

Evolutionary Methods for 2-D Cellular Automata Computation

Samuel Inverso Daniel Kunkle Chadd Merrigan

May 21, 2002

Abstract

This paper describes methods for evolving 2-D cellular automata to perform global computations. This is a difficult task because global behaviors must arise from local computations of many parallel cells. We present the results of numerous tests involving different genetic algorithm methods to perform the 2-D equivalent of classic 1-D CA tasks, including *density classification* and *synchronization*, and our own 2-D CA *balanced surface minimization* task. The performance of the GA was improved greatly by the use of totalistic CA rule tables, increasing the fidelity of fitness functions, and with coevolutionary techniques.

1 Introduction

Emergent behaviors of simple individuals to perform complex patterns are a basic property of cellular automata (CA). This property is represented in Conway's Game of Life and solutions to density classification and synchronization found by Melanie Mitchell [1]. Because complex behaviors are formed by simple rules represented by bit strings, a genetic algorithm is particularly suited to developing the rules. For the experiments performed, an adaptation of Peter Anderson's genetic algorithm program evolved density classification, synchronization, and balance surface minimization rules for a two state, two dimensional cellular automata.

The density classification, synchronization, and balanced surface minimization problems for 2D cellular automata differ significantly from the traditional 1D CA approach. In a one dimensional CA, the cells only consider

neighbors to their left and right. While in a 2D CA the cells must use rules that account for neighbors above, below, left, and right of itself. If it does not, the CA will only solve the problem along rows and columns causing a striped state matrix. The spatial increase in the neighborhood also increases the complexity of the CA implementation because the environment changes from a ring to a torus, where the top and bottom edges are connected and the left and right edges are connected in the state matrix.

2 Design

2.1 Neighborhood

A cellular automaton's neighborhood determines cells' update rules and consequently has a major impact on the Genetic Algorithm used to find those rules. In an n neighborhood CA the genome's length is 2^n , which makes the search space 2^{2^n} . For these experiments a neighborhood of nine was selected to increase the knowledge of an individual cell because it knows about all of its neighbors, Figure 1 depicts this. A neighborhood of nine yields a genome length of 512 and search space of 2^{512} , or 1.34×10^{154} .

Figure 1 also depicts other possible neighborhoods. In future work, the neighborhoods between three and nine should be experimented with to determine the best balance between cell knowledge and computational complexity. If classification can be achieved with the next smallest neighborhood size, five, the search space will be reduced from 2^{512} to 2^{32} , a massive 2^{480} times decrease in the search space. A neighborhood of two should not be used because the cell only will know about its row or column which is equivalent to running many one dimensional cellular automata in parallel.

2.2 Density Classification

With D representing a percentage, in density classification the CA starts with a board that consists of D% ones and 100-D% zeros, e.g. 70% ones 30% zeros abbreviated 70-30, and forms a board with either all ones or all zeros depending on which had the higher initial density. For example, if the CA's initial state was 70-30 there are more 1's than 0's so the CA should become 100% 1's and remain in that state. Likewise, if the initial state was 30-70 the CA should become all 0's, Figure 2.

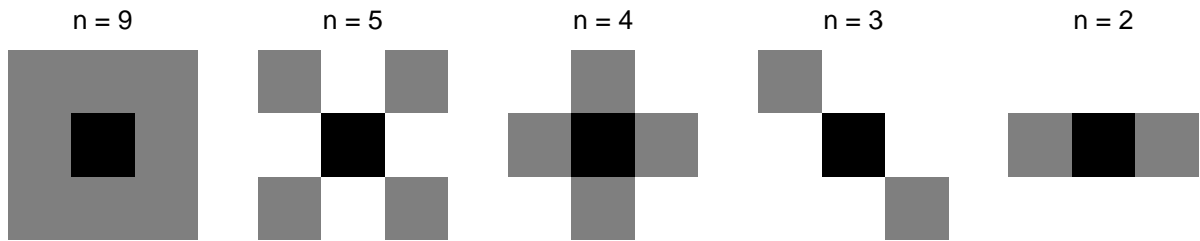


Figure 1: Depictions of Neighborhood Sizes (n). The black square is the cell the rule applies to. Gray squares are the active neighbor cells used to determine the new value.

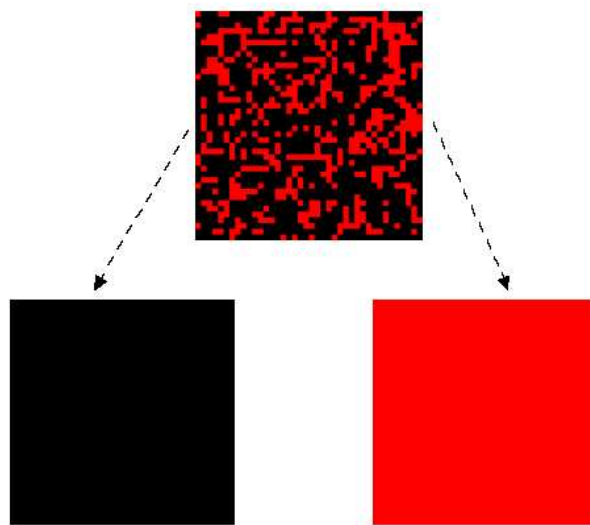


Figure 2: Illustration of a random board and density classification. Dashed arrows indicated the CA moves to either an all one or zero state based on the original density.

To determine the fitness of a density classifying genome the cellular automata was run twice by the GA, once with more 1's than 0's (sum of 1's), then again with more 0's than 1's (sum of 0's). This was done so the GA did not find a rule that always turned the states to either all 1's or all 0's regardless of density. Equation 1 shows the fitness function. In a nine by nine CA, the number of cells equals 81; therefore a 100% fitness is a GA with

a hero equal to 162.

$$\text{Density fitness function} = \text{sum of 1's} + \text{sum of 0's} \quad (1)$$

2.3 Synchronization

In synchronization the CA starts with a random board, turns the board to all 1's or 0's, then oscillates between all 1's and 0's every time step, Figure 3. To determine the fitness of a synchronizing genome the CA is run for a number of time steps. The CA cells' values are summed, sum_1 , and the CA is run for another step, and the cells' values are summed again, sum_2 . The CA is run one last time and the cells' values are summed sum_3 .

$$\text{Synchrony fitness function} = |sum_1 - sum_2| + |sum_2 - sum_3| \quad (2)$$

As we see from the Equation 2, maximum fitness occurs when the CA oscillates between 1's and 0's or 0's and 1's across the three time interval check.

