Evolving Cellular Automata to Perform User-Defined Computations

Paul Grouchy[1] and Gabriele M.T. D'Eleuterio[1]

[1]University of Toronto Institute for Aerospace Studies, Toronto, Ontario, Canada M3H 5T6
paul.grouchy@mail.utoronto.ca, gabriele.deleuterio@utoronto.ca

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alif2016/28/84/2325865/978-0-262-33936-0-ch021.pdf by guest on 14 August 2024

Abstract

A novel genetic algorithm for evolving both uniform and nonuniform cellular automata (CA) to perform user-defined computations is presented. Unlike previous approaches, the CAs evolved here can in general take as their input and their output only a subset of the cells, allowing for the design of CAs that are larger than the number of inputs required by the desired computation. It also provides greater flexibility compared with previous work in terms of the number of possible outputs. We test our algorithm by attempting to evolve both uniform and nonuniform 1D CAs of varying sizes to compute the sum of two 4-bit strings, a computation requiring 8 inputs and 5 outputs. Results demonstrate that while the algorithm is unable to discover solutions using 8-cell CAs, expanding the number of cells beyond the number of inputs enables the autonomous design of 4-bit adders.

Introduction

In their most basic form, cellular automata (CA) are a collection of simple identical components ("cells") whose behaviors are governed by local interactions (Von Neumann et al., 1966; Wolfram, 1984). Time in a CA is discrete and, at each timestep, a cell can be in one of k states. If $k = 2$, for example, a cell's state can be either 0 or 1 at a given timestep. To execute a CA, one must first "seed" each cell with an initial value. At each subsequent timestep, the CA's rule set determines how a given cell's state is updated based on the cell's current state and the states of its neighbors. The size of the neighborhood of surrounding cells that can influence a given cell's state is determined by the CA's radius r. If $r = 1$, for example, a cell's state is updated based on its current state and the states of its adjoining neighbors. Example rule sets for 1D CAs with $r = 1$ can be found in Figs. 1 and 3.

CAs typically have periodic boundary conditions, i.e., cells in 1D CAs are organized in a ring, cells in 2D CAs are organized in a toroid, etc. Canonical CAs are uniform (homogeneous), meaning that each cell's state is updated using the same rule set. However nonuniform (nonhomogeneous) CAs, where a cell's state can be updated based on one of two or more rule sets, were also studied (e.g., Sipper, 1996). Uniform 1D CAs with $k = 2$ and $r = 1$ are called elementary CAs and it has been shown that at least one such CA, "rule 110," is Turing complete and thus capable of universal computation (Cook, 2004).

Owing to the size of the set of possible rule sets ($k^{k^{2r+1}}$ in the case of a uniform 1D CA), a brute-force search for CAs that perform a specific computation is often intractable. Therefore, an important area of research is the autonomous design of CAs via genetic algorithms (GAs) and other forms of evolutionary computation (Sapin et al., 2009; Cenek and Mitchell, 2009).

Here we present a novel, flexible genetic algorithm for evolving uniform and nonuniform CAs to perform user-defined computations. A key feature of this algorithm is that the number of inputs and outputs for the desired computation are each allowed to vary independently from each other, as well as from the number of cells in the CA, as long as the CA is at least the same size as the larger of the two. We demonstrate the capabilities of this algorithm by evolving 1D CAs of various sizes to successfully compute the sum of any two 4-bit strings.

Related Work

Nils Aall Barricelli was probably the first person to experiment with evolving CAs, using one of the first computers ever built (Barricelli, 1963). More recently, the work in (Packard, 1988), as well as by the Evolving Cellular Automata group at the Santa Fe Institute (Mitchell et al., 1996), sparked a flurry of research into the use of genetic algorithms to discover CAs capable of performing user-specified computations that continues to this day (for a review, see Iclănzan et al., 2011). In (Mitchell et al., 1993), for example, a GA was employed to discover rule sets for uniform 149-cell CAs with $k = 2$ and $r = 3$ that solve the density classification task (DCT). To solve the DCT, a CA that has its cells seeded with a vector of bits must, after a fixed number of iterations, resolve all of its cells to 1 in the case where the original vector contained more than 50% 1s, otherwise all of its cells should be 0. Thus the number of inputs to the CA is the same as its size, and the output of the computation is determined from the final state of all of the CA cells. In (Sipper, 1996), nonuniform CAs were evolved to

solve the DCT task using a coevolutionary algorithm where each cell had its own genome (rule set) and its own fitness (evaluated independently from the other cell's final states), and selection and reproduction occurred only within local neighborhoods. This work also demonstrated that for $r = 1$, a nonuniform CA could be evolved to achieve a high score on the DCT problem that is theoretically impossible to attain with a uniform CA.

