D. Stauffer, editor, *Annual Reviews of Computational Physics*, volume V,
pages 243-285, 1997.

Evolving Uniform and Non-uniform Cellular Automata Networks

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

Natural evolution has "created" many parallel cellular systems, in which emergent computation gives rise to impressive computational capabilities. In recent years we are witness to a rapidly growing interest in such *complex adaptive systems*, addressing, among others, the major problem of designing them to exhibit a specific behavior or solve a given problem. One possible approach, which we explore in this paper, is to employ artificial evolution. The systems studied are based on the cellular automata (CA) model, where a regular grid of cells is updated synchronously in discrete time steps, according to a local, identical interaction rule. We first present the application of a standard genetic algorithm to the evolution of CAs to perform two non-trivial computational tasks, density and synchronization, showing that high-performance systems can be attained. The evolutionary process as well as the resulting emergent computation are then discussed. Next we study two generalizations of the CA model, the first consisting of non-uniform CAs, where cellular rules need not be identical for all cells. Introducing the *cellular programming* evolutionary algorithm we apply it to a complex computational tasks, demonstrating that high-performance systems can be evolved. The second generalization involves non-standard, evolving connectivity architectures, where we demonstrate that yet better systems can be obtained. Evolving, cellular systems hold potential both scientifically, as vehicles for studying phenomena of interest in areas such as complex adaptive systems and artificial life, as well as practically, showing a range of potential future applications ensuing the construction of adaptive systems, and in particular 'evolvable hardware', *evolware*.

1 Introduction

Natural evolution has "created" many systems in which the actions of simple, locally-interacting components give rise to coordinated global information processing. Insect colonies, cellular assemblies, the retina, and the immune system, have all been cited as examples of systems in which *emergent computation* occurs. This term refers to the appearance of global information processing capabilities that are not explicitly represented in the system's elementary components nor in their interconnections.

The parallel cellular systems "designed" by Nature display an impressive capacity to successfully confront extremely difficult computational problems. In recent years we are witness to a rapidly growing interest in such *complex adaptive systems*;[1–3] under this heading we find researchers from different

fields, studying diverse systems, natural- as well as human-made, with the underlying two-fold goal of: (1) enhancing our understanding of the functionings of natural systems, as well as of the ways by which they might have evolved, and (2) mimicking Nature's achievement, creating artificial systems based on these principles, matching the problem-solving capacities of their natural counterparts.

In this paper we shall attempt to gain insight into these issues, using the well-known cellular automata (CA) model. CAs are dynamical systems in which space and time are discrete; a cellular automaton consists of a regular grid 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. CAs exhibit three notable features, namely massive parallelism, locality of cellular interactions, and simplicity of basic components (cells); thus, they present an excellent point of departure for our forays into parallel cellular systems. We shall study the original, classical model as well as a number of generalized ones; these cellular automata networks are referred to henceforth as *cellular systems*.

A major problem common to such local, parallel systems is the painstaking task one is faced with in designing them to exhibit a specific behavior or solve a particular problem. This results from the local dynamics of the system, which renders the design of local interaction rules to perform global computational tasks extremely arduous. Toward this end we turn to Nature, seeking inspiration in the process of evolution. The idea of applying the biological principle of natural evolution to artificial systems, introduced more than three decades ago, has seen an impressive growth in the past few years. Usually grouped under the term *evolutionary algorithms* or *evolutionary computation*, we find the domains of genetic algorithms, evolution strategies, evolutionary programming, and genetic programming. Central to all these different methodologies is the idea of solving problems by evolving an initially random population of possible solutions, through the application of "genetic" operators, such that in time "fitter" (i.e., better) solutions emerge.[4–11] We shall employ artificial evolution, based on the genetic algorithms approach, to evolve ("design") parallel cellular systems.

As pointed out in Ref. 12, it is important to distinguish between emergent computation in natural systems, and what is often called "emergent" or "self-organizing" pattern formation, as found in weak turbulence, critical phenomena, oscillating chemical reactions, and developmental morphogenesis. They argued that the emergent patterns in such systems are subjective since the existence of a pattern is decided only by an outside observer; emergent computation also involves emergent patterns, but these are *used by the sys-*

tem itself to perform information processing.

Though the results described herein have been obtained through software simulation, one of the major goals is to attain truly 'evolving ware', *evolware*, with current implementations centering on hardware, while raising the possibility of using other forms of ware in the future, such as *bioware*.[13,14] This idea, whose origins can be traced to the cybernetics movement of the 1940s and the 1950s, has recently resurged in the form of the nascent field of bio-inspired systems and evolvable hardware.[15] The field draws on ideas from the evolutionary computation domain as well as on recent hardware developments. We have recently implemented an evolving, on-line, autonomous hardware system based on the cellular programming approach described in Section 5.[14]

Employing parallel cellular models, along with an evolutionary process that acts as an idealized version of natural evolution, enables us to attain progress in the pursuit of both aforementioned goals; this framework allows us to investigate issues pertaining to evolving, emergent computation in natural systems, as well as to derive methods by which successful, artificial systems may be constructed.

The paper is organized as follows: the next two sections present introductory expositions of cellular automata and genetic algorithms. In Section 4 we present the application of a genetic algorithm to evolve cellular automata to perform two non-trivial computational tasks, density and synchronization. It is shown that high-performance systems can be evolved for both tasks, after which the evolutionary process as well as the resulting emergent computation are discussed. In Section 5 we study a generalization of the original CA model, *non-uniform* CAs, where cellular rules need not be identical for all cells. Introducing the *cellular programming* algorithm for co-evolving such CAs, we apply it to the above two tasks, as well as to several others, demonstrating that high-performance systems can be attained. As opposed to the standard genetic algorithm, where a population of *independent* problem solutions *globally* evolves, cellular programming involves a grid of rules that *co-evolves locally*. In Section 6 we generalize on a second aspect of CAs, namely their standard, homogeneous connectivity. We study *non-standard* architectures, where each cell has a small, identical number of connections, yet not necessarily from its most immediate neighboring cells. We show that such architectures are computationally more efficient than standard architectures in solving global tasks; furthermore, it is shown that one can successfully *evolve* non-standard architectures through a two-level evolutionary process, in which the cellular rules evolve concomitantly with the cellular connections. Finally, we present our conclusions and directions for future research in Section 7.

2 Cellular automata

Cellular automata were originally conceived by Ulam and von Neumann in the 1940s to provide a formal framework for investigating the behavior of complex, extended systems.[16] CAs are dynamical systems in which space and time are discrete. A cellular automaton consists of a regular grid of cells, each of which can be in one of a finite number of k 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.[17,18]

The infinite or finite cellular array (grid) is n-dimensional, where $n = 1, 2, 3$ is used in practice; in this work we shall concentrate on $n = 1, 2$, i.e., one- and two-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 *neighborhood* of a cell consists of the surrounding (adjacent) cells; for one-dimensional CAs, a cell is connected to r local neighbors (cells) on either side, where r is a parameter referred to as the *radius* (thus, each cell has $2r + 1$ neighbors, including itself). For two-dimensional CAs, two types of cellular neighborhoods are usually considered: 5 cells, consisting of the cell along with its four immediate nondiagonal neighbors, and 9 cells, consisting of the cell along with its eight surrounding neighbors. The term *configuration* refers to an assignment of states to cells in the grid. When considering a finite-sized grid, spatially periodic boundary conditions are frequently applied, resulting in a circular grid for the one-dimensional case, and a toroidal one for the two-dimensional case. A one-dimensional CA is illustrated in Figure 1 (based on Ref. 6).

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., Refs. 18, 19, 20, 21). One of the most well-known CA rules, namely the "game of life", was conceived by Conway in the late 1960s[22,23] and was shown by him to be computation-universal.[24] A review of computation theoretic results is provided in Ref. 25.

The question of whether cellular automata can model not only general phenomenological aspects of our world, but also directly model the laws of physics themselves was raised in Refs. 26, 27. A primary theme of this research is the formulation of computational models of physics that are *information-preserving*, and thus retain one of the most fundamental features of microscopic physics, namely *reversibility*.[27–29] CAs have been used to provide extremely

Rule table:

neighborhood:	111	110	101	100	011	010	001	000
output bit:	1	1	1	0	1	0	0	0

Grid:

$t = 0$	0	1	1	0	1	0	1	1	0	1	1	0	0	1	1
$t = 1$	1	1	1	1	0	1	1	1	1	1	1	0	0	1	1

Figure 1: Illustration of a one-dimensional, 2-state CA. The connectivity radius is $r = 1$, meaning that each cell has two neighbors, one to its immediate left and one to its immediate right. Grid size is $N = 15$. The rule table for updating the grid is shown on top. The grid configuration over one time step is shown at the bottom. Spatially periodic boundary conditions are applied, meaning that the grid is viewed as a circle, with the leftmost and rightmost cells each acting as the other's neighbor.

simple models of common differential equations of physics, such as the heat and wave equations[30] and the Navier-Stokes equation.[31,32] CAs also provide a useful discrete model for a branch of dynamical systems theory which studies the emergence of well-characterized collective phenomena such as ordering, turbulence, chaos, symmetry-breaking, fractality, etc'.[33,34] The systematic study of CAs in this context was pioneered by Wolfram and studied extensively by him, identifying four qualitative classes of CA behavior (referred to as Wolfram classes), with analogs in the field of dynamical systems.[17,35,36] Finally, biological modeling has also been carried out using CAs (see, e.g., review in Ref. 37).

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.[4–11] In this paper we shall concentrate on one type of evolutionary algorithms, namely *genetic algorithms*.

Genetic algorithms were invented by John Holland in the 1960s.[11] His orig-

inal goal was not to design algorithms to solve specific problems, but rather to formally study the phenomenon of adaptation as it occurs in Nature and to develop ways in which the mechanisms of natural adaptation might be imported into computer systems. Nowadays, genetic algorithms are ubiquitous, having been successfully applied to numerous problems from different domains, including optimization, automatic programming, machine learning, economics, immune systems, ecology, population genetics, studies of evolution and learning, and social systems.[6] For recent reviews of the current state of the art, the reader is referred to Refs. 38, 39.

A genetic algorithm is an iterative procedure that consists of 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 symbol alphabet used is often binary due to certain computational advantages purported in Ref. 11 (see also Ref. 10); this has been extended in recent years to include character-based encodings, real-valued encodings, and tree representations.

The standard genetic algorithm proceeds as follows: an initial population of individuals is generated at random or heuristically. 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 Holland's original *fitness-proportionate selection*, where individuals are selected with a probability proportional to their relative fitness. 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_{cross} (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 enables 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_{mut}. 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 or as the attainment of an acceptable fitness level. Figure 2 presents the standard genetic algorithm in pseudo-code format.

```
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
```

Figure 2: Pseudo-code of the standard genetic algorithm.