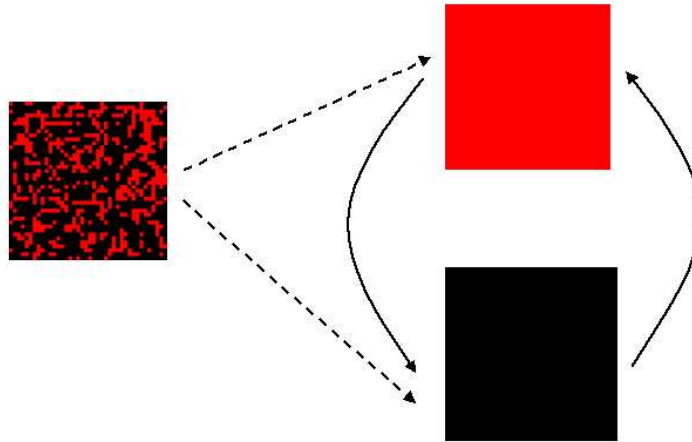


Figure 3: Illustration of a random board synchronizing and oscillating between homogeneous states.

2.4 Balanced Surface Minimization

Balanced surface minimization is a multi-objective problem where the CA must contain the same number of 1's and 0's (balance), and homogeneous collections of 1's and 0's where the number of cells with adjacent neighbors that have different values is minimized (surface minimization). In a 2D CA cells have eight adjacent neighbors: north, northeast, east, southeast, west, south, southwest, and northwest. These objectives are in opposition because the most minimal surface has all ones or zeros while the best balance has 50-50 ones and zeros.

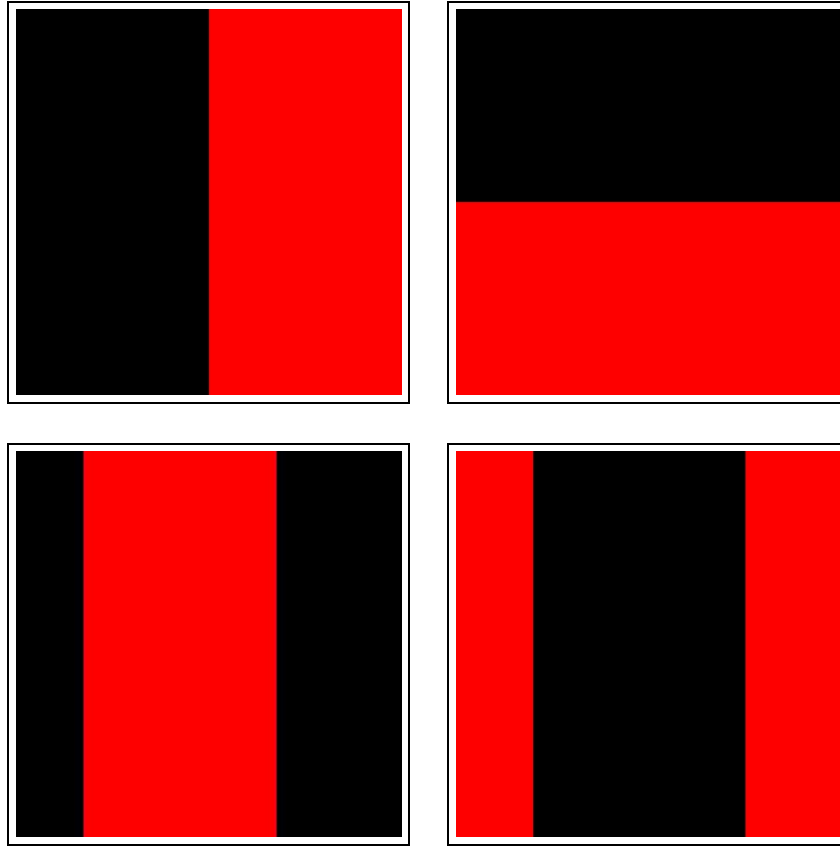


Figure 4: Examples of optimal balanced surface minimized cellular automata.

Figure 4 shows optimal balance surface minimizations where there are two homogeneous collections of 1's and 0's (remembering that the CA board is

toroidal). In these CAs there are equal numbers of 1's and 0's, corresponding to the red and black pixels, and the majority of cells have neighbors of equal value. At worst, a cell has three adjacent neighbors with a value different from its own: the cells on the edge of the homogeneous collections. Figure 5 shows a suboptimal surface minimization where the cells always have four adjacent neighbors of different value. The worst configuration of cells for surface minimization is a striped pattern, where each stripe has a cell width of one. In this configuration, the cells have six neighbors of different value.

Equation 4 is the GA fitness function to find a balanced surface minimizing genome.

$$\begin{aligned}
 & \textit{Balanced Surface Minimization Fitness Function} && (3) \\
 & \text{if } sum > num_cells * 2 \text{ then } sum = num_cells - sum \text{ end} \\
 & \textit{fitness} = sum * weight + (6 * num_cells - num_edges)
 \end{aligned}$$

The fitness function first sums all the cells in the CA. Because an optimal balance is 50-50 ones and zeros, a higher fitness is given to 50-50 by using the sum if it is less than half the number of cells, or number of cells minus the sum, if the sum is more than half the number of cells.

The surface minimization is a function of the worst balanced surface, six adjacent neighbors with different values, times the number of cells minus the number of edges, where an edge is defined as two adjacent cells with different values. This produces a large number when the surface is minimized because there are few edges to subtract from $6 * num_cells$. The fitness function compensates for the opposition between balance and surface minimization through a weighting factor on the balance fitness, a weight of eight was used for all experiments. The weight avoids the problem of the surface minimization fitness becoming larger than the balance fitness causing the GA to produce rules that tend to favor surface minimization over balance.