CAs can be evolved to perform a wide range of computational tasks by first framing the problem in a manner similar to the DCT described above. For example, in the synchronization task (e.g., Oliveira et al., 2009), the CA is tasked with alternating between having all of its cells set to 1 and all cells set to 0, after having initially been iterated for a fixed number of timesteps starting from an arbitrary initial state. An interesting property of these types of CA computations is that global consensus among cells is achieved solely through local interactions. Besides searching the space of possible rule sets, genetic algorithms can also search for appropriate neighborhood topologies for a given task, as in (Darabos et al., 2013).

Conway's Game of Life, a 2D CA, is also Turing complete (Berlekamp et al., 2004). Computations in this case are performed by interpreting the interactions of patterns called "gliders" (Sapin et al., 2009). By searching for rules that produce these patterns, GAs can discover novel CAs for simulating logic gates such as AND and NOT gates (e.g., Sapin et al., 2004, 2009).

Finally, controllable CAs (CCAs) are evolvable 1D nonuniform CAs tasked with producing pseudorandom numbers (Guan and Zhang, 2003). At each iteration, the current state of a separate uniform CA (that uses a preselected rule set) is used to determine which of two other preprogrammed rule sets to apply to each cell in the CCA. In addition, a second separate uniform CA (using a fourth preselected rule set) is used to determine whether "controllable" cells act as normal cells, or operate according to a predefined behavior, such as "keep current state." The genetic algorithm is then tasked with determining which cells in the CCA are controllable, and which are considered to be output cells. The values of the output cells get translated into a number after each iteration of the CCA, and after many iterations the final set of numbers are evaluated on their randomness to determine the fitness of the CCA in question. In the algorithm presented below, the locations of input and output cells are evolved in a manner similar to how the locations of the controllable and output cells are evolved in CCAs. However, contrary to CCAs, cells in the work presented here use the same rule set at each iteration, and the number of possible rule sets that can be applied to a cell, as well as the rules themselves, are also evolvable. Furthermore, initial conditions are also evolvable in this work, whereas they are randomly generated in the case of CCAs.

EvoCA

In an effort to expand the range of problems one can tackle using evolved CAs, we present EvoCA, a genetic algorithm that allows users to evolve CAs to perform computations requiring a number of cells equal to *or greater than* the number of inputs. Furthermore, EvoCA allows for the number of output cells to vary from a single cell to having each cell in the CA be an output. The number of allowed output cells is independent from the number of inputs, and vice versa. EvoCA implements these additional features by allowing the genetic algorithm to choose a unique cell for each input and output. While a given cell can only be assigned to at most one input and at most one output, cells can be concurrently designated as both an input and an output.

EvoCA is capable of evolving both uniform and nonuniform CAs. If the user does not confine the search space to uniform CAs, the evolutionary process is free to select the number of potential rule sets that can be applied to a cell, from one (i.e., a uniform CA) to a user-defined maximum. The evolutionary process also selects which rule set is applied to each cell in the CA.

A CA evolved using EvoCA performs a computation as follows. The CA is initialized using the evolvable initial-value vector stored in its genome (see below and Fig. 1). For each cell that is linked to an input, its initial value is overwritten by the current value of its corresponding input. Thus if, for example, our computation takes one input and we set the CA to have five cells in total, then the cell assigned to the lone input will have its initial value defined by the value of the input, while the other four cells will have their initial values determined by the initial-value vector in the CA's genome. Once the CA is initialized in this fashion, it is run for a user-defined number of iterations, with each cell (including those designated as input and output cells) acting as "normal" cells, updating in parallel using the evolvable rule set assigned to them in the genome. At the end of the fixed number of CA iterations, the current values of the designated output cells are taken as the outputs of the computation. Example computations can be found in Fig. 2.

An island model is used as part of the canonical EvoCA algorithm. Island models enhance the canonical single-population GA to evolve multiple reproductively isolated populations in parallel, with periodic exchanges of genomes between islands through "migration." Implementing island models is straightforward, adds little computational overhead, and has been shown to significantly improve GA performance (see Cohoon et al., 1987; Grouchy et al., 2009).