Let us consider the following simple example, due to Ref. 6, demonstrating the genetic algorithm's workings. The population consists of 4 individuals, which are binary-encoded strings (genomes) of length 8; the fitness value equals the number of ones in the bit string, with $p_{cross} = 0.7$, and $p_{mut} = 0.001$. More typical values of the population size and the genome length are in the range 50-1000; also note that fitness computation in this case is extremely simple since no complex decoding nor evaluation is necessary. The initial (randomly generated) population might look like this:

Label	Genome	Fitness
A	00000110	2
B	11101110	6
C	00100000	1
D	00110100	3

Using fitness-proportionate selection we must choose 4 individuals (two sets of parents), with probabilities proportional to their relative fitness values.

In our example, suppose that the two parent pairs are {B,D} and {B,C} (note that A did not get selected as our procedure is probabilistic). Once a pair of parents is selected, with probability p_{cross} they cross over to form two offspring; if they do not cross over, then the offspring are exact copies of each parent. Suppose in our example that parents B and D cross over after the (randomly chosen) first bit position to form offspring E=10110100 and F=01101110, and parents B and C do not cross over, forming offspring that are exact copies of B and C. Next, each offspring is subject to mutation with probability p_{mut} per bit. For example, suppose offspring E is mutated at the sixth position to form E′=10110000, offspring F and C are not mutated at all, and offspring B is mutated at the first bit position to form B′=01101110. The next generation population, created by the above operators of selection, crossover, and mutation is therefore:

Label	Genome	Fitness
E′	10110000	3
F	01101110	5
C	00100000	1
B′	01101110	5

Note that in the new population, although the best individual with fitness 6 has been lost, the *average fitness* has increased. Iterating this procedure, the genetic algorithm will eventually find a perfect string, i.e., with maximal fitness value of 8.

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 possibly 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 in Refs. 38, 40); 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. In Section 5 we shall study non-uniform CAs, introducing the *cellular programming* algorithm, whose locality renders it more amenable to parallel implementation and to the construction of evolware.

4 Evolving uniform cellular automata

A major impediment preventing ubiquitous computing with CAs stems from the difficulty of utilizing their complex behavior to perform useful computations. As noted in Ref. 41, the difficulty of designing CAs to have a specific

behavior or perform a particular task has severely limited their applications; automating the design (programming) process would greatly enhance the viability of CAs. One possible approach is to utilize artificial evolution techniques.

The application of genetic algorithms to the evolution of *uniform* cellular automata was initially studied in Ref. 42 and recently undertaken by the EVCA (evolving CA) group.[12,41,43–46] The goals of their research, as stated in Ref. 41, are: (1) to better understand the ways in which CAs can perform computations; (2) to learn how best to use genetic algorithms to evolve computationally useful CAs; and (3) to understand the mechanisms by which evolution, as modeled by a genetic algorithm, can create complex, coordinated global behavior in a system consisting of many locally interacting simple parts.

Typically, a CA performing a computation means that the input to the computation is encoded as an initial configuration, the output is the configuration after a certain number of time steps, and the intermediate steps that transform the input to the output are considered to be the steps in the computation.[6] The "program" emerges through "execution" of the CA rule in each cell. Note that this use of CAs as computers differs from the less practical though theoretically interesting method of constructing a universal Turing machine[47] in a CA (for a comparison of these two approaches see Ref. 44; see also Ref. 21).

The EVCA group employed a standard genetic algorithm, as outlined in Section 3, to evolve uniform CAs to perform two computational tasks, namely density and synchronization; their results are described in the next two subsections. The CAs in question are one-dimensional with $k = 2$ and $r = 3$, i.e., 2 possible states per cell, with each cell connected to its three left neighbors and its three right ones (refer to Section 2). Spatially periodic boundary conditions are used, resulting in a circular grid. 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 3).

4.1 The density task

The one-dimensional density task is to decide whether or not the initial configuration contains more than 50% 1s. Following Ref. 43, let ρ denote the density of 1s in a grid configuration, $\rho(t)$ the density at time t, and ρ_c the threshold density for classification (in our case 0.5). The desired behavior (i.e., the result of the computation) is for the CA to relax to a fixed-point pattern of all 1s if $\rho(t = 0) > \rho_c$, and all 0s if $\rho(0) < \rho_c$. If $\rho(0) = \rho_c$, the desired behavior is undefined (this situation shall be avoided by using odd grid sizes).

This task is an example of "useful computation" as characterized above; the rule table applied to all cells is interpreted as a program performing a computation, the initial configuration is interpreted as the input to the program, and the CA runs for some specified number M of time steps or until it converges to one of the two fixed-point patterns. The final configuration is interpreted as the output.[41]

Designing an algorithm to perform the density task is trivial for systems with a central controller of some kind, such as a standard computer with a counter register, or a neural network with global connectivity; however, it is difficult to do so using a decentralized, spatially-extended system such as a CA.[12] As noted in Ref. 41, the density task comprises a non-trivial computation for a small radius CA ($r \ll N$, where N is the grid size); the 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. Note that the density task cannot be perfectly solved by a uniform, two-state CA, as proven in Ref. 48; however, no upper bound is currently available on the best possible imperfect performance.

A $k = 2$, $r = 3$ rule which successfully performs this task was discussed in Ref. 42; this is the Gacs-Kurdyumov-Levin (GKL) rule, defined as follows:[49,50]

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

where $s_i(t)$ is the state of cell i at time t.

Figure 3 depicts the behavior of the GKL rule on two initial configurations, $\rho(0) < \rho_c$ and $\rho(0) > \rho_c$. We observe that a transfer of information about local neighborhoods takes place to produce the final fixed-point configuration. Essentially, the rule's "strategy" is to successively classify local densities with the locality range increasing over time. In regions of ambiguity a "signal" is propagated, seen either as a checkerboard pattern in space-time or as a vertical white-to-black boundary.[43] It should be emphasized that the GKL's success on the density task is a serendipitous effect since it was not invented for the purpose of performing any particular computational task.[41]

The standard genetic algorithm employed in Refs. 41, 43 uses a randomly generated initial population of uniform, size $N = 149$ CAs with $k = 2$, $r = 3$. Each CA is represented by a bit string, delineating its rule table, containing the next-state (output) bits for all possible neighborhood configurations, listed in lexicographic order (i.e., the bit at position 0 is the state to which neighborhood

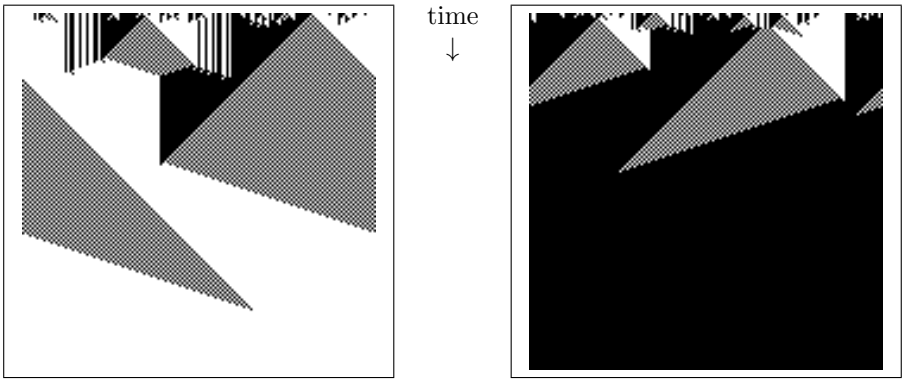


Figure 3: The density task: Operation of the GKL rule. CA is one-dimensional, uniform, 2-state, with connectivity radius $r = 3$. Grid size is $N = 149$. White squares represent cells in state 0, black squares represent cells in state 1. The pattern of configurations is shown through time (which increases down the page). (a) Initial density of 1s is $\rho(0) \approx 0.47$ and final density at time $t = 150$ is $\rho(150) = 0$. (b) Initial density of 1s is $\rho(0) \approx 0.53$ and final density at time $t = 150$ is $\rho(150) = 1$. The CA relaxes in both cases to a fixed pattern of all 0s or all 1s, correctly classifying the initial configuration.

configuration 0000000 is mapped to and so on until bit 127 corresponding to neighborhood configuration 1111111). The bit string ("genome") is of size $2^{2r+1} = 128$, resulting in a huge search space of size 2^{128}.

Each uniform CA (rule) in the population is run for a maximum number of M time steps, where M is selected anew for each rule from a Poisson distribution with mean 320. A rule's fitness is defined as the fraction of cell states correct at the last time step, averaged over $100 - 300$ initial configurations. At each generation a new set of configurations is generated at random, uniformly distributed over densities, i.e., $\rho(0) \in [0.0, 1.0]$. All rules are tested on this set and the population of the next generation is created by copying the top half of the current population (ranked according to fitness) unmodified; the remaining half of the next generation population is created by applying the genetic operators of crossover and mutation to selected rules from the current population. Note that while these operators act on the local mappings comprising a CA rule table (the "genotype"), selection is performed according to the dynamical behavior of the CA over a sample of initial configurations (the "phenotype").

Ref. 41 pointed out that sampling initial configurations with uniform distribution over $\rho \in [0.0, 1.0]$ is biased with respect to an unbiased distribution,

which is binomially distributed over $\rho \in [0.0, 1.0]$, and very strongly peaked at $\rho = 0.5$. Their preliminary experiments indicated that such a biased distribution is needed in order for the genetic algorithm to make progress in early generations. They found that in later generations this distribution tends to impede the genetic algorithm, because, as increasingly fitter rules are evolved, the initial configurations sample becomes less and less "challenging" for the evolutionary process (we shall re-encounter the unbiased versus biased issue in Section 6).

Using the genetic algorithm highly successful rules were found, with the best fitness values being in the range $0.93 - 0.95$. Under the above fitness function, the GKL rule has fitness ≈ 0.98; the genetic algorithm never found a rule with fitness above 0.95.[41,43]

In Refs. 41, 43, the evolutionary process, mediated by the genetic algorithm, was found to undergo a series of "epochs of innovation". The onset of an epoch is defined to be the generation at which a rule with a significant innovation in strategy is discovered. The onset generation of each epoch corresponded to a marked jump in the best fitness measured in the population. Four epochs of innovation were identified. The first epoch involves rules that specialize in low $\rho(0)$ or high $\rho(0)$; there are two best-performing strategies-rules that always relax to a fixed point of all 0s and rules that always relax to a fixed point of all 1s. Thus, these rules have fitness values of 0.5, having correctly classified half of the random initial configurations. At epoch 2, the genetic algorithm discovers rules that, while similar in behavior to epoch 1 rules, correctly classify some additional initial configurations with extreme $\rho(0)$ (i.e., a "low-$\rho(0)$ specialist" that classifies some very high-$\rho(0)$ initial configurations, and vice versa). Epoch 3 is characterized by a major innovation, upon the genetic algorithm's discovery of two distinct "block-expanding" strategies: go to the fixed point of all zeros (ones) unless there is a sufficiently large block of adjacent ones (zeros) in the initial configuration- if so, expand that block. Epoch 4 begins when no additional improvements are made; from that time on, the best fitness values remain approximately constant.