2.5 Implementation

The two-dimensional cellular automata used was designed to be generic, reusable, and fast. A 2D CA was initially developed in MATLAB, but it was apparent the implementation was too slow to run a reasonable range of experiments. The CA was then written in C++, which significantly increased

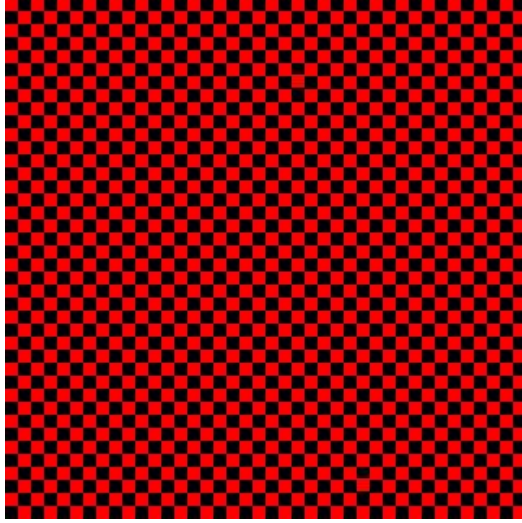


Figure 5: Suboptimal surface minimization with high balance. Each cell has four neighbors of different value.

the performance. For example, the MATLAB CA performed 300 iterations on a twenty by twenty state 2D CA with neighborhood size nine in 50 to 60 seconds, while the same configuration for the C++ CA finished in less than one second.

The C++ CA also provides a view interface, *CAHistory.h*, which allows a program to monitor every state change as it occurs. This is useful in many ways, such as visualization of the CA during processing, maintaining a history of the state after every processing step, and performing calculations on new states. In addition, helper methods were created to easily operate on the CA.

A *traverse* method was created to iterate over the state matrix. *Traverse* takes a method as a parameter and passes the the value of each cell to that method as it traverses the matrix. This simplifies code and reduces coding time because a nested *for* loop does not need to be written for every traversal. Similarly, the method *setStates* takes a method as a parameter and sets the return value of that method to every cell in the state matrix.

One drawback to the C++ implementation of the cellular automata was the genetic algorithm program *ga* needed to be converted to C++. To accomplish this in the period of time allotted to finish this assignment, *ga* was converted in utility only, and does not have true Object-Oriented function-

ality.

With the transition from C to C++ new functionality was added to the GA program including the ability to set the running duration in seconds and a parameter that forces the GA to use the current time as a seed instead of a constant.

If the duration is set the GA will stop in that number of seconds. This helps in running experiments because the duration of an iteration does not need to be determined by experimenters that want to run the GA for a certain period of time instead of a number of iterations.

3 Basic Genetic Algorithm

A basic bit-string GA is used as a first attempt to solve the problems of density classification, synchronization, and balanced surface minimization. It is a steady state GA (population size remains constant) based in part on an implementation by Peter Anderson. A population of possible solutions, in this case the CA rule tables, are evaluated for fitness and recombined using genetic operators in a search for optimal solutions.

3.1 Parameters

The following is a list of some important parameters used by the genetic algorithm, the effect they have on the GA, and a description of the values chosen.

- **Population size:** The number of individuals maintained as possible solutions by the GA at any time. A value of 200 was used for most runs.
- **Tournament size:** In choosing parents from which children will be created by genetic operators, two tournaments are held, the winner of each is a parent and the loser of each is replaced by a child. The tournament size describes the number of random individuals that are chosen from the population for each tournament. For all reported experiments tournament size was set at 2.
- **Crossover Type:** Genetic crossover can be set to one-point, two-point, or uniform. For all reported experiments one-point crossover was chosen.

- **Mutation Rate:** The probability that a new child will undergo a mutation, as described by the *mutation effect*. A value of 0.5 was used for experiments reported here.
- **Mutation Effect:** The probability that any bit in a child selected for mutation will be flipped. For example, if mutation rate is 0.5 and mutation effect is 0.05, on average half all new children will have 5% of their bits flipped. A value of 0.05 was used for experiments reported here.
- **Duration:** The amount of actual processing time the GA was allowed to evolve solutions for. This was set to 30 minutes for most experiments.
- **Loops:** An alternate description of the length of time the GA is allowed to evolve solutions for. In this case, it describes the number of child-creation cycles allowed. With n loops, $(2 \times n + \text{population size})$ fitness evaluations will be done. This was usually set to 0, representing no limit on the number of loops, to allow the GA to run for the specified time duration.
- **CA Rows and CA Columns:** The number of rows and columns of the CA state matrix. For most experiments reported here this value was 15×15 , though many other sizes were used in testing.
- **CA Neighborhood Size:** The neighborhood size determines the neighborhood pattern discussed earlier. This was set to 9, corresponding to the center cell and the eight cells adjacent by a chess king's move.
- **CA Iterations:** The number of iterations the cellular automata runs for. A value of 300 was used for all experiments.
- **CA Initial Maximum Density:** The CA state matrix's initial density. Values are specified for the maximum percentage of ones, e.g. 70 creates a table with 70% ones. For density classification values 60, 70, and 80 were used. For synchronization and balanced surface minimization the density was kept at 50%.

3.2 Basic GA Results

The initial results from the basic GA were disappointing for each of the three problems when the board size was significant (dimensions 15×15) and the time allowed for evolution was reasonable (30 minutes). For small CAs or very long evolution times solutions could be found by the GA. Improvements to the GA described later performed much better with large board sizes and short evolution times.

Figure 6 shows the number of fitness evaluations, time spent, and heros (best fitness found) for the basic GA for each of the three problems.

Figure 7 shows the results of testing the best genome found for each problem compared to random rule tables. To achieve the random results, 1500 random individuals, or rule tables, were generated and each was evaluated for a different random CAs. All reported fitness values are normalized to the range $[0, 100]$. The best found genomes were then tested against their own set of 1500 random CAs. These results show no significant improvement of the best found genome of the GA over random genomes.

Method	Number of		
	Fitness Evals	Time (min)	Hero
Density Classification	4008	30.0	59.1
Synchronization	7860	30.0	23.77
Balanced Surface Minimization	7920	30.0	70.2

Figure 6: Evolution times and heros for basic GA method for three CA problems.