Users must define several parameters before beginning the evolutionary process. A radius r must be provided, as well as the size of the CA and the number of CA iterations per computation. Users must also provide a maximum number of rule sets, where choosing 1 forces the algorithm to search only the space of uniform CAs. Besides these parameters that apply to the CAs themselves, a variety of typical GA

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alif2016/28/84/2325865/978-0-262-33936-0-ch021.pdf by guest on 14 August 2024

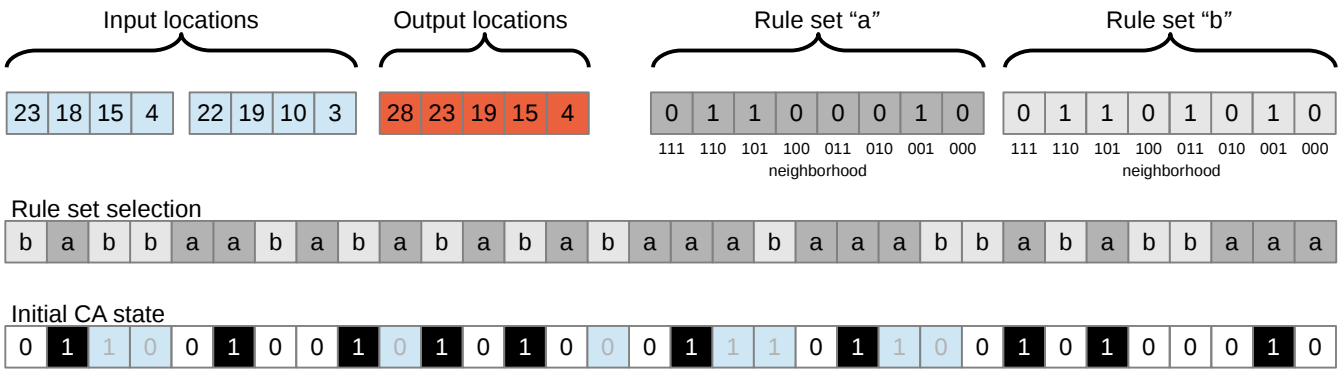

Figure 1: An example EvoCA genome that encodes a 32-cell CA with 8 inputs and 5 outputs. This genome also has two rule sets that differ by a single rule. This set of rule sets is one of the two most frequent sets to appear in discovered solutions. Input cell locations are highlighted in light blue on the initial CA state vector, indicating that the encoded initial values for these cells (shown in light gray) do not affect the CA's behavior. This genome encodes the 4-bit adder whose example behaviors are shown in Fig 2.

parameters must also be set (see below).

The algorithm presented here evolves 1D CAs with $k = 2$ and periodic boundary conditions, although extending EvoCA to evolve other types of CAs is possible.

Genome

We may regard a CA α as a map $\alpha : A^L \rightarrow A^L$, where A is the alphabet of states on which the CA operates (for binary CAs $A = \{0, 1\}$) and L is the length of the CA. The cardinality of A, $|A|$, is k and the radius of operation for each cell is set to a constant r, as described previously. However, the actual function of interest that we wish to evolve may be expressed as $\rho_\alpha : A^{L_I} \rightarrow A^{L_\Omega}$, where I represents the subset of cells from α that serve as inputs and Ω the subset of cells that serve as outputs; $|I| = L_I \leq L$ is accordingly the number of desired inputs and $|\Omega| = L_\Omega \leq L$ the number of desired outputs. In the present embodiment of this concept, the alphabet A, the radius r, the length of the CA L, and input and output sizes L_I and L_Ω are fixed. The location of input and output cells are mutable although their numbers are not.

Figure 1 shows an example EvoCA genome that encodes a 32-cell CA for a computation requiring 8 inputs and 5 outputs (in fact, this genome encodes a 4-bit adder CA, see below). There is one gene for each input and each output, with each of these genes containing the position of its associated cell represented as an integer in the range $[1, L]$. The genome also contains one or more rule sets, each of size 2^{2r+1}. Owing to the fact that additional rule sets can be added via mutation (see below), genomes in EvoCA are variable-length. To determine which rule set to apply for each cell (in the case of nonuniform CAs), genomes also contain a vector of genes of length L, where each gene corresponds to a cell in the CA and contains the unique identification tag of the rule set to be applied to that cell. Finally, genomes contain a second vector of genes of length L containing initial values (either 0 or 1 for the work presented here) for each cell. Note that the genes in this initial-value vector that correspond to input cells are “neutral,” meaning that they do not affect the behavior of the CA in any way. This is owing to the fact that these initial values get overwritten by input values before the initial iteration of the CA.

Initialization. Before the evolutionary process can begin (at generation 0), an initial population of genomes is needed. These genomes are generated randomly as follows: Input cells are randomly chosen one at a time without replacement from the set of all CA cells. Output cells are chosen in the same fashion. Initial genomes start with only a single rule set, as this was found to produce the best results (see Table 2). To generate a random rule set, a single value λ is chosen at random from the uniform distribution $[0, 1]$. Then for each rule in the rule set, a random value β is drawn from the same distribution as λ. If $\beta < \lambda$, then the current rule will be set to a 1, otherwise it will be set to 0. Using this λ term produces populations of genomes whose rule sets are uniformly distributed across different densities of 1s, as in (Sipper, 1996).