Ref. 41 analyzed the mechanisms by which the genetic algorithm evolved such strategies. This analysis uncovered interesting aspects of the algorithm, including a number of impediments that prevented it from discovering (on most runs) better-performance rules. A primary impediment is the algorithm's tendency to break the task's symmetries by producing low-$\rho(0)$ or high-$\rho(0)$ specialists. A pressure toward symmetry-breaking is effectively built into the fitness function, since specializing on one half of the initial configurations is an easy way to obtain higher fitness than that of a random rule. Symmetry-breaking thus produces a short-term gain for the genetic algorithm, but later

prevents it from making further improvements. Another impediment of the genetic algorithm applied to the density task is the biased sample of initial configurations, as discussed above. Interestingly, most of the identified impediments are also forces that help the genetic algorithm in the initial stages of its search.

Another interesting result of Ref. 43 concerns the λ parameter introduced in Refs. 51, 52 in order to study the structure of the space of CA rules. The λ of a given CA rule is the fraction of non-quiescent output states in the rule table, where the quiescent state is arbitrarily chosen as one of the possible k states. For binary-state CAs the quiescent state is usually 0 and therefore λ equals the fraction of output 1 bits in the rule table.

In recent years it has been speculated that computational capability can be associated with phase transitions in CA rule space.[51–53] This phenomenon, generally referred to as the "edge of chaos", asserts that dynamical systems are partitioned into ordered regimes of operation and chaotic ones with complex regimes arising on the edge between them. These complex regimes are hypothesized to give rise to computational capabilities. For CAs this means that there exist critical λ values at which phase transitions occur. The "edge of chaos" idea is similar to that of self-organized criticality studied by physicists.[54,55]

One of the main results of Ref. 43 regarding the density task is that most of the rules evolved to perform it are clustered around $\lambda \approx 0.43$ or $\lambda \approx 0.57$. This is in contrast to Ref. 42, where most rules are clustered around $\lambda \approx 0.24$ or $\lambda \approx 0.83$, which correspond to λ_c values, i.e., critical values near the transition to chaos.

The results obtained in Ref. 43 concerning the density task, coupled with a theoretical argument given in their paper lead to the conclusion that the λ value of successful rules performing the density task is more likely to be close to 0.5, i.e., depends upon the ρ_c value. They argued that for this class of computational tasks, the λ_c values associated with an edge of chaos are not correlated with the ability of rules to perform the task. More recently, Ref. 56 have re-examined this issue, suggesting that in order to find out whether there is an edge of chaos and if so, whether evolution can take us to it, one must define a good measure of complexity. It was suggested that convergence time is such a measure, and demonstrated that on average critical rules converge more slowly than non-critical rules; furthermore, genetic evolution driven by convergence time produces a wide variety of complex rules. However, other results suggest that this may not always be a correct measure for a transition.[57]

How are we to comprehend the emergent computation exhibited by the evolved CAs? Understanding the results of an evolutionary algorithm is a general problem; typically the algorithm is set to find high-fitness individuals,

without specifying how these should be attained. In many cases it is difficult to understand the precise manner by which an evolved, high-fitness individual works.[6] In the case of CAs, the problem becomes even more difficult, since the emergent "program" is generally impossible to extract from the bits of the rule table.[a] As in natural evolution, it is very difficult to understand the "phenotype" (algorithm) from studying the "genotype" (rule table).

One possible approach, proposed in Refs. 58, 59, is to examine the space-time patterns created by the CA, and to reconstruct from these the "algorithm". They have developed a general method for analyzing the "intrinsic" computation embedded in space-time patterns in terms of "regular domains", "particles", and "particle interactions"; this is part of their "computational mechanics" framework for understanding computation in physical systems, based on concepts from formal computation theory. A "regular domain" is, roughly, a homogeneous region of space-time in which the same "pattern" appears (these can sometimes be identified by simple observation, as in Figure 3). The notion of a domain can be formalized by describing its pattern using the minimal deterministic finite automaton (DFA)[47] that accepts all and only those spatial configurations that are consistent with the pattern. Once a CA's regular domains have been detected, nonlinear filters can be constructed to filter them out, leaving just the deviations from these regularities. The resulting filtered space-time diagram reveals the propagation of domain walls; if these walls remain spatially localized over time, then they are called "particles". These particles are a primary mechanism for carrying information (or "signals") over long space-time distances. Logical operations on the signals are performed when the particles interact. The collection of domains, domain walls, particles, and particle interactions for a CA represents the basic information-processing elements embedded in the CA's behavior- the CA's "intrinsic" computation. The application of computational mechanics to the understanding of rules evolved by the genetic algorithm is discussed in Refs. 45, 46, 12, 6.

4.2 The synchronization task

The second task investigated by the EVCA group is the one-dimensional synchronization task:[46] given any initial configuration, the CA must reach a final configuration, within M time steps, that oscillates between all 0s and all 1s on successive time steps (e.g., Figure 7). As noted in Ref. 46, this is perhaps the simplest, non-trivial synchronization task.

The task is non-trivial since synchronous oscillation is a global property

[a] This can be compared to the weights of an artificial neural network, attained via some learning process, which do not yield to analysis through direct observation.

of a configuration, whereas a small radius CA employs only local interactions. Thus, while local regions of synchrony can be directly attained, it is more difficult to design CAs in which spatially distant regions are in phase. Since out-of-phase regions can be distributed throughout the lattice, transfer of information must occur over large distances (i.e., $O(N)$) to remove these phase defects and produce a globally synchronous configuration. Ref. 46 reported that in 20% of the evolutionary runs the genetic algorithm discovered CAs that successfully solve the task. As with the density problem, they found that the evolutionary process progresses through a series of "epochs of innovation", five of which were identified for the synchronization task; furthermore, they applied computational mechanics to study the evolved, emergent computation.

It is interesting to point out that the phenomenon of synchronous oscillations occurs in Nature, a striking example of which is exhibited by fireflies; thousands such creatures may flash on and off in unison, having started from totally uncoordinated flickerings.[60] Each insect has its own rhythm, which changes only through local interactions with its neighbors' lights. Another interesting case involves pendulum clocks; when several of these are placed near each other, they soon become synchronized by tiny coupling forces transmitted through the air or by vibrations in the wall to which they are attached (for a review on synchronous oscillations in Nature see Ref. 61).

5 Co-evolving non-uniform CAs by cellular programming

5.1 Non-uniform CAs

Up to this point we have been studying the classical CA model, using a standard genetic algorithm to evolve such CAs to perform non-trivial computational tasks. In the next two sections we study two generalizations of the original CA. We first consider *non-uniform CAs*, which function in the same way as uniform ones, the only difference being in the cellular rules that need not be identical for all cells. In the next section we generalize on a second aspect of CAs, namely their standard, homogeneous connectivity, demonstrating that (non-uniform) cellular rules can co-evolve concomitantly with the cellular connections, to produce higher-performance systems.

We first turn our attention to non-uniform CAs. These have been investigated in Ref. 62 that discusses a one-dimensional CA in which a cell probabilistically selects one of two rules at each time step. They showed that complex patterns appear characteristic of (Wolfram) class IV behavior (see also Ref. 63). Ref. 64 presents two generalizations of cellular automata, namely discrete neural networks and automata networks; these are compared

to the original model from a computational point of view which considers the classes of problems such models can solve. Our interest below lies in studying the non-uniform CA model from a computational aspect as well as from an evolutionary one.

Note that non-uniform CAs share the basic "attractive" properties of uniform ones, namely massive parallelism, locality of cellular interactions, and simplicity of cells (Section 1). From a hardware point of view we observe that the resources required by non-uniform CAs are identical to those of uniform ones since a cell in both cases contains a rule. Although simulations of uniform CAs on serial computers may optimize memory requirements by retaining a single copy of the rule, rather than have each cell hold one, this in no way detracts from our argument. Indeed, one of the primary 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.[65] As noted in Section 1, one of our goals is the attainment of truly 'evolving ware', *evolware*, with current implementations centering on hardware, while raising the possibility of using other forms of ware in the future, such as *bioware*.[13,14]

Note that the original, uniform CA model is essentially "programmed" at an extremely low-level;[66] a *single* rule is sought that must be applied universally to all cells in the grid, a task that may be arduous even if one takes an evolutionary approach. For non-uniform CAs search space sizes are vastly larger than with uniform CAs, a fact that initially seems as an impediment. However, we have found that the model presents novel dynamics, offering new and interesting paths in the study of complex adaptive systems.

We have previously applied the non-uniform CA model to the investigation of artificial life issues,[67–69] and have also demonstrated its computation-universality.[70,71] The latter involves the demonstration that non-uniform CAs are equivalent to universal Turing machines in terms of computational power.[47] The universal systems presented are simpler than 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. We have also introduced the *cellular programming* algorithm, by which non-uniform CAs are co-evolved to perform computations; as opposed to the standard genetic algorithm, where a population of *independent* problem solutions *globally* evolves (Section 3), our approach involves a grid of rules that *co-evolves locally*.[13,71–76] In what follows we focus our attention on the co-evolution of non-uniform CAs, first presenting the cellular programming algorithm, followed by its application to several computational tasks, including density and synchronization.

5.2 The cellular programming algorithm

We study 2-state, non-uniform CAs, in which each cell may contain a different rule. A cell's rule table is encoded as a bit string, known as the "genome", containing the next-state (output) bits for all possible neighborhood configurations (as explained in Section 4.1). 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.[b] 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 (in our simulations we used $M \approx N$ so that computation time is linear with grid size). 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 Figure 4. In our simulations, the total number of initial configurations per evolutionary run was in the range $[10^5, 10^6]$.[c]

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 algorithm and the standard genetic algorithm (Section 3): (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 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. (b) The standard genetic algorithm involves a population of *independent* problem solu-

[b] To increase rule diversity in the initial grid, the rule tables were randomly chosen so as to be uniformly distributed among different λ values. Note that our algorithm is not necessarily restricted to a single, non-uniform CA since an ensemble of distinct grids can evolve independently in parallel.

[c] By comparison, Refs. 41, 43 employed a genetic algorithm with a population size of 100, which was run for 100 generations; every generation each CA was run on $100 - 300$ initial configurations, resulting in a total of $[10^6, 3 \cdot 10^6]$ configurations per evolutionary run.

```
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 the cellular neighborhood includes
                                     more than two cells)
            end if
            f_i = 0
        end parallel for
    end if
end while
```

Figure 4: Pseudo-code of the cellular programming algorithm.

Table 1: List of computational tasks for which non-uniform CAs were evolved via cellular programming.

