Genetic Algorithms Applied to 1-D Cellular Automata

Matthew Aasted Jeffrey DeCew Anthony Roldan

Franklin W. Olin College of Engineering
Discrete Math
December 17, 2005

***Abstract*—In this project, we investigate the use of genetic algorithms to evolve specific rulesets governing one-dimensional cellular automata. We outline the basic concepts of cellular automata and genetic algorithms, explain in depth why we selected the algorithms which we chose, and outline the basics of our implementation. We discuss our success in developing the final state of automata by evolving rulesets to solve specific problems given randomized initial conditions. Finally, we reflect on the ability of a ruleset to determine the steady state in spite of random initial environments, and discuss future opportunities for research.**

I. Introduction

The goal of this project was to evolve rulesets to govern Cellular Automata. Cellular automata are exciting because they hold the possibility of emergent complex behavior on a global scale being born from simple interactions on the local level. Genetic algorithms are a current hot area of investigation in the computer science community due to their potential for solving problems that are difficult for a human to analyze.

In this paper, we outline the basic ideas behind cellular automata and genetic algorithms, move on explain how we apply them in this project specifically, then offer detailed analysis of the results we obtained from experimenting with CA ruleset evolution and possible future directions for research in this field.

II. Cellular Automata: The Basics

A. Definition

Cellular automata (CA) are environments consisting of discrete cells placed around a uniform grid. The states of the cells in a given timestep are governed by some set of rules applied to the previous time step.

Cells in the grid are the most basic component of CA. Essentially, a cell is simply a location which has a state. The number of states is variable, as is the geometry of the cell. That is to say, the grid can be n-dimensional and any number of cells can be adjacent. For example, a one-dimensional line of cells, a two-dimensional honeycomb-esque hexagonal lattice, or a three-dimensional grouping of cubes would all be valid grids for a CA.

Time is discrete in CA. The states of the cells in timestep t are determined by the states of the cells in $t-1$. The next state of a particular cell is determined by its current state and the state of its neighbors – not necessarily only adjacent cells, but all cells defined to be in its neighborhood. The neighborhood consists of only adjacent cells in many CAs, but could be restricted to only the cells that can be connected with straight instead of diagonal lines in a 2D CA, for example.

B. Uses and Intrigue

The most famous CA is Conway's Game of Life, which first appeared in a 1970 issue of Scientific American and whose ruleset makes for unpredictable and interesting patterns.

The Game of Life takes place on an infinite two-dimensional grid, where cells can have only two states – "alive" and "dead" (or "on" and "off").

The Game of Life's rules are totalistic – that is to say, they all deal with the total number of cells that are adjacent to the current cell that are on and not with specific arrangements of cells. The rules are simple: Any live cell with fewer than two neighbors dies, as if of loneliness. Any live cell with more than three neighbors dies as if of overcrowding, and any dead cell with three neighbors comes to life. Live cells with two or three neighbors do not change their state and continue to live.

The Game of Life, like any CA "game," is not really a game, so to speak – all future states of the Game are defined by the initial conditions, as is true of CAs in general. Special initial arrangements of cells have been discovered that can continuously translate across the grid ("gliders"), or ones which continuously create gliders ("guns"). The Game of Life makes a convenient starting point for explorations into CA, because it is both simple to understand but holds nearly infinite possibilities.

1) Applications: CAs do not currently have any widely-used applications, but provide a valuable modelling tool for simple biological systems; certain seashells, for example, have CA-like patterns on their shells. They have been proposed for usage in cryptography, since it is easy to generate future states from a CA if you know the ruleset, but very hard to run a CA in reverse to determine an initial condition. This could be used in a public-key crypto system although no such implementations are currently on the market.

C. Considerations in Implementations

The iterative, discrete nature of CAs lends itself to straightforward implementation on a computer system. One large

consideration is how to deal with boundary conditions in CAs. An infinite grid of cells, as occurs in idealized CAs, all of whom change state on every timestep is not possible to simulate on a computer with finite resources. In some situations, such as the Game of Life, where cells bordered only by cells of a certain state remain in that state (cells far away from any living cells will continue to remain dead), an infinite grid can be implemented, but in many situations, it cannot be. In these cases, it is necessary to define spatial boundary conditions. Circular (or toroidal, for 2D CAs) grids are often used to avoid this problem, where the end of a row or column of cells simply wraps around to the opposite side. Another possibility is fixed boundary conditions, where one simply defines boundary cells to always behave as if they were a certain state.

Another consideration is the complexity of the ruleset. Obviously, a ruleset has to be small enough that the next state of a cell can be computed without extensive searching. For a non-totalistic CA in two dimensions looking out three levels, there are 36 cells whose states have to be described to determine the next state of the current cell. For just two states, that means 2^{36} or over 68 billion definitions one would have to make in order to account for every possible combination. For this reason, smaller neighborhoods or totalistic CAs are generally favored over more complicated ones when it comes to computer implementation.