Since genomes are initialized with only a single rule set, the vector in the genome responsible for assigning rule sets to cells will be initialized with each gene pointing to the same initial rule set. However, in experiments where the number of initial rule sets is allowed to vary, a rule set is chosen at random (with uniform probability) for each cell.

Finally, the initial-value vector is randomly initialized in the same fashion as the initial rule set, i.e., using a randomly selected λ term to determine the expected density of 1s.

Reproduction. To produce the next generation of CA genomes (offspring) from the current (parent) generation,

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alif2016/28/84/2325865/978-0-262-33936-0-ch021.pdf by guest on 14 August 2024

relatively high fitness parent genomes are cloned. If crossover is to be used during the creation of an offspring genome, a second high-fitness parent genome is selected and merged with the offspring genome that was cloned from the first parent. The original offspring genome is preserved from its beginning (i.e., the top-left gene in Fig 1) to a randomly selected crossover point on its genome (moving from left to right and top to bottom along the various parts of the genome, as organized in Fig 1). From the crossover point onwards, the remaining genes on the offspring genome are replaced with their corresponding genes from the second parent's genome. Note that if the offspring genome contains more rule sets than the second parent's genome, its additional rule sets will be preserved regardless of the location of the crossover point. Conversely, if the second parent's genome contains more rule sets than the offspring genome, its additional rule sets will be ignored during crossover. Finally, when copying an input gene from the second parent to the offspring genome, if it is found that the second parent's gene points to a cell that is already associated with a previous input gene on the offspring genome, crossover does not occur for the gene in question, leaving the offspring's original gene unmodified. This rule is also applied when copying output genes.

Mutations. The resulting offspring genomes (either asexually cloned or generated via sexual reproduction, i.e., crossover) are then subject to one or more of a variety of potential mutations:

- Rules in an offspring genome's one or more rule sets can be modified by a bit-flip mutation.
- If the genome does not already contain the user-defined maximum number of rule sets, a new rule set can be created through a mutation. Rules in the new rule set are randomly generated as when rule sets are first initialized at generation 0 (see above).
- Input genes can be assigned to a new, randomly selected CA cell that is not already designed as an input cell. A mutation can also cause two input genes to swap associated cells. Finally, a mutation can cause an input gene to become associated with a cell adjacent to the cell that it is currently associated with. If the adjacent cell is already designated as an input, the next adjacent cell is selected instead (this process will repeat until a cell that is not currently associated with an input is found).
- Output genes are subject to the same three types of mutation as input genes.
- All genes in the initial-value vector are subject to bit-flip mutations.
- If a genome contains two or more rule sets, each gene that determines which rule set to apply to its associated cell is subject to a mutation that randomly selects a new rule set to govern its cell's behavior.
- If a genome contains two or more rule sets and at least one of these rule sets is not associated with any cells, the genome can undergo a mutation that removes all rule sets that are not in use.

Mutations to initial-value genes that are associated with cells designated as inputs are "neutral" in the sense that they will not modify the behavior of the CA. This is again owing to the fact that the initial values of input cells are overwritten by their associated input's value. These neutral genes may be expressed in future generations however, as a mutation may change the cell associated with an input, causing the former input cell to be initialized to its previously neutral initial-value gene at the beginning of a CA computation.

It should be noted that a mutation that adds a rule set to a EvoCA genome does not force it to be used by any of the CA's cells, and thus is also a neutral mutation. However, a future mutation may associate one or more cells with this new rule set, thus expressing the originally neutral mutation. Genomes with two or more rule sets may also end up with neutral rule sets through the accumulation of mutations that disassociate its cells from a previously used rule set (by associating them with other rule sets in the genome). When a rule set is not associated with any cells, its rules are still subject to bit-flip mutations. However, since these rules are not used by the CA, these mutations will also be neutral, perhaps getting expressed in future generations if genomes evolve to (re)use this rule set.

4-bit Adder Experiments

To demonstrate the capabilities of EvoCA, we attempt to evolve CAs that function as 4-bit adders[1]. This requires 8 inputs to the CA (the two 4-bit strings to be summed) and 5 outputs (the resulting 4-bit sum and the final carry bit). This problem is interesting in that at least four computational operations must be done *in the correct order* before a correct answer can be produced. Furthermore, to allow carry bits from previous operations to be passed to subsequent operations, a solution to this problem will necessitate some form of memory.

For all of the experiments described here, a minimal radius of $r = 1$ is used: Each cell has access to only its own state and that of its two adjacent neighbors. Each CA in the population is evaluated on all possible combinations of inputs ($2^8 = 256$ total training cases). For each input string, the CA being evaluated has the Hamming distance between its 5-bit output string and the correct 5-bit answer added to its fitness, which is initially set to 0. This is therefore a minimization problem, and a 4-bit adder CA will have a fitness of 0.