4 Totalistic Rules

Normal full rule tables for CAs specify the next state of a cell based on the state of each of that cell's neighbors. With s possible states and n neighbors the rule table length is s^n bits. In this case, with 2 states and 9 neighbors for each cell, the rule table (and the genome used by the GA) is 2^9 , or 512, bits long. With each update, a cell computes the states of its neighbors, $x_1x_2x_3x_4x_5x_6x_7x_8x_9$, and uses this binary number as a lookup into the rule table. It is this type of rule table that most research on constructing

Method	Mean	Minimum	Maximum
Density Classification - Random	50.0(± 2.3)	41.6	57.3
Density Classification - GA	50.0(± 2.4)	43.3	58.2
Synchronization - Random	3.8(± 2.3)	0	14.7
Synchronization - GA	5.1(± 3.4)	0	25.3
Balanced Surface Min - Random	62.6(± 2.2)	55.2	59.5
Balanced Surface Min - GA	63.3(± 2.2)	55.2	70.8

Figure 7: Testing statistics for random genomes and the best genomes found by the evolution methods presented in Figure 6. Both the random genomes and the evolved genomes were tested over 1500 random CAs. Results from this basic GA were poor.

or evolving computing cellular automata has used up to this point. For examples, see [1, 5, 7, 2]. Also, unlike the 2-D CAs used here, all of these previous works use 1-D CAs.

Often, when attempting to construct interesting CA rule tables by hand, a type of *totalistic* rule table will be used. In this case the next state of a cell is determined by its own current state and the sum of the states of its neighbor cells. Instead of using the entire string $x_1x_2x_3x_4x_5x_6x_7x_8x_9$ as a lookup into the rule table, totalistic rules depend on $(x, \sum x_i)$ for $i \in neighbors$, where x is the state of the cell being updated and x_i is the state of neighbor i . This results in a rule table of size sn , where s is the number of possible states and n is the number of neighbors. In this case, the totalistic rule table has length 18.

One of the main changes when using totalistic over full rule tables is the difference in the size of the rule table, and hence a substantial reduction in the size of the genome used by the GA and the search space. The search space when using full rules is 2^{512} and for totalistic rules is only 2^{18} .

Not only are totalistic rules a small subset of full rules, but they are a special type. They seem to produce behaviors that “make sense” when trying to achieve a goal. For example, when trying to synchronize it would be logical to find out what state most of the neighboring cells are in on this step and change to the opposite of that state next time. This simple heuristic by itself will not provide synchronization, but rules similar to it, such as many provided by totalistic rules, will most likely perform well.

Surprisingly good results were achieved when using totalistic rule tables

instead of full rules tables as the genomes being manipulated by the GA. Figure 8 shows the number of fitness evaluations and time spent on evolving rule tables by the GA when using full and totalistic rules. The GA was limited to 30 minutes evolution time but stopped early if a rule table was found to be “perfect”. This was the case in the density classification and synchronization problems, where a perfect rule table was found in the initial population of totalistic rules. The more difficult balanced surface minimization problem allowed the GA using totalistic rules to run for the entire 30 minutes.

Density Classification

Method	Number of		
	Fitness Evals	Time (min)	Hero
GA - Full Rules	4008	30.0	59.1
GA - Totalistic	162	1.2	100

Synchronization

Method	Number of		
	Fitness Evals	Time (min)	Hero
GA - Full Rules	7860	30.0	23.77
GA - Totalistic	51	0.2	100

Balanced Surface Minimization

Method	Number of		
	Fitness Evals	Time (min)	Hero
GA - Full Rules	7920	30.0	70.2
GA - Totalistic	8016	30.0	99.0

Figure 8: Evolution times and heros for different GA methods for three CA problems.

Figure 9 shows the improved results for both random search and the GA when using totalistic rules instead of full rules. The random search results represent the fitness measures of 1500 random rule tables over 1500 random CA starting configurations. The GA results represent fitness measures of the best rule table, found by at most 30 minutes of evolution, over 1500 random

CA starting configurations. Totalistic rules provided a significant benefit to both random search and the GAs over all three problems. Balanced surface minimization, believed to be the hardest of the three problems, received the least benefit.

Density Classification

Method	Mean	Minimum	Maximum
Random Search - Full Rules	50.0(± 2.3)	41.6	57.3
Random Search - Totalistic Rules	51.6(± 8.2)	26.7	100
GA - Full Rules	50.0(± 2.4)	43.3	58.2
GA - Totalistic	99.5(± 0.8)	93.1	100

Synchronization

Method	Mean	Minimum	Maximum
Random Search - Full Rules	3.8(± 2.3)	0	14.7
Random Search - Totalistic Rules	5.8(± 11.8)	0	100
GA - Full Rules	5.1(± 3.4)	0	25.3
GA - Totalistic	90.9(± 7.7)	0.4	100

Balanced Surface Minimization

Method	Mean	Minimum	Maximum
Random Search - Full Rules	62.6(± 2.2)	55.2	59.5
Random Search - Totalistic Rules	60.4(± 5.8)	44.0	94.4
GA - Full Rules	63.3(± 2.2)	55.2	70.8
GA - Totalistic	68.8(± 9.2)	63.7	98.8

Figure 9: Testing statistics for random genomes and the best genomes found by the evolution methods presented in Figure 8. Both the random genomes and the evolved genomes were tested over 1500 random CAs.

It is apparent from these results that totalistic rules improve the maximum fitness found with random search but does little to improve the average fitness over many random rule tables. Figure 10 illustrates this graphically with histograms of fitness measures for random rule tables over 1500 random CAs using both full and totalistic rules for each of the three problems. The

general shape of the histogram is consistent within each problem between full and totalistic rules, but the totalistic rules have a small number of very high fitness outliers. This is possibly due to the reduced search space provided by totalistic rules and to the fact that totalistic rules are a subset of the rule space that provide consistent and symmetric behavior, therefore producing inherently better solutions to the problems presented here.

5 Fidelity vs. Complexity in Fitness

For the experiments presented to this point the fitness of a rule table, or genome, is the value of the problem specific fitness calculation at the end of one run of a random CA configuration. It is the hope that by testing a population of solutions with random CA configurations that a solution will emerge that performs well over all possible CA starting configurations.

This fitness evaluation method is of very low fidelity as it is taking into account only one time step of one random CA out of a very large space of possible CA configurations.

A possible improvement is to test each rule table on one random CA, but calculate its fitness periodically throughout the evolution of the CA. Because the calculations of fitness are cheap when compared to the execution of the CA, this periodic testing adds little extra complexity. This fitness evaluation is of higher fidelity in that it further ensures that the CA was consistently performing its given task. These additional calculations also allow for the assignment of extra fitness to those rule tables that solve their given task faster than others.