D. Ruleset Encodings

Rulesets are fundamental to CA implementations. As such, having a compact form to describe them is wildly convenient. To this end, notice that only S^n characters are required to describe all possible situations for a CA rule, where S is the number of states and n is the number of cells in the neighborhood, including the cell itself. Many of the most common CAs are two-state, since their binary nature is especially well-suited to computer implementations. In these binary implementations, it is conventional to represent the ruleset as a bit string, which is then converted to decimal for compactness.

For example, for a one-dimensional CA with a neighborhood of only one cell in either direction, a bit string of length three describes a possibility for the next state. The first bit represents the state (1 or 0) of the cell to the left, the second bit represents the state of the current cell, and the third bit represents the state of the cell to the right. That means that 2^3 or 8 bits (conveniently, one byte) are required to describe every possible state. In the final string, the value of bit i (where $0 \leq i < 8$) represents the resulting state of the current cell based on the condition represented by i as translated in binary.

Consider the ruleset represented by the bit string 10101011. That translates into the following final states:

bit (bin)	bit (dec)	resulting cell state
000	0	1
001	1	1
010	2	0
011	3	1
100	4	0
101	5	1
110	6	0
111	7	1

This method of representing rulesets as strings is conventional for cases with more than two states or a neighborhood of size more than three as well. In those cases, the string of length S^n is converted from a base S number to a decimal number for convenience. Either way, this makes for a reasonably compact notation which can fully describe any set of CA rules.

III. Genetic Algorithms: The Basics

Genetic algorithms are a method of finding extrema within a single or multivariable space. Under usual conditions, this means finding either maxima or minima. Typically, they are used to explore spaces which are either very large or spaces in which it would be difficult to find extrema by analytical methods. Genetic algorithms have less trouble with local extrema and are fairly resilient; however, genetic algorithms alone find the true local extrema relatively infrequently without special adaptation of the GA. This is because they easily find a solution which is near the extrema, from which it is possible to use other methods, such as hill climbing algorithms, to find the true extrema.

To find extrema, the genetic algorithm needs only to be able to evaluate for fitness and somehow randomly alter solutions within the solution space. It is convenient to use genetic algorithms because, despite their rather high computational overhead, it is typically simple to develop a fitness criteria to a problem. Genetic Algorithms can sidestep the need to do analysis of the space, and typically can find a good solution to a problem within a relatively short amount of time. [2]

A. Definition

A generation is the set of solutions which are considered at a given time step. The first generation is the input to the Genetic Algorithm; typical methods of finding a first generation include random generation, seeding with some hand-picked value and mutating it to get the rest, or simply manually entering a solution set. Meanwhile, fitness is defined as the quality of a given solution in solving the originally posed problem. It is the strongest requirement on a genetic algorithm; fitness must be evaluable at each generation for the genetic algorithm to be able to weed out the weak and identify the strong to be selected for breeding. This typically presupposes an interpretation step between the evolvable encoding and the fitness evaluation, as this solves many of the problems associated with encoding evolution.

Breeding typically consists of taking the strong members of a generation and either performing mutation or crossover

on them. Mutation, in the case of a bit string, means a probabilistic flipping of random bits to arrive at a new solution. Crossover is typically taking two strong members A and B, choosing a random point along them, and creating a new specimen by concatenating the portion of A before the point with the portion of B after the point. If the solution space has been encoded in such a way that it has multiple chromosomes, this crossover may happen on a single or multiple chromosomes, or may even consist of swapping chromosomes. Notice that, if one were to encode their solutions in such a way that they wrote working chromosomes which are not functionally divisible, crossover could be used by swapping sets of chromosomes without breaking the chromosomes up internally.

The encoding of the elements of the search space is frequently non-trivial. The reason for this is that the encoding needs to intelligently deal with random mutation, crossover, or both. This is relatively difficult, especially if the evaluation requires that the specimen meet strict criteria. Fortunately, most of these problems can be solved by writing a clever interpretation of the encoding such that garbage in a solution is ignored. Alternately, a well-encoded search space can make the development of uninterpretable data within the encoding impossible because all possible combinations result in an evaluable solution.

B. Uses and Intrigue

Genetic algorithms are very popular because frequently the encoding and evaluation of fitness can be done in an intuitive way on spaces where analytical solutions are non-intuitive or impossible. When you can state your problem as a maximization (even a binary maximization), you're typically almost immediately able to formulate it into a fitness criteria. This makes them very attractive to researchers and designers because it means that the human computation time is much lower for the same results; since digital computation is very cheap, this means reduced costs for research. Furthermore, the appeal to biology inherent in both name and function give genetic algorithms remarkable buzzword properties and a similar ability to create novel solutions which a human being would be unlikely to produce.

IV. The Cellular Automata Used in This Project

In order to begin our project, we had to decide on a particular form of cellular automata so that we could do an in-depth and relatively concrete investigation into the automata and, more importantly, into genetic algorithms. In order to make a good decision about our cellular automata, we had to come up with some criteria against which we could compare our possibilities.