[1]The source code for these experiments is freely available at `https://github.com/pgrouchy/EvoCA`

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alif2016/28/84/2325865/978-0-262-33936-0-ch021.pdf by guest on 14 August 2024

We use 100 islands arranged in a ring, with each island having a population of 100 CAs, for a total population size of 10,000 CAs per run of EvoCA. Islands evolve in parallel and are reproductively isolated, except for occasional migration events. In the experiments presented here, migration is implemented as follows: islands receive 5 randomly selected genomes from the island to their right every 10 generations and these 5 incoming genomes overwrite randomly selected preexisting genomes in the population.

On each island, tournament selection, where the CA genome with the best fitness (smallest Hamming distance) is selected from a group of randomly chosen CA genomes, is used to select parents for reproduction. A tournament size of 10 is used for all experiments in this paper. To produce an offspring genome, a parent genome selected via tournament selection is cloned. If crossover is to be applied, tournament selection is run a second time to select a second parent whose genome will be spliced with the offspring genome. Offspring genomes are also subject to a variety of mutations (see above). The current top CA on an island is copied into the offspring population without modification (elitism). Otherwise, crossover is applied with a probability of p_c and each gene of offspring i is subject to mutation with a probability of $1/S_i$, where S_i is the size of the offspring's genome. There is a 1% chance that all unused rule sets (i.e., a rule set that is not applied to any of the CA's cells) will be removed from an offspring genome.

A variety of experiments are run to investigate the influence of various parameters of EvoCA on its performance on this problem. For each parameter configuration, 100 runs with varying initial populations are performed and all runs last for 1,000 generations.

Results and Discussion

The results from all performed experiments are summarized in Tables 1, 2, and 3. The fitness value of a given run is reported as the lowest fitness achieved by any genome throughout the entire evolutionary run. P values are calculated using the two-tailed Wilcoxon rank-sum test. Note that preliminary experiments (see Table 3) determined that EvoCA performs better without crossover enabled (i.e., $p_c = 0$), therefore all experiments reported here are with $p_c = 0$, except where noted.

The results in Table 1 demonstrate the need for additional cells beyond the number of inputs. When L is equal to the number of inputs (i.e., $L = 8$), EvoCA is unable to find a CA that can correctly compute the addition of every possible combination of two 4-bit strings. Furthermore, increasing the number of CA iterations has no effect on performance. By increasing L beyond the number of inputs however, significant performance improvements are achieved and EvoCA is able to discover nonuniform CAs that are 4-bit adders (no solutions employing uniform CAs emerged from any of the reported experiments). These results also