Task	**Description**	**Grid**
Density	Decide whether the initial configuration contains a majority of 0s or of 1s	1D, r=1 2D, 5-neighbor
Synchronization	Given any initial configuration, relax to an oscillation between all 0s and all 1s	1D, r=1 2D, 5-neighbor
Ordering	Order initial configuration so that 0s are placed on the left and 1s are placed on the right	1D, r=1
Rectangle-Boundary	Find the boundaries of a randomly-placed, random-sized non-filled rectangle	2D, 5-neighbor
Thinning	Find thin representations of rectangular patterns	2D, 5-neighbor
Random Number Generation	Generate "good" sequences of pseudo-random numbers	1D, r=1

tions; the CAs 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 *co-evolves* since each cell's fitness depends upon its evolving neighbors.[d]

This latter point comprises a prime difference between our algorithm and parallel genetic algorithms, which have attracted attention over the past few years. These aim to exploit the inherent parallelism of evolutionary algorithms, thereby decreasing computation time and enhancing performance.[38–40] A number of models have been suggested, among them coarse-grained, island models,[77–79] and fine-grained, grid models.[80,81] The latter resemble our system in that they are massively parallel and local; however, the co-evolutionary aspect is missing. As we wish to attain a system displaying global computation, the individual cells do not evolve independently as with genetic algorithms (be they parallel or serial), i.e., in a "loosely-coupled" manner, but rather co-evolve, thereby comprising a "tightly-coupled" system.

In the subsections below we apply the cellular programming algorithm to six computational tasks, "revisiting" the density and synchronization problems, as well as studying four additional ones, which are motivated by real-world applications; these tasks are summarized in Table 1.

[d]This may also be considered a form of symbiotic cooperation, which falls, as does co-evolution, under the general heading of "ecological" interactions (see Ref. 6, pages 182-183).

5.3 The density task revisited

Our first application of the cellular programming algorithm involves the co-evolution of non-uniform CAs to perform the density task.[72] While the work reported in Section 4 concentrated solely on one-dimensional, uniform CAs with radius $r = 3$, we studied, in addition to $r = 3$ *non-uniform* CAs, also minimal radius, $r = 1$ ones, as well as two-dimensional grids. We found that high-performance systems can be evolved in all cases; below, we review some of our results pertaining to $r = 1$ CAs and two-dimensional grids. Remember that with $r = 1$, each cell has access only to its own state and that of its two adjoining neighbors. Note that *uniform*, one-dimensional, 2-state, $r = 1$ CAs are not computation-universal[82] nor do they exhibit class IV behavior.[17,35]

For the cellular programming algorithm we used randomly generated initial configurations, uniformly distributed over densities in the range $[0.0, 1.0]$, with the size $N = 149$ CA being run for $M = 150$ time steps (as noted above, computation time is thus linear with grid size). Fitness scores are assigned to each cell following the presentation of an initial configuration, according to whether the cell is in the correct state after M time steps or not (as described in Section 5.2). In what follows, the *performance* measure reported is defined as the average fitness of all grid cells over the last C configurations, normalized to the range $[0.0, 1.0]$.[e]

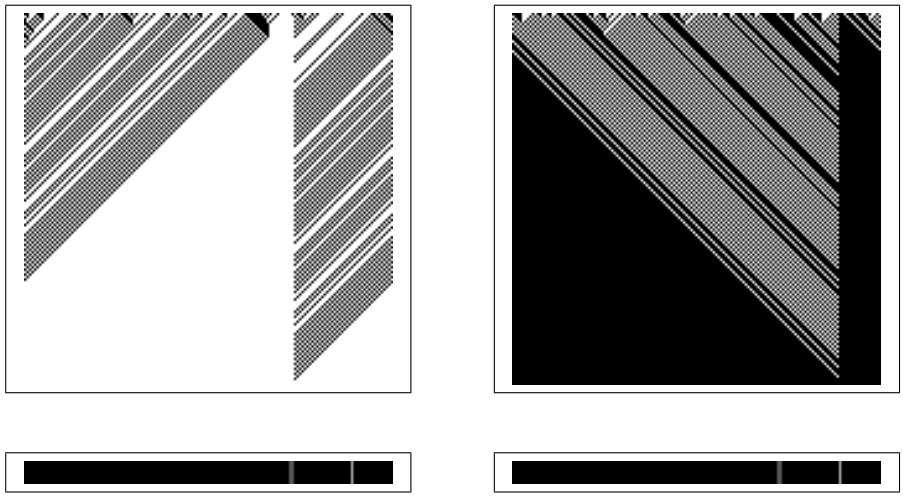
The search space of one-dimensional, non-uniform, $r = 1$ CAs is extremely large; since each cell contains one of 2^8 possible rules, this space is of size $(2^8)^{149} = 2^{1192}$. In contrast, the size of *uniform*, $r = 1$ CA rule space is small, consisting of only $2^8 = 256$ rules; this enabled us to test each and every one of these rules on the density task, a feat not possible for larger values of r. We found that the maximal performance of uniform CAs is 0.83.[f] Applying cellular programming yielded non-uniform CAs that exhibit performance values as high as 0.93; furthermore, these consist of a grid with a small number of distinct rules, one of which is "dominant", a situation referred to as quasi-uniformity (see Section 5.1). Thus, one of our major results is that co-evolved, quasi-uniform, $r = 1$ CAs outperform any possible uniform, $r = 1$ CA.[72]

Figure 5 demonstrates the operation of one such co-evolved CA along with a rules map, depicting the distribution of rules within the grid by assigning a unique color to each distinct rule. In this example, the grid consists of 146 cells containing rule 226, 2 cells containing rule 224 and one cell containing

[e] This is somewhat different than the performance measures defined in Refs. 41, 43; for a discussion of this issue see Ref. 75.

[f] The fitness (performance) measure for uniform CAs is identical to that of the evolved non-uniform CAs, i.e., the number of cells in the correct state after M time steps, averaged over the presented random initial configurations, normalized to the range $[0.0, 1.0]$.

rule 234.[g] The non-dominant rules act as "buffers", preventing information from flowing too "freely", and making local corrections to passing signals. A detailed investigation of the evolutionary process, engendered by application of the cellular programming algorithm, as well as the resulting systems can be found in Refs. 71, 72.



(a) (b)

Figure 5: One-dimensional density task: Operation of a co-evolved, 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 1s is 0.40, final density is 0. (b) Initial density of 1s is 0.60, final density is 1.

Both the density and synchronization tasks can be extended in a straightforward manner to two-dimensional grids.[h] Applying our algorithm to the evolution of such CAs to perform the density task yielded notably higher performance than the one-dimensional case, with peak values of 0.99; furthermore, computation time, i.e., the number of time steps taken by the CA until convergence to the correct final pattern, is shorter (we shall elaborate upon these improved results in Section 6). Figure 6 demonstrates the operation of one such

[g] Rule numbers are given in accordance with Wolfram's convention,[35] representing the decimal equivalent of the binary number encoding the rule table. For example, the rule depicted in Figure 1 is rule 232.

[h] Spatially periodic boundary conditions are applied, resulting in a toroidal grid.

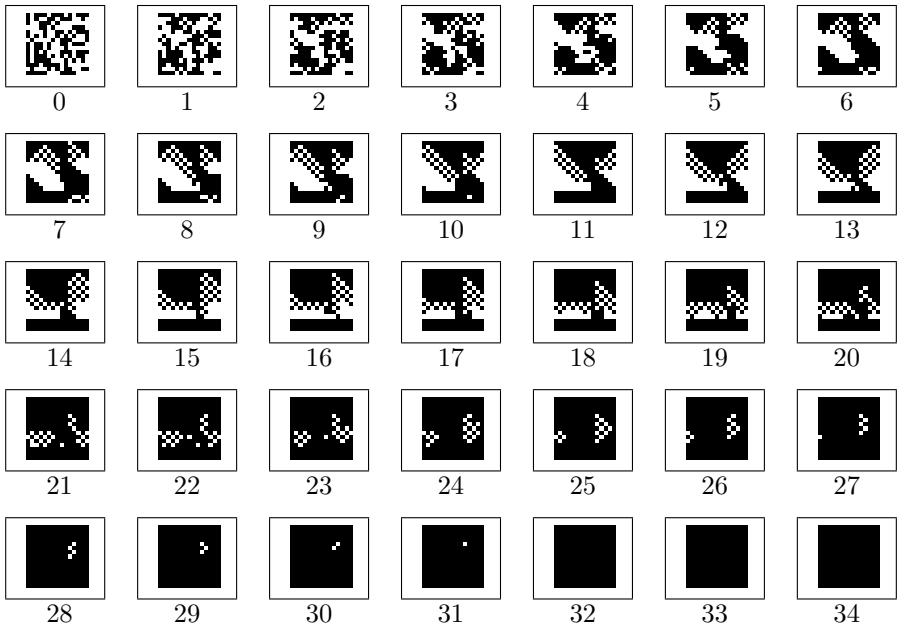


Figure 6: Two-dimensional density task: Operation of a co-evolved, non-uniform, 2-state, 5-neighbor CA. Grid size is $N = 225$ (15×15). Initial density of 1s is 0.51, final density is 1. Numbers at bottom of images denote time steps.

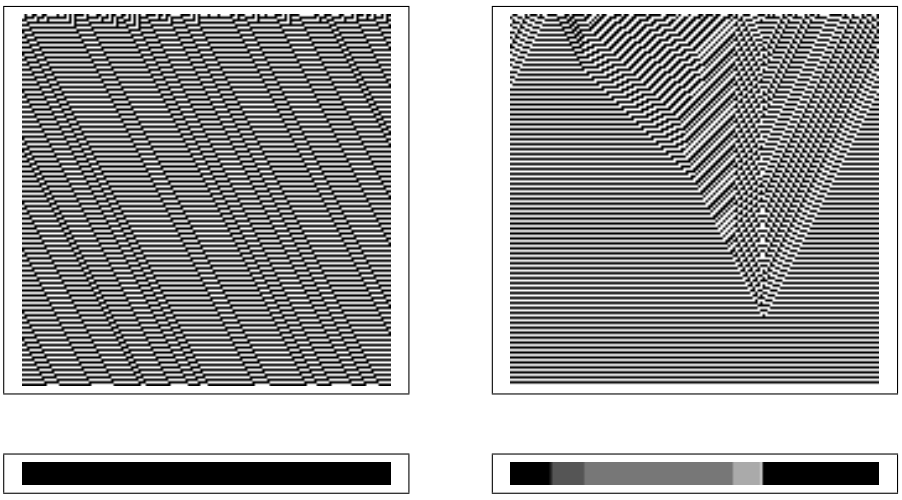
co-evolved CA. Qualitatively, we observe the CA's "strategy" of successively classifying local densities, with the locality range increasing over time; "competing" regions of density 0 and density 1 are manifest, ultimately relaxing to the correct fixed point.

5.4 The synchronization task revisited

Applying cellular programming to the synchronization task yielded (on most runs) quasi-uniform, $r = 1$ CAs with a fitness value of 1, i.e., perfect performance is attained.[i] As for the density task, we tested all possible *uniform*, $r = 1$ CAs on this task, finding that the highest performance value is 0.84. Thus, we note again that non-uniform CAs can be co-evolved to successfully solve this

[i] The term 'perfect' is used here in a stochastic sense since we cannot exhaustively test all 2^{149} possible initial configurations nor are we in possession to date of a formal proof; nonetheless, we have tested our best-performance CAs on numerous configurations, for all of which synchronization was attained.

task, their performance surpassing that of all uniform, $r = 1$ CAs. Figure 7 depicts the operation of two CAs: a high-performance uniform CA and a co-evolved, non-uniform CA. We have also experimented with two-dimensional grids obtaining highly successful results as with the one-dimensional case.



