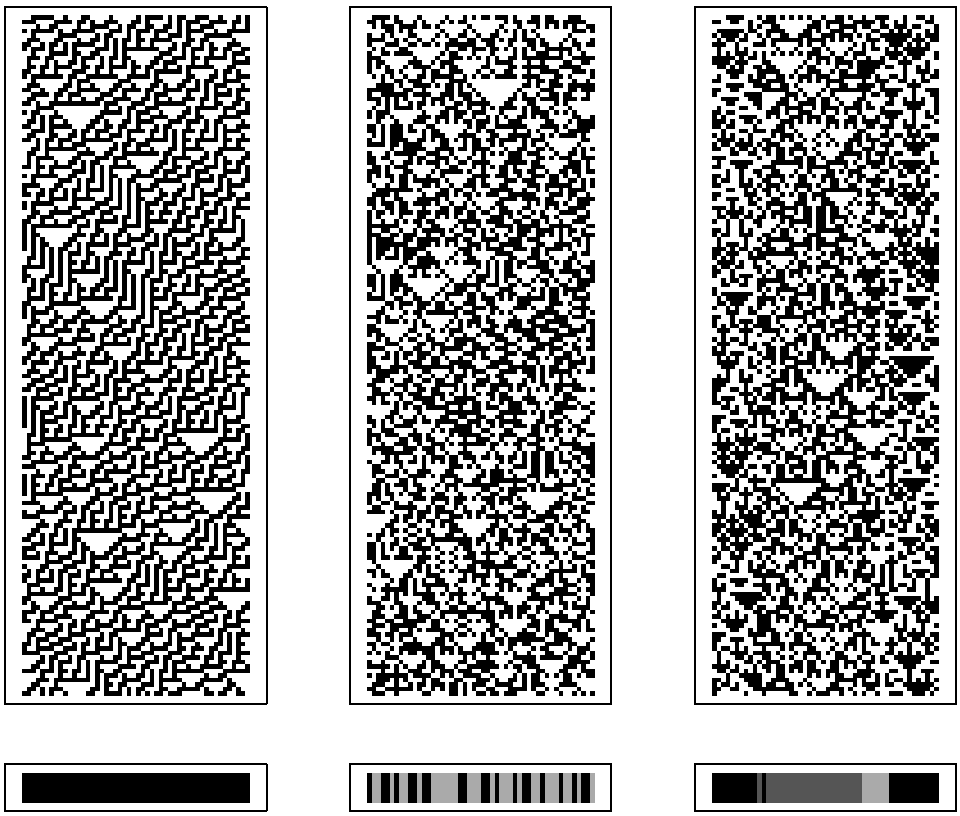

Fig. 2. One-dimensional random number generators: Operation of three CAs. Grid size is $N = 50$, radius is $r = 1$. White squares represent cells in state 0, black squares represent cells in state 1. The pattern of configurations is shown through time (which increases down the page); the initial configurations were generated by randomly setting the state of each grid cell to 0 or 1 with uniform probability. Top figures depict space-time diagrams, bottom figures depict rule maps. (a) Rule 30 CA. (b) Rules $\{90, 150\}$ CA. (c) A co-evolved, non-uniform CA, consisting of three rules: rule 165 (22 cells), rule 90 (22 cells), rule 150 (6 cells).

5. Concluding remarks

We presented the cellular programming algorithm for co-evolving non-uniform CAs and applied it to the problem of generating random number generators. While a more extensive suite of tests should be conducted, it seems safe to say at this point that our co-evolved generators are at least as good as the best available CA randomizers.

We observed that our evolved CAs are quasi-uniform, involving a small number of distinct rules; 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 would be the addition of restrictions to the evolutionary process, e.g., by pre-specifying 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 hold potential for studying phenomena of interest in areas such as complex systems, artificial life and parallel computation. This work has shed light on the possibility of using such CAs as random number generators, and demonstrated the feasibility of their evolution.