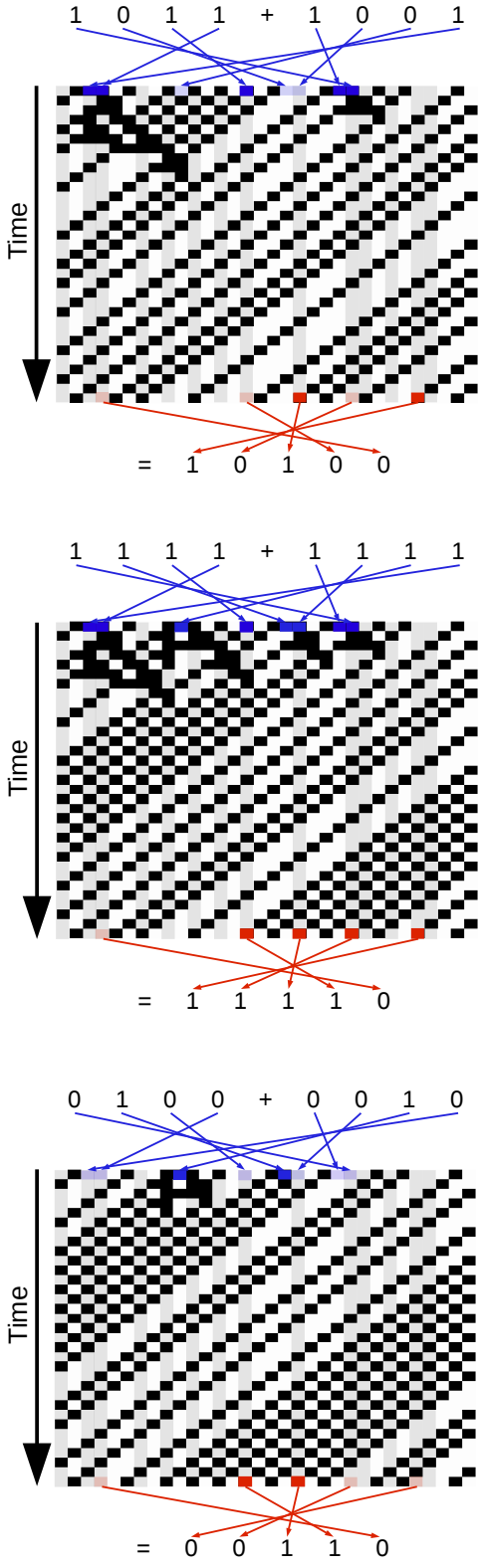

Figure 2: Three different computations from the successfully evolved 4-bit adder CA described by the genome in Fig 1. Cells whose states are updated using rule set "a" are white, while cells whose states are updated using rule set "b" are gray.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alif2016/28/84/2325865/978-0-262-33936-0-ch021.pdf by guest on 14 August 2024

CA size	# CA iterations	# success	Fitness μ	Fitness σ	Rank-sum
8	8	0	158.43	41.06	$P > 0.999$
8	32	0	156.16	38.37	$P < 0.001$
24	24	0	70.36	52.51	$P < 0.020$
24	32	1	51.77	42.92	$P < 0.015$
32	24	11	80.76	62.95	$P < 0.060$
32	32	21	64.09	59.50	$P < 0.017$
32	40	10	82.31	61.94	$P > 0.965$
40	40	11	87.12	60.67	$P < 0.016$
40	32	6	108.51	53.79	

Table 1: Summary of the results from experiments where the CA size and the number of CA iterations were varied. Mean and standard deviation are labeled as μ and σ respectively. All runs are mutation-only (i.e., $p_c = 0$) and use the island model described in the main text.

show that there is a point at which adding additional cells no longer affects performance (e.g., when going from 32 cells to 40 cells), while continuing to increase computational costs.

Why would EvoCA only be able to fully solve this problem using CAs with more cells than inputs? One reason could be that with only 8 cells, the GA gets stuck in local minima. By adding additional cells, the GA can exploit added dimensions in the search space to escape such minima. Another possibility is that additional cells beyond the number of inputs allow EvoCA to better control the order and timing of key computations during the CA iterations, e.g., to control when the carry bit from the addition of the least significant input bits is added into the addition of the next two input bits. Lending evidence to this hypothesis is the fact that in all successful results one finds that while two inputs representing the same bit from the two 4-bit input strings (e.g., one input is the most significant bit from the first input string and the second input is the most significant bit from the second input string) are often found associated with adjoining cells, there are always at least two noninput cells separating inputs from different locations in the bit strings (e.g., Figs. 1, 2, and 4). Since $r = 1$ in all experiments, this separation ensures that cells associated with different bit locations in the input strings will not influence each other's initial state updates. Moreover, by having additional cells beyond the number of inputs, the EvoCA algorithm can exert greater control over which rules get applied at the initial iteration via the evolvable initial values. It is likely that all of the aforementioned hypotheses contribute to EvoCA's success when the number of CA cells is greater than the number of inputs, although further experimentation is required before conclusions can be drawn.

The data in Table 1 also yield no obvious rule for choosing the number of CA iterations to maximize EvoCA performance given a specific CA size. This indicates that future versions of EvoCA should consider allowing the number of iterations to be evolvable, alongside the other parameters already incorporated into the genome.

The experiments summarized in Table 2 explore EvoCA performance with and without restrictions on the use of nonuniform CAs. The top two rows show the results from experiments where nonuniform CAs are allowed to evolve, the difference being that the data in the first row are from experiments where populations were initialized with uniform CAs only, while the data in the second row are from experiments where populations are initialized with a variety of uniform and nonuniform CAs. The third row of data are from experiments where only uniform CAs could be evolved. These results clearly demonstrate that allowing nonuniform CAs to evolve is necessary for the success of the EvoCA algorithm on this task. Furthermore, significant performance improvements are achieved by restricting the initial population of CAs to be uniform only. This is to be expected, however, as additional rule sets increase the size

CA type	# init. rule sets	# success	Fitness μ	Fitness σ	Rank-sum
n-u	1	21	64.09	59.50	$P < 0.001$
n-u	[1-8]	2	129.20	45.27	$P < 0.001$
u	1	0	175.18	46.43	

Table 2: Summary of the results from experiments that explored GA performance with and without restrictions on the use of nonuniform CAs. Note that the top-most row of data are reproduced from Table 1 for comparison. Mean and standard deviation are labeled as μ and σ respectively. Experiments with a CA type of "n-u" allow both uniform and nonuniform CAs to evolve, while experiments with a CA type of "u" are restricted to uniform CAs only. All runs are mutation-only (i.e., $p_c = 0$) and use the island model described in the main text.