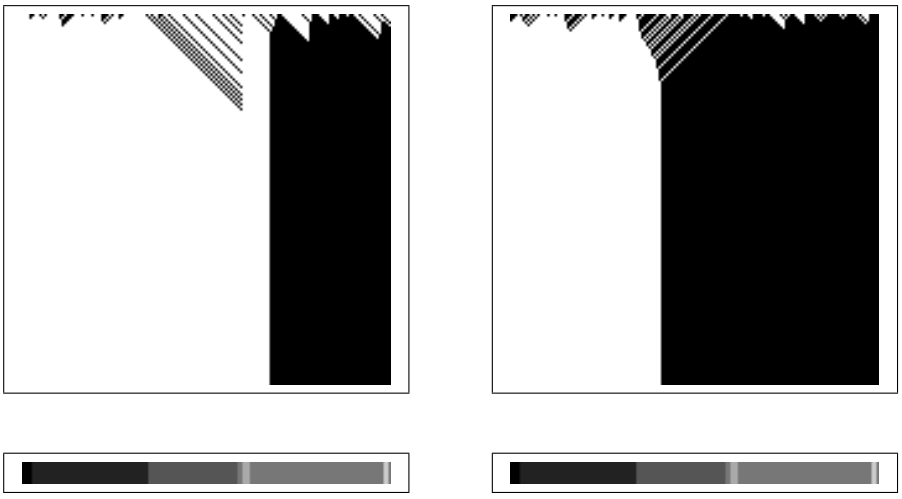
(a) (b)

Figure 7: One-dimensional synchronization task: Operation of two $r = 1$ CAs. Grid size is $N = 149$. Initial configurations were generated at random. Top figures depict space-time diagrams, bottom figures depict rule maps. (a) Uniform rule 31 (one of the best-performance uniform CAs for this task). (b) A co-evolved, non-uniform, $r = 1$ CA.

5.5 The ordering task

In this task, the non-uniform, one-dimensional, $r = 1$ CA, given any initial configuration, must reach a final configuration in which all 0s are placed on the left side of the grid and all 1s on the right side. This means that there are $N(1 - \rho(0))$ 0s on the left and $N\rho(0)$ 1s on the right, where $\rho(0)$ is the density of 1s at time 0, as defined in Section 4.1 (thus, the final density equals the initial one, however the configuration consists of a block of 0s on the left, followed by a block of 1s on the right). The ordering task may be considered a variant of the density task and is clearly non-trivial following our reasoning of Section 4.1. It is interesting in that the output is not a uniform configuration of all 0s or all 1s as with the density and synchronization tasks. Applying

cellular programming yielded quasi-uniform, $r = 1$ CAs with fitness values as high as 0.93, one of which is depicted in Figure 8; as with the previous tasks we were able to ascertain that this performance level is better than the maximal uniform, $r = 1$ CA performance, which is 0.71.



(a) (b)

Figure 8: One-dimensional ordering task: Operation of a co-evolved, non-uniform, $r = 1$ CA. Top figures depict space-time diagrams, bottom figures depict rule maps. (a) Initial density of 1s is 0.315, final density is 0.328. (b) Initial density of 1s is 0.60, final density is 0.59.

5.6 Image processing tasks

The possibility of applying CAs to perform image processing tasks arises as a natural consequence of their architecture; in a two-dimensional CA, a cell (or a group of cells) can correspond to an image pixel, with the CA's dynamics designed so as to perform a desired image processing task. Earlier work in this area, carried out mostly in the 1960s and the 1970s, was treated in Ref. 83, with more recent applications presented in Refs. 84, 85.

The next two tasks involve image processing operations. The first is a two-dimensional boundary computation: given an initial configuration consisting of a non-filled rectangle, the CA must reach a final configuration in which the rectangular region is filled, i.e., all cells within the confines of the rectangle

are in state 1, and all other cells are in state 0. Initial configurations consist of random-sized rectangles placed randomly on the grid (in our simulations, cells within the rectangle in the initial configuration were set to state 1 with probability 0.3; cells outside the rectangle were set to 0). Note that boundary cells can also be absent in the initial configuration. This operation can be considered a form of image enhancement, used, e.g., for treating corrupted images. Using cellular programming, non-uniform CAs were evolved with performance values of 0.99, one of which is depicted in Figure 9.

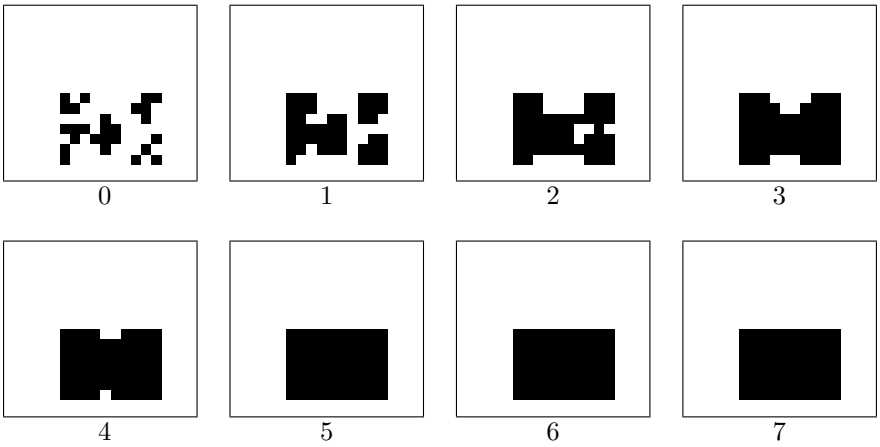


Figure 9: Two-dimensional rectangle-boundary task: Operation of a co-evolved, non-uniform, 2-state, 5-neighbor CA. Grid size is $N = 225$ (15×15). Numbers at bottom of images denote time steps.

Upon studying the (two-dimensional) rules map of the co-evolved, non-uniform CA, we found that the grid is quasi-uniform, with one dominant rule present in most cells. This rule maps the cell's state to zero if the number of neighboring cells in state 1 (including the cell itself) is less than two, otherwise mapping the cell's state to one;[j] thus, growing regions of 1s are more likely to occur within the rectangle confines than without.

The second task is that of thinning (also known as skeletonization), a fundamental preprocessing step in many image processing and pattern recognition algorithms. When the image consists of strokes or curves of varying thickness it is usually desirable to reduce them to thin representations located along the

[j] This is referred to as a totalistic rule, in which the state of a cell depends only on the sum of the states of its neighbors at the previous time step, and not on their individual states.[35]

approximate middle of the original figure. Such "thinned" representations are typically easier to process in later stages, entailing savings in both time and storage space.[86]

In Ref. 86 four sets of binary images were considered, two of which consist of rectangular patterns oriented at different angles. The algorithms presented therein employ a two-dimensional grid with a 9-cell neighborhood; each parallel step consists of two sub-iterations in which distinct operations take place. The set of images considered by us includes rectangular patterns oriented either horizontally or vertically; while more restrictive than that of Ref. 86, it is noted that we employ a smaller neighborhood (5-cell) and do not apply any sub-iterations.

Figure 10 demonstrates the operation of a co-evolved CA performing the thinning task. Although the evolved grid does not compute perfect solutions, we observe, nonetheless, good thinning "behavior" upon presentation of rectangular patterns as defined above (Figure 10a); furthermore, partial success is demonstrated when presented with more difficult images involving intersecting lines (Figure 10b).

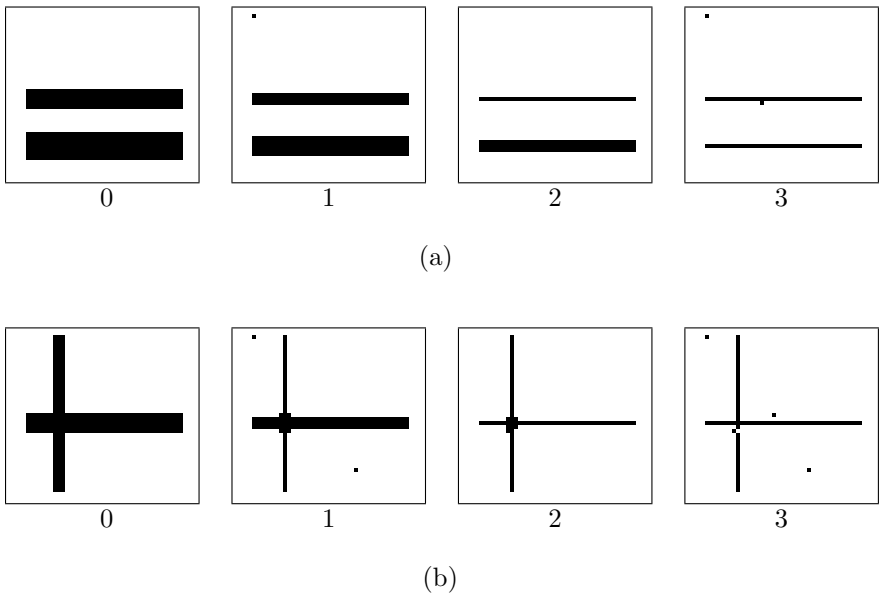


Figure 10: Two-dimensional thinning task: Operation of a co-evolved, non-uniform, 2-state, 5-neighbor CA. Grid size is $N = 1600$ (40×40). Numbers at bottom of images denote time steps. (a) Two separate lines. (b) Two intersecting lines.

5.7 Random number generation

Random numbers are needed in a variety of applications, yet finding good random number generators, or randomizers, is a difficult task.[87,88] 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. In the last decade cellular automata have been used to generate random numbers.[9,89,90]

In Refs. 73, 74 we applied the cellular programming algorithm to evolve random number generators. Essentially, the cell's fitness score for a single configuration (refer to Figure 4) is the entropy of the temporal bit sequence of that cell; higher entropy implies better fitness. This fitness measure was used to drive the evolutionary process, after which standard tests were applied to evaluate the quality of the evolved CAs. Our results suggest that good generators can indeed be evolved; these exhibit behavior at least as good as that of previously described CAs, with notable advantages arising from the existence of a "tunable" algorithm for obtaining random number generators.[73,74]

6 Co-evolving cellular architectures by cellular programming

In the previous section we presented the cellular programming approach, by which non-uniform CAs can be co-evolved to perform computational tasks. Such CAs comprise a generalization of the original CA model, by removing the uniformity-of-rules constraint. In this section we generalize on a second aspect of CAs, namely their standard, homogeneous connectivity.