References

1. S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988.
2. D. E. Knuth. *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*. Addison-Wesley, Reading, MA, second edition, 1981.
3. S. Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, 55(3):601–644, July 1983.
4. T. Toffoli and N. Margolus. *Cellular Automata Machines*. The MIT Press, Cambridge, Massachusetts, 1987.
5. G. Y. Vichniac, P. Tamayo, and H. Hartman. Annealed and quenched inhomogeneous cellular automata. *Journal of Statistical Physics*, 45:875–883, 1986.
6. H. Hartman and G. Y. Vichniac. Inhomogeneous cellular automata. In E. Bienenstock, F. Fogelman, and G. Weisbuch, editors, *Disordered Systems and Biological Organization*, pages 53–57. Springer-Verlag, Berlin, 1986.
7. M. Sipper. Non-uniform cellular automata: Evolution in rule space and formation of complex structures. In R. A. Brooks and P. Maes, editors, *Artificial Life IV*, pages 394–399, Cambridge, Massachusetts, 1994. The MIT Press.
8. M. Sipper. Co-evolving non-uniform cellular automata to perform computations. *Physica D*, 92:193–208, 1996.
9. S. Wolfram. Random sequence generation by cellular automata. *Advances in Applied Mathematics*, 7:123–169, June 1986.
10. Thinking Machines Corporation, Cambridge, Massachusetts. *The Connection Machine: CM-200 Series Technical Summary*, June 1991.
11. 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):1466–1473, October 1989.
12. 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*, 8(8):842–859, August 1989.
13. 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):1260–1264, October 1995.
14. M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.
15. M. Tomassini. A survey of genetic algorithms. In D. Stauffer, editor, *Annual Reviews of Computational Physics*, volume III, pages 87–118. World Scientific, 1995. Also available as: Technical Report 95/137, Department of Computer Science, Swiss Federal Institute of Technology, Lausanne, Switzerland, July, 1995.
16. D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.
17. N. H. Packard. Adaptation toward the edge of chaos. In J. A. S. Kelso, A. J. Mandell, and M. F. Shlesinger, editors, *Dynamic Patterns in Complex Systems*, pages 293–301. World Scientific, Singapore, 1988.
18. M. Mitchell, P. T. Hraber, and J. P. Crutchfield. Revisiting the edge of chaos: Evolving cellular automata to perform computations. *Complex Systems*, 7:89–130, 1993.
19. M. Mitchell, J. P. Crutchfield, and P. T. Hraber. Evolving cellular automata to perform computations: Mechanisms and impediments. *Physica D*, 75:361–391, 1994.
20. M. Mitchell, J. P. Crutchfield, and P. T. Hraber. Dynamics, computation, and the "edge of chaos": A re-examination. In G. Cowan, D. Pines, and D. Melzner, editors, *Complexity: Metaphors, Models and Reality*, pages 491–513. Addison-Wesley, Reading, MA, 1994.
21. R. Das, M. Mitchell, and J. P. Crutchfield. A genetic algorithm discovers particle-based computation in cellular automata. In Y. Davidor, H. -P. Schwefel, and R. Männer, editors, *Parallel Problem Solving from Nature- PPSN III*, volume 866 of *Lecture Notes in Computer Science*, pages 344–353, Berlin, 1994. Springer-Verlag.
22. R. Das, J. P. Crutchfield, M. Mitchell, and J. E. Hanson. Evolving globally synchronized cellular automata. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 336–343, San Francisco, CA, 1995. Morgan Kaufmann.
23. J. R. Koza. *Genetic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.
24. M. Sipper. Studying artificial life using a simple, general cellular model. *Artificial Life Journal*, 2(1):1–35, 1995. The MIT Press, Cambridge, MA.
25. M. Sipper. An introduction to artificial life. *Explorations in Artificial Life (special issue of AI Expert)*, pages 4–8, September 1995. Miller Freeman, San Francisco, CA.
26. M. Sipper. Quasi-uniform computation-universal cellular automata. In F. Morán, A. Moreno, J. J. Merelo, and P. Chacón, editors, *ECAL'95: Third European Conference on Artificial Life*, volume 929 of *Lecture Notes in Computer Science*, pages 544–554, Berlin, 1995. Springer-Verlag.
27. M. Sipper. Complex computation in non-uniform cellular automata, 1996. (submitted).

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 $N = 50$. Co-evolved CA (1) consists of three rules: rule 165 (22 cells), rule 90 (22 cells), and rule 150 (6 cells); co-evolved 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 Ref. 2): (i) For the 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 percentage of sequences passing the chi-square test is also listed in the table. Knuth suggests that at least three sequences from a generator be subjected to the chi-square test and if a majority pass then the generator is considered to have passed (with respect to chi-square) (ii) The serial correlation coefficient should be close to zero. (iii) Entropy should be close to 8. (iv) 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.

	co-evolved CA (1)		co-evolved CA (2)		rule 30 CA		rules {90,150} CA	
Test	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
Chi-square	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%
	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%
(% success)	100%	100%	90%	90%	50%	90%	70%	100%
Serial correlation coefficient	0.001849	-0.000848	-0.003904	0.019516	0.000523	-0.246845	0.006464	0.000362
	-0.003855	-0.002284	0.002275	0.021441	-0.001752	-0.248375	-0.000714	-0.001935
	0.001923	-0.002973	0.000480	0.019697	0.001561	-0.242909	0.002054	-0.003217
	-0.000111	-0.002484	-0.002366	0.021919	0.004775	-0.237349	0.001768	0.000940
	-0.000596	-0.007619	0.001942	0.019370	0.002137	-0.241494	-0.000751	0.003778
Entropy	7.998185	7.998277	7.998074	7.998269	7.998406	7.998421	7.998209	7.997965
	7.998214	7.998173	7.998352	7.998098	7.997887	7.998197	7.997877	7.998066
	7.998376	7.998094	7.998446	7.997860	7.998487	7.997697	7.997928	7.998087
	7.997998	7.998312	7.998060	7.998082	7.997329	7.998069	7.998318	7.998038
	7.998079	7.998011	7.998293	7.998079	7.998444	7.998349	7.998507	7.997999
Monte Carlo π	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%
	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%

Table 2. Chi-square test results of scaled, $N = 500$ CAs. These were created from the corresponding co-evolved, $N = 50$ CAs by duplicating the evolved grid ten times. 20 random sequences were generated by each CA, 10 by procedure (a) and 10 by procedure (b); chi-square test results are shown, along with the percentage of sequences passing the test. The other tests were also conducted, obtaining similar results to the original, non-scaled CAs.

		% success	Chi-square test results									
scaled	(a)	100%	75.00%	50.00%	25.00%	25.00%	25.00%	75.00%	50.00%	50.00%	90.00%	50.00%
CA (1)	(b)	90%	75.00%	50.00%	50.00%	97.50%	50.00%	75.00%	50.00%	50.00%	75.00%	50.00%
scaled	(a)	80%	50.00%	50.00%	25.00%	50.00%	2.50%	50.00%	95.00%	50.00%	50.00%	50.00%
CA (2)	(b)	90%	50.00%	50.00%	50.00%	75.00%	25.00%	50.00%	50.00%	25.00%	5.00%	10.00%