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alif2016/28/84/2325865/978-0-262-33936-0-ch021.pdf by guest on 14 August 2024

p_c	Island model	# success	Fitness μ	Fitness σ	Rank-sum
0.0	yes	21	64.09	59.50	
					$P < 0.001$
0.3	yes	3	96.56	58.40	
					$P > 0.853$
0.7	yes	8	96.84	66.02	
					$P < 0.001$
0.7	no	1	203.11	65.14	
					$P < 0.003$
0.0	no	1	174.41	58.09	

Table 3: Summary of the results from experiments where the CA size and the number of CA iterations were both fixed at 32, while the percent of offspring genomes created using crossover p_c and the type of GA used (island model or canonical) are varied. Note that the top-most row of data are reproduced from Table 1 for comparison. Mean and standard deviation are labeled as μ and σ respectively.

of the genome, and thus the size of the search space (e.g., Stanley and Miikkulainen, 2002).

The experiments summarized in Table 3 explore EvoCA performance when varying whether or not an island model and/or crossover are used. These results demonstrate that significant performance improvements are achieved using the island model described here. Surprisingly, these results also show that using the crossover method described here significantly *reduces* performance, and thus best results are achieved using mutation-only (i.e., $p_c = 0$) EvoCA. There are many possible reasons for such a result. Perhaps the single-point crossover mechanism as described needs to be

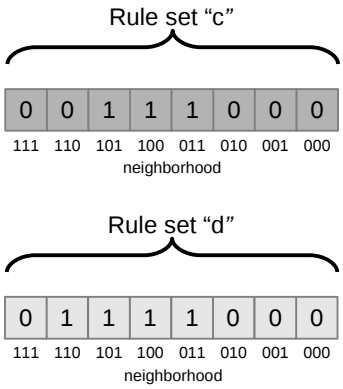

Figure 3: This set of rule sets and the one shown in Fig. 1 are the two most common sets among the 4-bit adder CAs discovered by EvoCA. An example behavior of a solution that employs these rule sets is shown in Fig. 4.

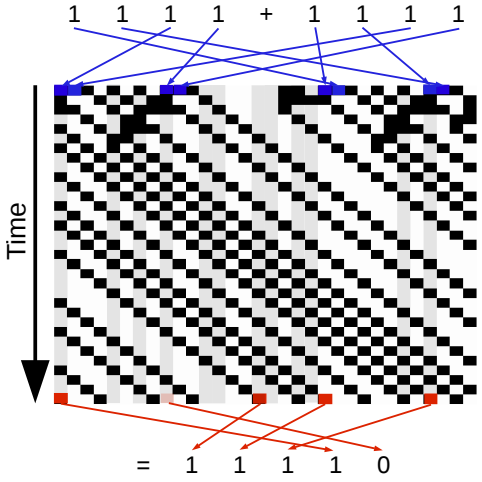

Figure 4: An example behavior from an evolved 4-bit adder that employs the set of rule sets shown in Fig. 3. Cells whose states are updated using rule set “c” are white, while cells whose states are updated using rule set “d” are gray.

refined. Or perhaps single-point crossover works better with a different arrangement of genes in the genome than the one used here (see Fig. 1). Better results might also be achieved by developing an alternative method of crossover, perhaps based on uniform crossover (where crossover is applied at each gene independently, thus the arrangement of genes in the genome has no effect on performance). Finally, it is possible that this problem domain, or even the search spaces engendered by the EvoCA algorithm in general, are best searched with mutation-only GAs. Again, conclusions cannot be drawn without further experimentation.

While the positions of the input and output cells vary considerably between solutions (which is to be expected considering the CA has periodic boundary conditions and addition is commutative), 85.3% of all successfully evolved 4-bit adders contain one of two sets of rule sets: the two rule sets shown in Fig. 1, and the two rule sets shown in Fig. 3. A sample behavior from a solution that employs the set of rule sets shown in Fig. 3 is shown in Fig. 4. Two types of solutions with three rule sets were also discovered, as well as an additional solution with two rule sets for a 40-cell CA. No uniform solutions or solutions with four or more rule sets emerged.

Conclusions and Future Work

We have presented EvoCA, a novel genetic algorithm for evolving both uniform and nonuniform CAs to perform user-defined computations. This algorithm allows the size of the CA to be greater than the number of inputs, a necessity when searching for solutions to the 4-bit adder problem described here, and contrary to previous work. Furthermore, the number of outputs is free to vary from 1 to the size of the CA,

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alif2016/28/84/2325865/978-0-262-33936-0-ch021.pdf by guest on 14 August 2024

independent of the number of inputs. The canonical version of EvoCA uses an island model to significantly improve the performance of the GA, although the presented crossover mechanism was found to be significantly detrimental to performance. Thus, future work should look to further investigate crossover mechanisms. Another avenue of research is to explore the trade-offs between increased computational complexity and increased performance when incorporating GA variants such as novelty search (Lehman and Stanley, 2008) and speciating GAs (e.g., Grouchy et al., 2009) into EvoCA. Of course, EvoCA has only proven itself on a single problem thus far, therefore future work should apply EvoCA to many other problems.