Further increases in fidelity can be gained by testing each rule table on several random CA configurations. This increase in fidelity does, however, come with a significant increase in time spent calculating fitness. The question is, is this increase in complexity worth the higher fidelity provided by it? In this case, it has been found that the increase in complexity is justified.

Figure 11 shows the number of fitness evaluations and the time spent for each GA. The maximum allowed evolution time was 30 minutes. Fitness type 1 evaluates the fitness once after 300 steps of one CA with random starting conditions. Fitness type 2 evaluates the fitness periodically, after 100, 200 and 300 steps of one CA with random starting conditions. Fitness type 3 evaluates the fitness of three CAs with random starting conditions, each after 300 steps. As expected, fitness types 1 and 2 computed about the

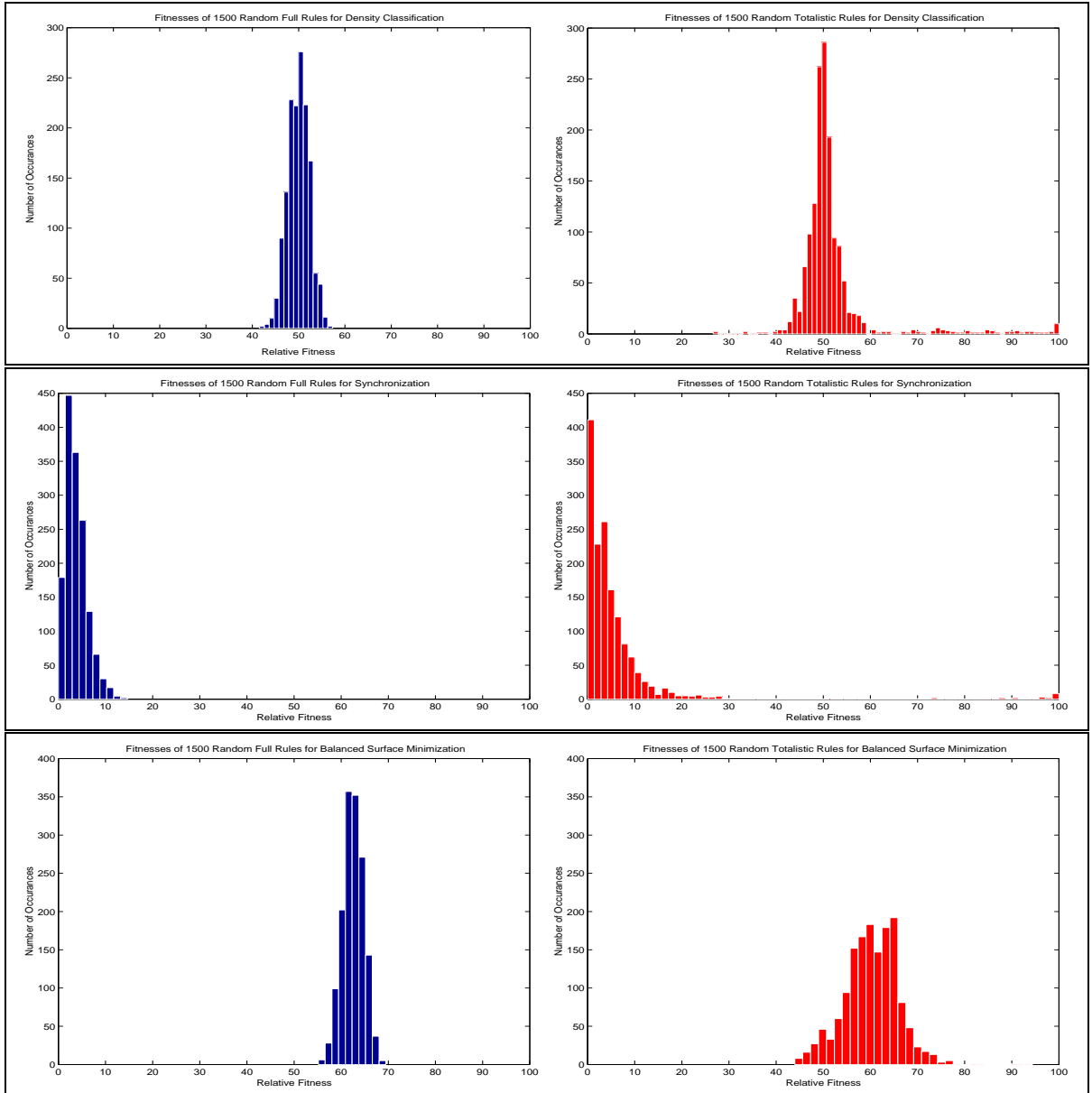


Figure 10: Fitness measure for 1500 random rule tables for each problem using both full and totalistic rules.

same number of fitness evaluations in the same period of time and fitness type 3 computed about one third as many as either types 1 or 2.

Figure 12 shows the results of testing the hero from GAs using each type of fitness evaluation. Each best found rule table was tested over 1500 CAs with random starting conditions. The results show that the higher fidelity fitness evaluation methods allowed the GA to find significantly better rule tables for the balanced surface minimization problem. Even though the third type of fitness evaluation computed one third as many fitnesses it improved the mean fitness of the best rule table from 68.8 to 82.9. Further work is needed to find the optimal trade-off between the complexity of the fitness function and its fidelity.

Balanced Surface Minimization

Method	Number of		
	Fitness Evals	Time (min)	Hero
GA - Totalistic - Fit Type 1	8016	30.0	99.0
GA - Totalistic - Fit Type 2	8024	30.0	98.4
GA - Totalistic - Fit Type 3	2724	30.0	94.4

Figure 11: Evolution times and heros for GAs solving the balanced surface minimization problem using three type of fitness evaluations.

Balanced Surface Minimization

GA - Totalistic - Fit Type 1	68.8(± 9.2)	63.7	98.8
GA - Totalistic - Fit Type 2	77.6(± 11.5)	64.4	98.8
GA - Totalistic - Fit Type 3	82.9(± 8.5)	65.2	98.6

Figure 12: Testing statistics for the best genomes found by the evolution methods presented in Figure 11. Genomes were tested over 1500 random CAs.