In Section 5.3 we noted that the density task can be extended in a straightforward manner to two-dimensional grids, resulting in markedly higher evolved performance coupled with shorter computation times, in comparison to the one-dimensional case. This finding is intuitively understood by observing that a two-dimensional, locally-connected grid can be embedded in a one-dimensional grid with local and distant connections. This can be achieved, for example, by aligning the rows of the two-dimensional grid so as to form a one-dimensional array; the resulting embedded one-dimensional grid has distant connections of order $\sqrt{N}$, where N is the grid size. Since the density task is global it is likely that the observed superior performance of two-dimensional grids arises from the existence of distant connections that enhance information propagation across the grid.

Motivated by this observation concerning the effect of connection lengths on performance, we set out in Refs. 75, 76 to quantitatively study the relationship between performance and connectivity on the global density task, in

one-dimensional CAs; the results are summarized below (for a detailed account the reader is referred to the aforementioned papers).

We use the term *architecture* to denote the connectivity pattern of CA cells. In the standard one-dimensional model a cell is connected to r local neighbors on either side (in addition to a self-connection), where r is the radius (Section 2). The model we consider is that of non-uniform CAs with non-standard architectures, in which cells need not necessarily contain the same rule nor be locally connected; however, as with the standard CA model, each cell has a small, identical number of impinging connections. In what follows the term *neighbor* refers to a directly connected cell. We employed the cellular programming algorithm to evolve cellular rules for non-uniform CAs whose architectures are fixed (yet non-standard) during the evolutionary run, or evolve concomitantly with the rules; these are referred to as fixed or evolving architectures, respectively. Note that this bears some resemblance to Kauffman's NK model[55,91] in that connections as well as rules are heterogeneous; however, in our case, while these are *initially* assigned at random, they then *evolve* to perform a veritable *computation*, whereas Kauffman used *random* boolean networks to study issues related to fitness landscapes engendered by arbitrarily complex epistatic couplings. Furthermore, the K parameter, denoting the number of connections per cell, may vary from $K = 1$ to $K = N$, the latter representing a fully connected grid; in our case, the number of impinging connections per cell is kept small (i.e., we concentrate on very small K values).

We consider one-dimensional, symmetrical architectures where each cell has four neighbors, with connection lengths of a and b, as well as a self-connection. Spatially periodic boundary conditions are used, resulting in a circular grid. This type of architecture belongs to the general class of circulant graphs,[92] and is denoted by $C_N(a, b)$, where N is the grid size, a, b the *connection lengths* (Figure 11). The *distance* between two cells on the circulant is the number of connections one must traverse on the shortest path connecting them.

We surmised that attaining high performance on global tasks requires rapid information propagation throughout the CA, and that the rate of information propagation across the grid inversely depends on the average cellular distance (*acd*). It is straightforward to show that every $C_N(a, b)$ architecture is isomorphic to a $C_N(1, d')$ architecture, for some d', referred to as the *equivalent* d'.[75,76] We may therefore study the performance of $C_N(1, d)$ architectures, our conclusions being applicable to the general $C_N(a, b)$ case.

To study the effects of different architectures on performance, the cellular programming algorithm was applied to the evolution of cellular rules using fixed, non-standard architectures. We performed numerous evolutionary runs

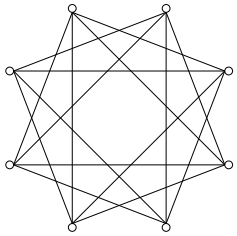


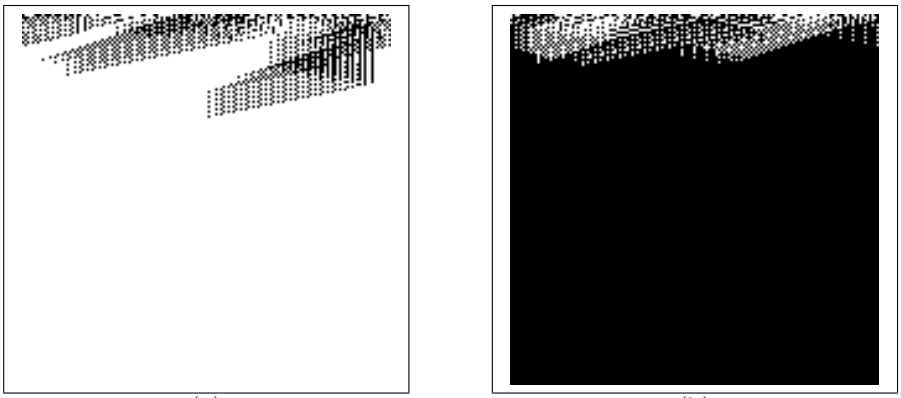
Figure 11: A $C_8(2,3)$ circulant graph. Each node is connected to four neighbors, with connection lengths of 2 and 3.

using $C_N(1,d)$ architectures with different values of d, recording the maximal performance attained during the run (for the definition of performance refer to Section 5.3). Our results showed that markedly higher performance is attained for values of d corresponding to low *acd* values and vice versa. While performance behaves in a rugged, non-monotonic manner as a function of d, we have found that it is *linearly* correlated with *acd* (with a correlation coefficient of 0.99, and a negligible p value).

These results demonstrate that performance is strongly dependent upon the architecture, with higher performance attainable by using different architectures than that of the standard CA model. As each $C_N(a,b)$ architecture is isomorphic to a $C_N(1,d)$ one, and as we have found that performance is correlated with *acd* in the $C_N(1,d)$ case, it follows that the performance of general $C_N(a,b)$ architectures is also correlated with *acd*. As an example of such an architecture, the operation of a co-evolved, $C_{149}(3,5)$ CA on the density task is demonstrated in Figure 12.

Having shown that cellular programming can be applied to the evolution of non-uniform CAs with fixed, non-standard architectures, we now ask whether a-priori specification of the connectivity parameters (a, b or d) is indeed necessary, or can an efficient architecture co-evolve along with the cellular rules. Moreover, can heterogeneous architectures, where each cell may have different d_i or (a_i,b_i) connection lengths, achieve high performance? Below we denote by $C_N(1,d_i)$ and $C_N(a_i,b_i)$ heterogeneous architectures with one or two evolving connection lengths per cell, respectively. Note that these are the cell's input connections, on which information is received; as connectivity is heterogeneous, input and output connections may be different, the latter specified implicitly by the input connections of the neighboring cells.

In order to evolve the architecture as well as the rules, the cellular programming algorithm of Section 5.2 is modified, such that each cellular "genome"



(a) (b)

Figure 12: The density task: Operation of a co-evolved, non-uniform, $C_{149}(3,5)$ CA. (a) Initial density of 1s is 0.48. (b) Initial density of 1s is 0.51. Note that computation time, i.e., the number of time steps until convergence to the correct final pattern, is shorter than that of the GKL rule (Figure 3). Furthermore, it can be qualitatively observed that the computational "behavior" is different than GKL, as is to be expected due to the different connectivity architecture.

consists of two "chromosomes". The first, encoding the rule table, is identical to that delineated in Section 5.2, while the second chromosome encodes the cell's connections. The two-level dynamics engendered by the concomitant evolution of rules and connections markedly increases the size of the space searched by evolution. Our results demonstrated that high performance can be attained, nonetheless, surpassing, in fact, that of the fixed-architecture CAs. Figure 13 demonstrates the operation of a co-evolved, $C_{129}(1,d_i)$ CA on the density task.

In Section 4.1 we noted that Refs. 41, 43 discussed two possible choices of initial configurations, either uniformly distributed over densities in the range $[0.0,1.0]$, or binomially distributed over initial densities. In our studies of non-uniform CAs as applied to the density task (Section 5.3) we concentrated on the former distribution; nonetheless, we find that evolved, non-uniform CAs can attain high performance even when applying the more difficult binomial distribution. Observing the results presented in Table 2, we note that performance exceeds that of previously evolved CAs, coupled with markedly shorter computation times (as demonstrated, e.g., by Figure 13). It is important to note that this is achieved using only 5 connections per cell, as compared to 7 used by the fixed, standard-architecture CAs. It is most likely that our CAs could attain even better results using a higher number of connections per cell,

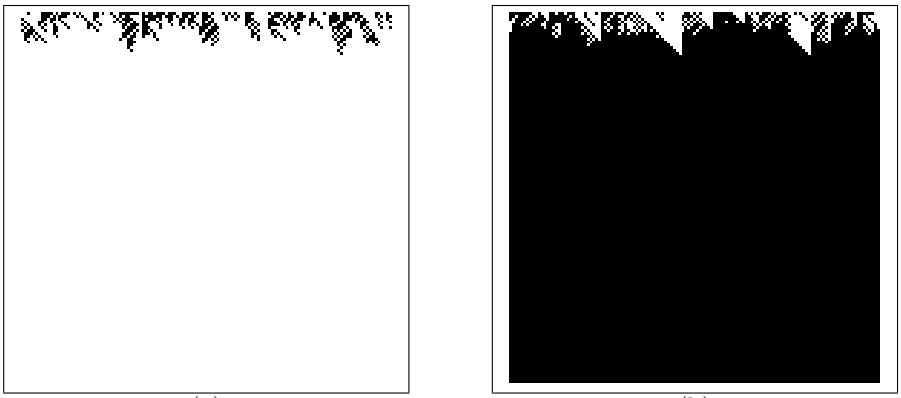


Figure 13: The density task: Operation of a co-evolved, non-uniform, $C_{129}(1, d_i)$ CA. (a) Initial density of 1s is 0.496. (b) Initial density of 1s is 0.504. Note that computation time is shorter than that of the fixed-architecture CA and markedly shorter than the GKL rule.

since this entails a notable reduction in *acd*.

In summary, our main findings concerning the co-evolution of cellular architectures are:

1. The performance of fixed-architecture CAs solving global tasks depends strongly and linearly on their average cellular distance. Compared with the standard $C_N(1, 2)$ architecture, considerably higher performance can be attained at very low connectivity values, by selecting a $C_N(1, d)$ or $C_N(a, b)$ architecture with a low *acd* value, such that $d, a, b \ll N$.

2. High performance architectures can be co-evolved using the cellular programming algorithm, thus obviating the need to specify in advance the precise connectivity scheme. Furthermore, as was shown in Ref. 75, it is possible to evolve such architectures that exhibit low *connectivity cost* per cell [k] as well as high performance.

7 Concluding remarks and future research

In this paper we explored the evolution of parallel cellular systems, embodied by cellular automata networks, pursuing the two-fold goal of: (1) enhancing our understanding of the fundamental principles of emergent computation in

[k] This is defined as d_i for the $C_N(1, d_i)$ case and $a_i + b_i$ for $C_N(a_i, b_i)$.