Finally, the EvoCA runs presented here were computationally intensive, with each run testing 10,000 CAs on 256 different training cases per generation, for 1,000 generations. Thus an important step towards applying EvoCA to more challenging problems will be to implement it on GPUs, something that should be relatively straightforward owing to the inherently parallel nature of both genetic algorithms and CAs (Žaloudek et al., 2010).

References

Barricelli, N. A. (1963). Numerical testing of evolution theories. *Acta Biotheoretica*, 16(3-4):99–126.

Berlekamp, E. R., Conway, J. H., and Guy, R. K. (2004). Winning ways for your mathematical plays.

Cenek, M. and Mitchell, M. (2009). Evolving cellular automata. *Encyclopedia of Complexity and Systems Science*, pages 3233–3242.

Cohoon, J. P., Hegde, S. U., Martin, W. N., and Richards, D. (1987). Punctuated equilibria: a parallel genetic algorithm. In *Proceedings of the second international conference on genetic algorithms and their application*, pages 148–154. L. Erlbaum Associates Inc.

Cook, M. (2004). Universality in elementary cellular automata. *Complex systems*, 15(1):1–40.

Darabos, C., Mackenzien, C. O., Tomassini, M., Giacobini, M., and Moore, J. H. (2013). Cellular automata coevolution of update functions and topologies: A tradeoff between accuracy and speed. In *Advances in Artificial Life, ECAL*, volume 12, pages 340–347.

Grouchy, P., Thangavelautham, J., and D'Eleuterio, G. M. (2009). An island model for high-dimensional genomes using phylogenetic speciation and species barcoding. In *GECCO'09*, pages 1355–1362. ACM.

Guan, S.-U. and Zhang, S. (2003). An evolutionary approach to the design of controllable cellular automata structure for random number generation. *Evolutionary Computation, IEEE Transactions on*, 7(1):23–36.

Iclănzan, D., Gog, A., and Chira, C. (2011). Enhancing the computational mechanics of cellular automata. In *Nature Inspired Cooperative Strategies for Optimization (NICSO 2011)*, pages 267–283. Springer.

Lehman, J. and Stanley, K. O. (2008). Exploiting open-endedness to solve problems through the search for novelty. In *ALIFE*, pages 329–336.

Mitchell, M., Crutchfield, J. P., Das, R., et al. (1996). Evolving cellular automata with genetic algorithms: A review of recent work. In *EvCA'96*. Moscow.

Mitchell, M., Hraber, P., and Crutchfield, J. P. (1993). Revisiting the edge of chaos: Evolving cellular automata to perform computations. *arXiv preprint adap-org/9303003*.

Oliveira, G. M., Martins, L. G., de Carvalho, L. B., and Fynn, E. (2009). Some investigations about synchronization and density classification tasks in one-dimensional and two-dimensional cellular automata rule spaces. *Electronic Notes in Theoretical Computer Science*, 252:121–142.

Packard, N. H. (1988). *Adaptation toward the edge of chaos*. University of Illinois at Urbana-Champaign, Center for Complex Systems Research.

Sapin, E., Bailleux, O., and Chabrier, J.-J. (2004). Research of complex forms in cellular automata by evolutionary algorithms. In *Artificial Evolution*, pages 357–367. Springer.

Sapin, E., Bull, L., and Adamatzky, A. (2009). Genetic approaches to search for computing patterns in cellular automata. *Computational Intelligence Magazine, IEEE*, 4(3):20–28.

Sipper, M. (1996). Co-evolving non-uniform cellular automata to perform computations. *Physica D: Nonlinear Phenomena*, 92(3):193–208.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127.

Von Neumann, J., Burks, A. W., et al. (1966). Theory of self-reproducing automata. *IEEE Transactions on Neural Networks*, 5(1):3–14.

Wolfram, S. (1984). Cellular automata as models of complexity. *Nature*, 311(5985):419–424.

Žaloudek, L., Sekanina, L., and Šimek, V. (2010). Accelerating cellular automata evolution on graphics processing units. *International Journal on Advances in Software*, 3(1 & 2).

Downloaded from http://direct.mit.edu/isal/proceedings-pdf/alif2016/28/84/2325865/978-0-262-33936-0-ch021.pdf by guest on 14 August 2024