6 Application of coevolution

The strategy of coevolution in GAs is often identified with the first publisher of the concept, Daniel Hillis, who proposed its use in developing structures

using minimal nodes in a network which would sort a list of numbers [3]. The idea is that the progress of evolution toward an ideally fit population is spurred on by competition by provoking an arms race which must be won for survival. Hillis' results showed great promise for the process.

Just as Hillis' model used coevolving number lists along with evolving node connections, most applications of coevolution have involved a population of problem solvers and a population of initial states. The problem solvers are evolved in the usual way, but in addition, those initial states which foil the problem solvers are reproduced. The problem solvers are thus presented with progressively harder problems.

There have already been several attempts to apply coevolution to the problem of finding density classification and synchronization rules for one-dimensional CAs. Each of these employed coevolving initial states. Paredis sent the first alert that coevolution may not have great promise for CAs in his "Beware the Red Queen" article [6]. In investigating the causes for the lack of progress of his coevolutionary GAs, he found that a population of initial states for density classifiers would gradually approach a 50% density, which would render a classifier powerless to decide reliably. At this stage, if the average density of the population of initial states were to cross the 50% boundary even slightly due to random perturbations, the predominant strategies of most if not all of the CA rules to bring the classifier to one side would be rendered obsolete, and the primary goals of each population would be reversed. At this point, the same cycle can restart.

Juille and Pollack later published a paper claiming that when used with a technique called resource sharing, coevolution had a synergistic effect that prodded the GAs along to much better solutions [4]. Verfel, Mitchell, and Crutchfield, however, showed clearly that this effect was nearly entirely the result of resource sharing and not coevolution [7].

Since the same dynamics of one-dimensional CA coevolution could be expected to be seen in those of a two-dimensional CA as well, a different and more natural strategy was adopted here. Rather than coevolving initial states against the population of CA rules, another population of "evil" rules would be developed to fight against the "good" rules to bring one CA to its terminal state. The fitness of the evil rule was thus the complement of the fitness of the good rule for any particular result. Each genome, whether good or evil, would behave and reproduce in the same manner.

Early results showed that coevolution by fighting was no better than a regular GA, and in many cases it was even slower. Therefore more algorithms

were constructed and run on each of the three problems, with both full and totalistic rules. The strategies tested were as follows:

- **Random search:** To ensure that the GAs were truly accelerating the search, the progress of a random search was included for comparison.
- **Regular GA:** A standard, non-coevolving GA.
- **Coevolution:** Coevolution by fighting. Good and evil rules alternate in transforming a CA.
- **Cyclical coevolution:** CA rules that are forced to compete with enemies are assumed to be more robust than their low fitness may indicate. Furthermore, they may be subject to the limitations of coevolution as described by Paredis. Therefore, a combination of coevolution and regular evolution was tried, alternating between the two every 500 fitness evaluations, or a full cycle of 1000.
- **Fighting by tens:** Instead of strict alternation, this method allows a good rule to transform a CA ten times before the evil rule works on that CA ten times, and so on. This was tried in case the strict alternation of good and evil rules did not allow either rule to significantly affect the bias of the CA before its foil was allowed to operate on that CA.
- **Fighting by twenties:** Just like fighting by tens, only with a fight granularity of twenty.

6.1 A CA suitable for fighting

To acquire the results using a CA which can be set with a good and evil rule, a separate C++ class was implemented, and since at least double the number of calculations per generation would be required, efficiency was put at a premium. This class represented cells as individual bits and operated on four cells at once by expanding the rules given to make a rule which takes a 3 by 6 rectangle of cells and computes the four middle cells in one lookup step. All operations performed on the cells were bitwise operations, including the necessary grid wraparound operations. The speed that resulted was between five and ten times the previous C++ implementation, depending on architecture, and thus 250-600 times faster than MATLAB.

6.2 Early results

Simple coevolution by fighting made no improvements upon a regular GA, as in figures 13 and 14. Furthermore, even random search bested coevolution for the totalistic synchronization problems. This condition was possible due to a low mutation rate.

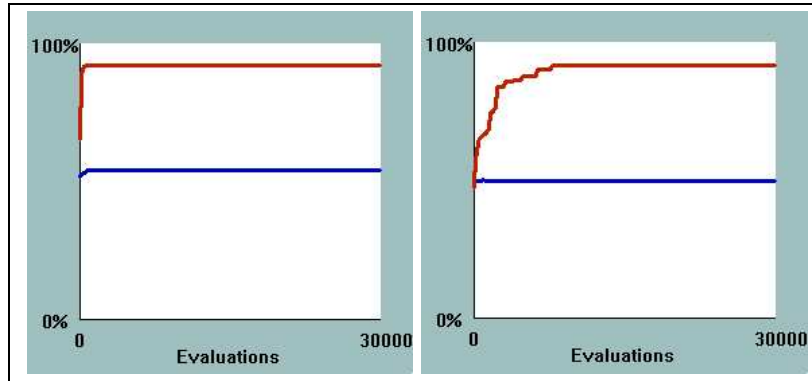


Figure 13: Regular GA (left) and coevolving by fighting (right), applied to the density classification problem with totalistic rules. The upper line represents the average current winner of all populations tested. The lower line represents the running average of all populations tested.

6.3 Cyclical phenomena

One strategy which performed as predicted was cyclical coevolution, although again, the performance overall was generally on a par with a regular GA. As is shown by figure 15, the moving average oscillates with the exact frequency as the cycle of coevolution and regular evolution. As predicted, average fitness shoots up quite reliably when coevolution is cancelled, but reverts downward as soon as it is applied again. However, there is no overall increase in fitness, even after 30 complete cycles. Full cycles of 200 and 2000 fitness evaluations were also tried but gave similar results.