Table 2: A comparison of performance and computation times of the best CAs. $\mathcal{P}_{129,10^4}$ is a measure introduced by Mitchell *et al.*, representing the fraction of correct classifications performed by the CA of size $N = 129$ over 10^4 initial configurations randomly chosen from a binomial distribution over initial densities. $\mathcal{T}_{129,10^4}$ denotes the average computation time over the 10^4 initial configurations, i.e., the average number of time steps until convergence to the final pattern. $\#_c$ is the number of connections per cell. The CAs designated by (1), (2), and (3) are three of our co-evolved CAs; those designated by ϕ_i are CAs reported by Mitchell *et al.*

designation	rule(s)	architecture	$\#_c$	$\mathcal{P}_{129,10^4}$	$\mathcal{T}_{129,10^4}$
CA (1)	evolved, non-uniform	evolved, non-std.	5	0.791	17
CA (2)	evolved, non-uniform	evolved, non-std.	5	0.788	27
CA (3)	evolved, non-uniform	evolved, non-std.	5	0.781	12
ϕ_{100}	evolved, uniform	fixed, standard	7	0.775	72
ϕ_{11102}	evolved, uniform	fixed, standard	7	0.751	80
ϕ_{17083}	evolved, uniform	fixed, standard	7	0.743	107
GKL	designed, uniform	fixed, standard	7	0.825	74

natural systems, as well as of the ways by which it might have evolved, and (2) mimicking Nature's achievement, creating artificial systems based on these principles, matching the problem-solving capacities of their natural counterparts.

Specifically, applying a standard genetic algorithm to the evolution of cellular automata, the EVCA group demonstrated that high-performance systems can be attained for two non-trivial, global computational tasks, density and synchronization. They observed that evolution progresses through a series of "epochs of innovation", ending with highly fit individuals, though held back by a number of impediments, such as symmetry-breaking and the sampling of initial configurations. It was shown that for the density task, the critical λ_c values associated with an edge of chaos are not necessarily correlated with the ability of rules to perform the task. We also outlined the computational mechanics framework which is used to understand the rules and strategies evolved by the genetic algorithm.

We then studied two generalizations of the original CA model. The first consists of non-uniform CAs, where cellular rules need not be identical for all cells. Introducing the cellular programming algorithm for co-evolving such CAs, we applied it to six computational tasks, demonstrating that high-performance systems can be attained. As opposed to the standard genetic algorithm, where a population of independent problem solutions globally evolves, cellular programming involves a grid of rules that co-evolves locally. This renders it more

amenable to implementation as 'evolving ware', evolware. The second generalization of the original CA model involved non-standard, evolving architectures, where we demonstrated that yet higher-performance systems can be attained.

The work reported herein represents a first step in an exciting, nascent domain. While results to date are encouraging, there are still several possible avenues of future research, some of which have been explored to a certain extent, while others await to be pursued; we have attempted to assemble some of these below:

1. What classes of computational tasks are most suitable for such evolving cellular systems? and, what possible applications do they entail? We have noted feasible application areas such as image processing and random number generation. Clearly, more research is necessary in order to elaborate these directions as well as to find new ones.

2. Computation in cellular systems. How are we to understand the emergent, global computation arising in our locally-connected systems? This issue is interesting both from a theoretical point of view as well as from a practical one, where it may help guide our search for suitable classes of tasks for such systems.

3. Studying the evolutionary process. Both the standard genetic algorithm used to evolve uniform CAs and the local cellular programming algorithm used to co-evolve non-uniform CAs present interesting dynamics worthy of further study. We wish to enhance our understanding of how evolution creates complex, global behavior in such locally interconnected systems of simple parts.

4. Modifications of the evolutionary algorithms. The representation of CA rules (i.e., the "genome") used in the experiments reported herein consists of a bit string containing a lexicographic listing of all possible neighborhood configurations (Section 4.1). It has been noted in Ref. 93 that this representation is fairly low-level and brittle since a change of one bit in the rule table can have a large effect on the computation performed. They evolved uniform CAs to perform the density task using other bit-string representations, as well as a novel, higher-level one consisting of condition-action pairs; it was demonstrated that better performance is attained when employing the latter. More recently, Refs. 94, 95 used genetic programming,[9] in which a rule is represented by a LISP expression, to evolve uniform CAs to perform the density task; this resulted in a CA which outperforms the best known uniform, $r = 3$ density classifier, namely the GKL rule. These experiments demonstrate that changing

the bit-string representation, i.e., the encoding of the "genome", may entail notable performance gains; indeed, this issue is of prime import where evolutionary algorithms in general are concerned (for a discussion see, e.g., Ref. 6, chapter 5). Such encodings could be incorporated in the evolutionary algorithms presented herein. We noted in Section 5.2 that fitness in the cellular programming algorithm is assigned locally to each cell; another possibility might be to assign fitness scores to blocks of cells, in accordance with their mutual degree of success on the task at hand. The cellular programming algorithm in particular presents novel dynamics due its local, co-evolutionary nature, and it is clear that there is much yet to be explored.

5. Modifications of the cellular system model. We presented a number of generalizations of the CA model, including non-uniformity of rules, non-standard architectures, and heterogeneous architectures. Other possible modifications include: (1) The application of asynchronous state updating, an issue which we had previously investigated in Ref. 68 in a somewhat different model (currently synchronous updating is applied, i.e., all cells are updated at once).[96] (2) Non-deterministic updating, also connected with the issue of robustness, namely how resistant are our systems in the face of errors (e.g., how is the computation affected when a small probability of error is introduced in the updating of cell states?).[96] (3) Three-dimensional grids (and tasks). In this paper we studied one- and two-dimensional grids; ultimately, three-dimensional systems may be built, enabling new kinds of phenomena to emerge, in analogy to the physical world.[97] As a simple observation consider the fact that signal paths are more collision-prone in two dimensions whereas in three dimensions they may pass each other unperturbed (for example, the mammalian brain). We also noted in Section 6 the advantages of two-dimensional grids in terms of signal propagation, with respect to one-dimensional grids; following this reasoning, three-dimensional arrays could result in yet better systems.[75] Current technology is mostly two-dimensional (e.g., integrated circuits are basically composed of one or more 2D layers). Future systems, based, e.g., on molecular computing,[98] will be three-dimensional, possibly displaying an array of biological phenomena, including self-repair, self-reproduction, growth and evolution.[99]

   One of the motivations for the above modifications of the cellular system model is the desire to attain realistic systems that are more amenable to implementation as evolware.

6. Scaling. This involves two separate issues: the evolutionary algorithm

and the evolved solutions.

(a) How does the evolutionary algorithm scale with grid size? Though to date experiments have been conducted with different grid sizes, a more in-depth inquiry is needed. Note that as the cellular programming algorithm is local it scales better in terms of hardware resources than the standard (global) genetic algorithm; adding grid cells requires only local connections whereas the standard genetic algorithm includes global operators such as fitness ranking and crossover.

(b) How can larger grids be obtained from smaller (evolved) ones, i.e., how can evolved solutions be scaled? This has been purported as an advantage of uniform CAs, since one can directly use the evolved rule in a grid of any desired size. However, this form of *simple* scaling does not bring about *task* scaling; as demonstrated, e.g., in Ref. 12 for the density task, performance decreases as grid size increases. For non-uniform CAs quasi-uniformity may facilitate scaling since only a small number of rules must ultimately be considered. To date we have attained successful systems for the random number generation task using a simple scaling scheme involving the duplication of the rules grid;[73] we are currently exploring a more sophisticated scaling approach, with preliminary encouraging results.

7. Hierarchy. The idea of decomposing a system into a hierarchy of layers, each carrying out a different function, is ubiquitous in natural as well as artificial systems. As an example of the former, one can cite the human visual system, which begins with low-level image processing in the retina, ending in high-level operations, such as face-recognition, performed in the visual cortex. Artificial, feed-forward neural networks are an example of artificial systems exhibiting a layered structure. This idea can be incorporated within our framework, thereby obtaining a hierarchical system, composed of evolving, layered grids; this could improve the system's performance, facilitate its scaling, and indeed enable entirely new (possibly more difficult) tasks to be confronted.

8. Implementation. As discussed above, this is a major motivation driving this work, the goal being to construct evolware.

Evolving, cellular systems hold potential both scientifically, as vehicles for studying phenomena of interest in areas such as complex adaptive systems and artificial life, as well as practically, showing a range of potential future applications ensuing the construction of adaptive systems. We hope this paper

has shed some light on the behavior of parallel cellular systems, the complex computation they exhibit, and the application of artificial evolution to their construction.

Acknowledgments

I wish to thank Daniel Mange, Melanie Mitchell, and Eduardo Sanchez for their valuable comments and suggestions. I am grateful to Eytan Ruppin for his general support and for our enjoyable mutual collaboration on the co-evolution of cellular architectures. I thank Dietrich Stauffer for his careful reading of this manuscript and his many helpful suggestions. I am especially grateful to Marco Tomassini for our many stimulating discussions and mutual work, results of which are disseminated throughout this paper.

References

1. P. Coveney and R. Highfield. *Frontiers of Complexity: The Search for Order in a Chaotic World*. Faber and Faber, London, 1995.
2. K. Kaneko, I. Tsuda, and T. Ikegami, editors. *Constructive Complexity and Artificial Reality, Proceedings of the Oji International Seminar on Complex Systems- from Complex Dynamical Systems to Sciences of Artificial Reality*, volume 75, Nos. 1-3 of *Physica D*, August 1994.
3. H. R. Pagels. *The Dreams of Reason: The Computer and the Rise of the Sciences of Complexity*. Bantam Books, New York, 1989.
4. T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, New York, 1996.
5. Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, Heidelberg, third edition, 1996.
6. M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.
7. H.-P. Schwefel. *Evolution and Optimum Seeking*. John Wiley & Sons, New York, 1995.
8. D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, Piscataway, NJ, 1995.
9. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, Massachusetts, 1992.
10. D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.
11. J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. University of Michigan Press, Ann Arbor, Michigan, 1975. (Second edition, Cambridge, MA: MIT Press, 1992).

12. J. P. Crutchfield and M. Mitchell. The evolution of emergent computation. *Proceedings of the National Academy of Sciences USA*, 92(23):10742–10746, 1995.
13. M. Sipper. Designing evolware by cellular programming. In T. Higuchi, M. Iwata, and W. Liu, editors, *Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware (ICES96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 81–95. Springer-Verlag, Heidelberg, 1997.
14. M. Goeke, M. Sipper, D. Mange, A. Stauffer, E. Sanchez, and M. Tomassini. Online autonomous evolware. In T. Higuchi, M. Iwata, and W. Liu, editors, *Proceedings of the First International Conference on Evolvable Systems: From Biology to Hardware (ICES96)*, volume 1259 of *Lecture Notes in Computer Science*, pages 96–106. Springer-Verlag, Heidelberg, 1997.
15. E. Sanchez and M. Tomassini, editors. *Towards Evolvable Hardware*, volume 1062 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 1996.
16. J. von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Illinois, 1966. Edited and completed by A. W. Burks.
17. S. Wolfram. Universality and complexity in cellular automata. *Physica D*, 10:1–35, 1984.
18. T. Toffoli and N. Margolus. *Cellular Automata Machines*. The MIT Press, Cambridge, Massachusetts, 1987.
19. A. Burks, editor. *Essays on Cellular Automata*. University of Illinois Press, Urbana, Illinois, 1970.
20. A. Smith. Cellular automata theory. Technical Report 2, Stanford Electronic Lab., Stanford University, 1969.
21. J.-Y. Perrier, M. Sipper, and J. Zahnd. Toward a viable, self-reproducing universal computer. *Physica D*, 97:335–352, 1996.
22. M. Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223(4):120–123, October 1970.
23. M. Gardner. On cellular automata, self-reproduction, the Garden of Eden and the game "life". *Scientific American*, 224(2):112–117, February 1971.
24. E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways for your Mathematical Plays*, volume 2, chapter 25, pages 817–850. Academic Press, New York, 1982.
25. K. Culik II, L. P. Hurd, and S. Yu. Computation theoretic aspects of cellular automata. *Physica D*, 45:357–378, 1990.
26. T. Toffoli. Cellular automata mechanics. Technical Report 208, Comp. Comm. Sci. Dept., The University of Michigan, 1977.
27. E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21:219–253, 1982.
28. N. Margolus. Physics-like models of computation. *Physica D*, 10:81–95, 1984.
29. T. Toffoli. Reversible computing. In J. W. De Bakker and J. Van Leeuwen, editors, *Automata, Languages and Programming*, pages 632–644. Springer-Verlag, 1980.