6.4 Fight granularity

Increasing the granularity of the fighting approach, i.e., allowing each rule to apply itself more than once in a row to a CA, did produce some improvement

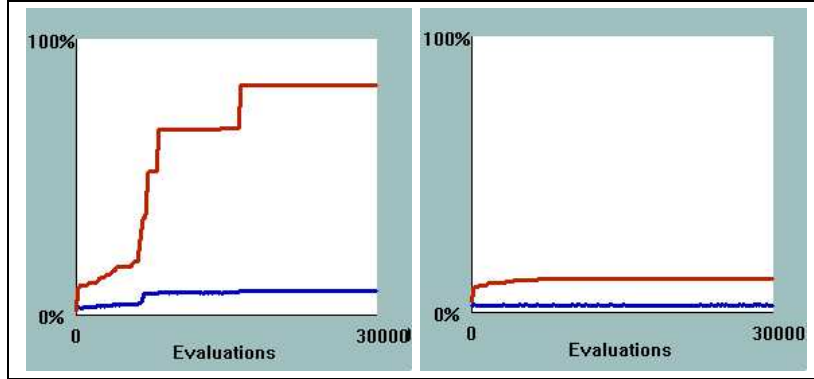


Figure 14: Regular GA (left) and coevolving by fighting (right), applied to the synchronization problem with full rules. Sudden leaps in fitness are an artifact of the process of averaging several runs.

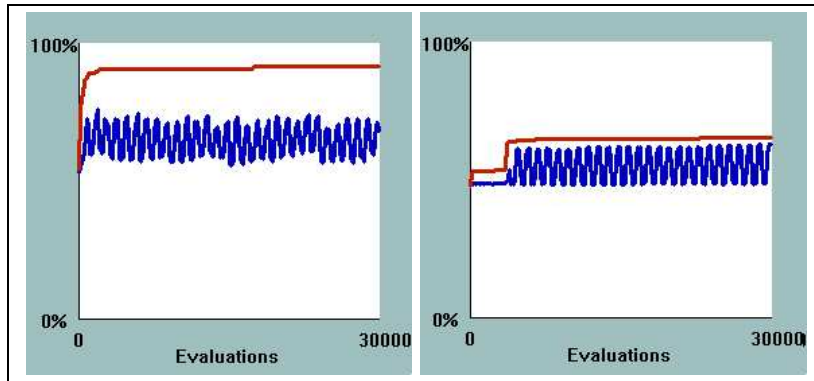


Figure 15: Cyclical coevolution with full cycle length 1000, applied to the balanced surface minimization problem with totalistic rules (left) and full rules (right). Averages move up when coevolution is not present and fall when coevolution is present. Little overall progress is made.

for a few problems (figures 16 and 17). This supports the hypothesis that a granularity of one does not allow a rule to properly apply its effect on a CA.

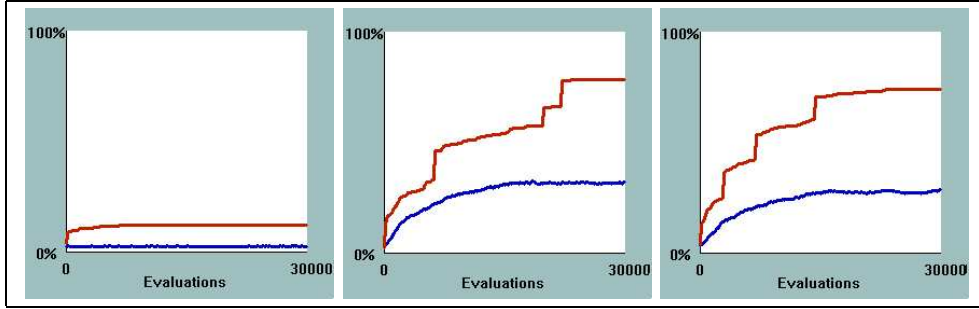


Figure 16: Coevolution of granularity one (left), ten (middle), and twenty (right), as applied to the full-rules synchronization problem.

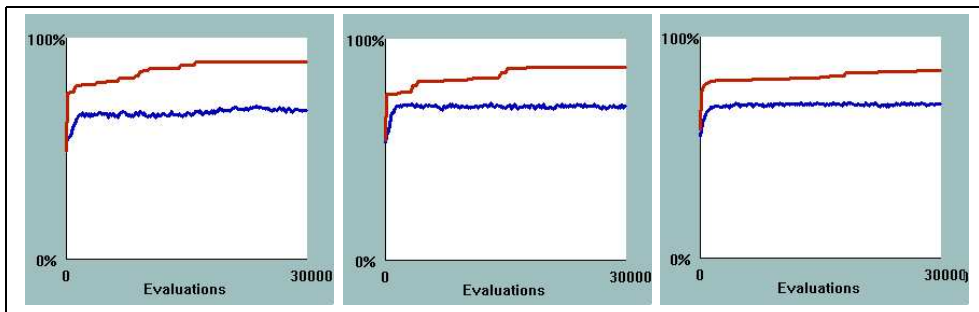


Figure 17: Coevolution of granularity one (left), ten (middle), and twenty (right), as applied to the totalistic balanced surface minimization problem. Higher granularities produce better fitnesses sooner.

7 The problem of synchronization

While most of the GA trials reported the final state of a CA created with the winning rule, they did not typically provide an animation of that CA from start to finish. Upon building a graphical viewer which scrolls through a file of collected best rules, it was evident that the vast majority of the rules which best solved the density and balanced surface minimization problems were also highly synchronized. Hand-calculated rules producing similar quality solutions did not synchronize, however, proving that synchronization was not a necessary quality for all solutions. Especially disturbing was the synchronized nature of the solutions to the density classification problems,

since the whole point of this problem is to produce a stable, completely on or off state.

Clearly the fitness function being used was too fragile, in that in most forms tried, it would only check one state of the board, i.e. just after the 300th step. And in all cases, it was checked after an *even* number of generations. Therefore the chances that a rule which solved the synchronization would also appear to solve the density classification problem was 25%. That is, if by 50% chance it was in the proper state after 300 steps when it was trying to turn all cells on, and by 50% chance on its evaluation against a mostly off CA it was in the completely off state after 300 steps, the rule would be evaluated as 100% fit and the GA would automatically stop. Inspection of the fitnesses evaluated in these GAs confirms that at a corresponding rate, rules were also found to have fitness values in the extremely low range, 0-5%. These represent the synchronizing rules which were unlucky enough to be in the wrong state for each half of their evaluations. The gap between the average, nearly random rules and the rules which achieved 100% fitness is thus highly pronounced, which explains why GAs with a very low average fitness nevertheless seemed to solve the problem.

It is interesting, as shown by this bias toward synchronizing solutions, that synchronizing solutions arise from the GA far sooner than properly classifying solutions do. Presumably this is due to the fact that there are simply *more* synchronizing solutions than classifying ones. However, the same bias toward synchronizing solutions can be found in the solutions of the balanced surface minimization as well. All of these solutions viewed brought a CA to a final condition of reversing its state every generation, in addition to satisfying the fitness condition. Yet hand-calculated solutions to the balanced surface minimization problem performed adequately without synchronizing. The mechanics of this mysterious bias must be a topic of further research.

7.1 A proper density classifier

Though this was a late addition to the research, one standard (not coevolutionary) GA using totalistic rules was run with a fitness function that evaluated not just after step 300, but after 301 as well. In fewer than 2000 evaluations, the GA generated a rule which solved the 70-30 density classification problem. It was confirmed using the graphical viewer that this solution was not a flashing, synchronized solution. Furthermore, its method

of classifying appeared quite powerful as cells in the minority state were quickly divided into convex blobs which would eventually disappear.

When tried on the 60-40 density classification problem, the rule continued to perform very well, though it was noticed that there was a slight bias toward identifying majority-on over majority-off. Thus with hand inspection of the rule, one bit was flipped to create the unbiased symmetric rule shown in figure 18

0:	0	0	0	0	1	0	1	1	1
1:	0	0	0	0	1	0	1	1	1

Figure 18: Totalistic rule for proper density classifier.

This improved version solved problems down to the hardest tried, 52-48, with impeccable accuracy. Figure 19 shows one typical progression.

Note that though the proper bias is always found, the CA does not always stabilize in a completely on or completely off state. This may be an example of where a strategy of first seeding a full rule population with totalistic rules, and then allowing the rules to evolve in the full rule space, may be able to find a full-rule improvement upon a totalistic rule.

8 Conclusion

This research touches on a number of issues in the application of GAs to two-dimensional CAs, including the most primary that will no doubt be a starting point for other research into this topic. These issues include

- the inefficacy of exploring the full rule space,
- the vastly greater power of totalistic rules,
- the power of higher-fidelity fitness evaluations over greater numbers of lower-fidelity evaluations
- the propensity of evolved rules to find synchronizing solutions, and
- the power of symmetric rules, at least for the density classification problem

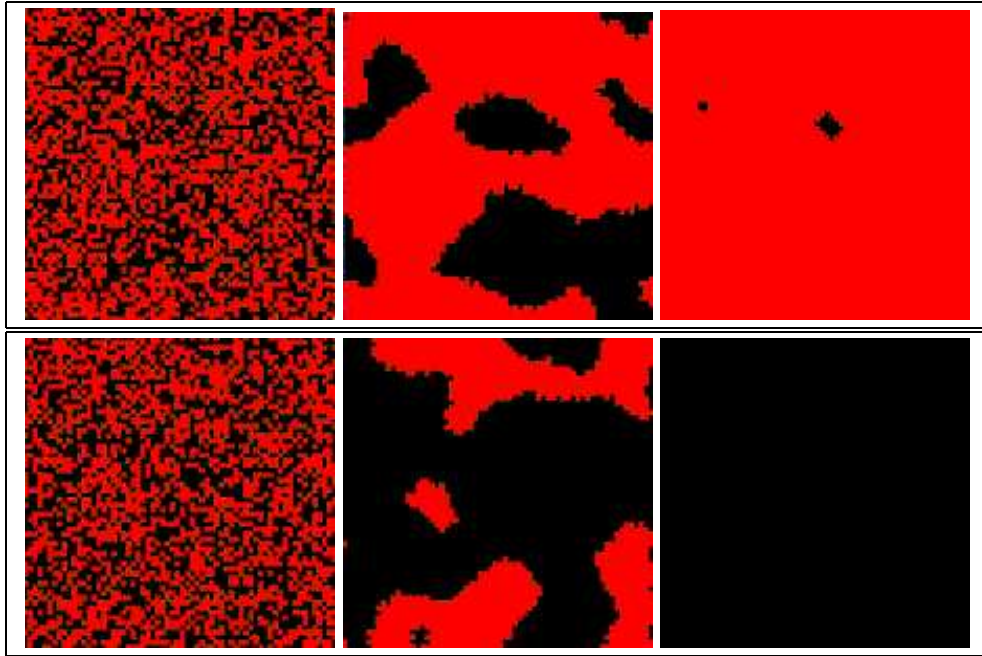


Figure 19: Behavior of a proper density classifier given an initial state of 52% red (top row) and 48% red (bottom row). The initial minority state is quickly separated into convex blobs which shrink until they disappear or become stable.

Furthermore, various forms of coevolution by fighting have been shown to be ineffective in this context, though the behaviors of the different forms have been demonstrated. In particular, the greater effectiveness of fight granularities higher than one are pronounced. Last, an excellent density classifier was exhibited which may be used as a benchmark for judging other classifiers as well as a specimen for the study of how it works.

Questions that remain involve the rule space of 2-D CAs. There must be a reason that good solutions are more often found in the totalistic rule space, and that synchronizing solutions appear to be more common than non-synchronizing ones. An understanding of these issues is likely to allow greater progress in the development of efficient GAs for similar problems.

References

- [1] Rajarshi Das, James P. Crutchfield, Melanie Mitchell, and James E. Hanson. Evolving globally synchronized cellular automata. In Larry Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 336–343, San Francisco, CA, 1995. Morgan Kaufmann.
- [2] A. David, B. Forrest, and H. Koza. Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem, 1996.
- [3] W. D. Hills. Coevolving parasites improve simulated evolution as an optimization procedure. In C. G. Langton, editor, *Artificial Life II*, pages 313–324. Addison-Wesley, Reading, MA, 1992.
- [4] H. Juille and J. B. Pollack. Coevolutionary learning: A case study. 1998.
- [5] Melanie Mitchell, James P. Crutchfield, and Rajarshi Das. Evolving cellular automata to perform computations. In Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors, *Handbook of Evolutionary Computation*, pages G1.6:1–9. Institute of Physics Publishing and Oxford University Press, Bristol, New York, 1997.
- [6] Jan Paredis. Coevolving cellular automata: Be aware of the red queen. In Thomas Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, San Francisco, CA, 1997. Morgan Kaufmann.
- [7] J. Werfel, M. Mitchell, and J. P. Crutchfield. Resource sharing and co-evolution in evolving cellular automata. *IEEE-EC*, 4(4):388, November 2000.