30. T. Toffoli. Cellular automata as an alternative to (rather than an approximation of) differential equations in modeling physics. *Physica D*, 10:117–127, 1984.
31. J. Hardy, O. De Pazzis, and Y. Pomeau. Molecular dynamics of a classical lattice gas: Transport properties and time correlation functions. *Physical Review A*, 13:1949–1960, 1976.
32. U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the Navier-Stokes equation. *Physical Review Letters*, 56:1505–1508, 1986.
33. G. Vichniac. Simulating physics with cellular automata. *Physica D*, 10:96–115, 1984.
34. C. Bennett and G. Grinstein. Role of irreversibility in stabilizing complex and nonenergodic behavior in locally interacting discrete systems. *Physical Review Letters*, 55:657–660, 1985.
35. S. Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, 55(3):601–644, July 1983.
36. S. Wolfram. Cellular automata as models of complexity. *Nature*, 311:419–424, October 1984.
37. G. B. Ermentrout and L. Edelstein-Keshet. Cellular automata approaches to biological modeling. *Journal of Theoretical Biology*, 160:97–133, 1993.
38. M. Tomassini. Evolutionary algorithms. In E. Sanchez and M. Tomassini, editors, *Towards Evolvable Hardware*, volume 1062 of *Lecture Notes in Computer Science*, pages 19–47. Springer-Verlag, Heidelberg, 1996.
39. M. Tomassini. A survey of genetic algorithms. In D. Stauffer, editor, *Annual Reviews of Computational Physics*, volume III, pages 87–118. World Scientific, Singapore, 1995.
40. E. Cantú-Paz. A summary of research on parallel genetic algorithms. Technical Report 95007, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL, July 1995.
41. M. Mitchell, J. P. Crutchfield, and P. T. Hraber. Evolving cellular automata to perform computations: Mechanisms and impediments. *Physica D*, 75:361–391, 1994.
42. 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.
43. 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.
44. 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.
45. 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, Heidelberg, 1994. Springer-Verlag.
46. 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.
47. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory Languages and Computation*. Addison-Wesley, Redwood City, CA, 1979.
48. M. Land and R. K. Belew. No perfect two-state cellular automata for density classification exists. *Physical Review Letters*, 74(25):5148–5150, June 1995.
49. P. Gacs, G. L. Kurdyumov, and L. A. Levin. One-dimensional uniform arrays that wash out finite islands. *Problemy Peredachi Informatsii*, 14:92–98, 1978.
50. P. Gonzaga de Sá and C. Maes. The Gacs-Kurdyumov-Levin automaton revisited. *Journal of Statistical Physics*, 67(3/4):507–522, 1992.
51. C. G. Langton. Computation at the edge of chaos: Phase transitions and emergent computation. *Physica D*, 42:12–37, 1990.
52. C. G. Langton. Life at the edge of chaos. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, volume X of *SFI Studies in the Sciences of Complexity*, pages 41–91, Redwood City, CA, 1992. Addison-Wesley.
53. W. Li, N. H. Packard, and C. G. Langton. Transition phenomena in cellular automata rule space. *Physica D*, 45:77–94, 1990.
54. P. Bak, C. Tang, and K. Wiesenfeld. Self-organized criticality. *Physical Review A*, 38(1):364–374, July 1988.
55. S. A. Kauffman. *The Origins of Order*. Oxford University Press, New York, 1993.
56. H. Gutowitz and C. Langton. Mean field theory of the edge of chaos. 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 52–64, Heidelberg, 1995. Springer-Verlag.
57. D. Stauffer and L. de Arcangelis. Dynamics and strong size effects of a bootstrap percolation problem. *International Journal of Modern Physics C*, 7:739–745, 1996.
58. J. E. Hanson and J. P. Crutchfield. The attractor-basin portrait of a cellular automaton. *Journal of Statistical Physics*, 66:1415–1462, 1992.
59. J. P. Crutchfield and J. E. Hanson. Turbulent pattern bases for cellular automata. *Physica D*, 69:279–301, 1993.
60. J. Buck. Synchronous rhythmic flashing of fireflies II. *The Quarterly Review of Biology*, 63(3):265–289, September 1988.
61. S. H. Strogatz and I. Stewart. Coupled oscillators and biological synchronization. *Scientific American*, 269(6):102–109, December 1993.
62. G. Y. Vichniac, P. Tamayo, and H. Hartman. Annealed and quenched inhomogeneous cellular automata. *Journal of Statistical Physics*, 45:875–883, 1986.

63. 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, Heidelberg, 1986.
64. M. Garzon. Cellular automata and discrete neural networks. *Physica D*, 45:431–440, 1990.
65. P. Gacs. Nonergodic one-dimensional media and reliable computation. *Contemporary Mathematics*, 41:125, 1985.
66. S. Rasmussen, C. Knudsen, and R. Feldberg. Dynamics of programmable matter. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, volume X of *SFI Studies in the Sciences of Complexity*, pages 211–254, Redwood City, CA, 1992. Addison-Wesley.
67. 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.
68. M. Sipper. Studying artificial life using a simple, general cellular model. *Artificial Life*, 2(1):1–35, 1995. The MIT Press, Cambridge, MA.
69. 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.
70. 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, Heidelberg, 1995. Springer-Verlag.
71. M. Sipper. *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*. Springer-Verlag, Heidelberg, 1997.
72. M. Sipper. Co-evolving non-uniform cellular automata to perform computations. *Physica D*, 92:193–208, 1996.
73. M. Sipper and M. Tomassini. Generating parallel random number generators by cellular programming. *International Journal of Modern Physics C*, 7(2):181–190, 1996.
74. M. Sipper and M. Tomassini. Co-evolving parallel random number generators. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 950–959. Springer-Verlag, Heidelberg, 1996.
75. M. Sipper and E. Ruppin. Co-evolving architectures for cellular machines. *Physica D*, 99:428–441, 1997.
76. M. Sipper and E. Ruppin. Co-evolving cellular architectures by cellular programming. In *Proceedings of IEEE Third International Conference on Evolutionary Computation (ICEC'96)*, pages 306–311, 1996.
77. T. Starkweather, D. Whitley, and K. Mathias. Optimization using distributed genetic algorithms. In H.-P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature*, volume 496 of *Lecture Notes in Computer Science*, page 176, Heidelberg, 1991. Springer-Verlag.
78. J. P. Cohoon, S. U. Hedge, W. N. Martin, and D. Richards. Punctuated

equilibria: A parallel genetic algorithm. In J. J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, page 148. Lawrence Erlbaum Associates, 1987.
79. R. Tanese. Parallel genetic algorithms for a hypercube. In J. J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, page 177. Lawrence Erlbaum Associates, 1987.
80. M. Tomassini. The parallel genetic cellular automata: Application to global function optimization. In R. F. Albrecht, C. R. Reeves, and N. C. Steele, editors, *Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms*, pages 385–391. Springer-Verlag, 1993.
81. B. Manderick and P. Spiessens. Fine-grained parallel genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, page 428. Morgan Kaufmann, 1989.
82. K. Lindgren and M. G. Nordahl. Universal computation in simple one-dimensional cellular automata. *Complex Systems*, 4:299–318, 1990.
83. K. Preston, Jr. and M. J. B. Duff. *Modern Cellular Automata: Theory and Applications*. Plenum Press, New York, 1984.
84. A. Broggi, V. D'Andrea, and G. Destri. Cellular automata as a computational model for low-level vision. *International Journal of Modern Physics C*, 4(1):5–16, 1993.
85. G. Hernandez and H. J. Herrmann. Cellular-automata for elementary image-enhancement. *CVGIP: Graphical Models and Image Processing*, 58(1):82–89, January 1996.
86. Z. Guo and R. W. Hall. Parallel thinning with two-subiteration algorithms. *Communications of the ACM*, 32(3):359–373, March 1989.
87. 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.
88. F. Schmid and N. B. Wilding. Errors in Monte Carlo simulations using shift register random number generators. *International Journal of Modern Physics C*, 6(6):781–787, 1995.
89. S. Wolfram. Random sequence generation by cellular automata. *Advances in Applied Mathematics*, 7:123–169, June 1986.
90. 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.
91. D. Stauffer and N. Jan. Size effects in Kauffman type evolution for rugged fitness landscapes. *Journal of Theoretical Biology*, 168:211–218, 1994.
92. F. Buckley and F. Harary. *Distance in Graphs*. Addison-Wesley, Redwood City, CA, 1990.
93. M. Land and R. K. Belew. Towards a self-replicating language for computation. In J. R. McDonnell, R. G. Reynolds, and D. B. Fogel, editors, *Evolutionary programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 403–413, Cambridge, Massachusetts, 1995. The MIT Press.

94. D. Andre, F. H Bennett III, and J. R. Koza. Evolution of intricate long-distance communication signals in cellular automata using genetic programming. In C. Langton and T. Shimohara, editors, *Artificial Life V: Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems*, Cambridge, MA, 1996. The MIT Press.
95. D. Andre, F. H Bennett III, and 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, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 3–11, Cambridge, MA, 1996. The MIT Press.
96. M. Sipper, M. Tomassini, and M. S. Capcarrère. Designing cellular automata using a parallel evolutionary algorithm. *Nuclear Instruments & Methods in Physics Research, Section A*, 389(1-2):278–283, 1997.
97. H. de Garis. "Cam-Brain" ATR's billion neuron artificial brain project: A three year progress report. In *Proceedings of IEEE Third International Conference on Evolutionary Computation (ICEC'96)*, pages 886–891, 1996.
98. K. E. Drexler. *Nanosystems: Molecular Machinery, Manufacturing and Computation*. John Wiley, New York, 1992.
99. D. Mange and A. Stauffer. Introduction to embryonics: Towards new self-repairing and self-reproducing hardware based on biological-like properties. In N. M. Thalmann and D. Thalmann, editors, *Artificial Life and Virtual Reality*, pages 61–72, Chichester, England, 1994. John Wiley.