A. What We Needed From our Automata

1) The Grid: Because the main investigation of our project was meant to be in the area of genetic algorithms on cellular automata and not a pure study of obscure cellular automata, we decided that any excessively complicating factors like using triangular, hexagonal, or other non-square grids would be an immense misappropriation of resources in the computer implementation.

To enforce an environment of finite size, we decided that we would use either a circular or toroidal environment for boundary conditions. We felt that static boundary cells could seriously skew results (by providing two cells who always have the same neighbors) while an infinite, unbounded environment would restrict the types of CA rules we could work with, virtually forcing us into only totalistic rules.

2) Complexity: The first consideration we had was that in order to run any trial-and-error type search on cellular automata, we would have to be running these simulations repeatedly. The speed of computers renders this a non-issue for isolated tests, but could become an issue if we selected a particularly computationally intensive CA. In order to avoid problems with extremely slow simulations that could take minutes or hours for us to run, we wanted our cellular automata to be relatively quick to calculate.

3) Versatility: The next consideration for our CA was that we wanted it to be capable of taking as many distinct forms as possible. In other words, the distinction between totalistic and non-totalistic cellular automata was important to us. We decided that using a totalistic automata was too restricting and would reduce the number of actual solutions we could find. We wanted our cellular automata to take – or at least have the ability to take – wild and asymmetric forms.

4) Encoding: While the encoding of our CA falls more into the realm of the implementation of our GA, we decided that we had to make certain restrictions anyway. First of all, we wanted to ensure that the encoding we used for our CA was computationally feasible. In other words, we had to have a relatively compact form for describing the ruleset of our cellular automata. Thus any cellular automata ruleset which couldn't be described with a number of reasonable size would be too complex for an investigative project into genetic algorithms.

5) Visualization Considerations: A major consideration in choosing our CA implementation is how easy it is to show our results. Time-based visualizations, while impressive in their own right, are difficult for us to think about on a large scale. We wanted our results to be visualizable in such a way that we wouldn't have to use video in order to convey the information about any given trial.

6) Expandability: Because we knew that we were fallible and that we might make a poor choice when selecting our automata, we were looking for automata that could be expandable so that if our original form was too simple we could easily expand our implementation to a more complicated form without too much lost time. In other words, we decided it would be better to err on the side of caution and pick a less complicated CA over a more complicated one.

B. Possible Ways to Meet our Needs

1) Non-Totalistic 2-D Binary Automata: The first CA type that we looked at was 2-D, because they have an aesthetic appeal that drew our attention. When talking about 2-D automata, one possible example which we were discussing

was the idea that a 2-D CA might be evolved that could take a vertical line and cause it to break apart and re-form itself after translating. We figured that this would be an incredibly good demonstration of a functioning genetic algorithm. It is immediately clear, however, that any CA like this is asymmetric. But because of the very nature of totalistic CAs, they are symmetrical. Thus to maximize the options for evolvable traits, totalistic CAs were out.

2) Totalistic 2-D Binary Automata: We started doing enumeration of the possible ways to define a ruleset in a 2-D binary automata using just the 8 local neighbors and itself. The number of possible states for 9 binary cells is 2^9 or 512. That means that a binary string of length 512 would be required to describe a single ruleset. We decided that this was too complicated and too large of an encoding for the scope of our project.

3) Non-Totalistic 1-D Binary Automata: Given our disapproval of the large encoding of 2-D Automata and the symmetry of totalistic automata, this option becomes the logical next step. We found that to describe a ruleset for this option we would only need a binary string of length 2^3 or 8. This seemed to be a good option given that this allows for 2^8 or 256 possible rulesets, which is a relatively large search space for a GA. Also, because it is 1-D, it would be very fast computationally, and because it is non-totalistic, it would give use a highly versatile form. This form had the immense benefit that it would be relatively easy once we completed our initial revision of code, to re-formulate as a ternary or quaternary automata. In these cases the rulesets are ternary strings of length 3^3 or 27 for the ternary form and quaternary strings which are 4^3 or 64 digits long for the quaternary form. While we knew it was unlikely that we would go to quaternary forms, it was comforting to know it was there. After close scrutiny, this looked like an acceptable choice.

4) Totalistic 1-D Non-Binary Automata: Even though non-totalistic 1-D binary automata seemed to be a great avenue for exploration, we wanted to look at a few more options before going on. The next feasible choice was non-binary, but totalistic automata. We immediately recognized that these had the great advantage of being easily scaled to any complexity. Unfortunately, a quaternary totalistic automata with only the local three cells would have 12 possible states, and 4 possible results for each state. This means there are only 48 possible rulesets. This is incredibly low, and because the scaling is only Sn, it would require a huge number of states to make a sufficiently complicated automata. Using 16 cell states would result in 765 possible rulesets. After a brief discussion we decided that even though this form could make for interesting GA analysis, it began to lose the discrete quality of the cell states and would sacrifice the elegant simplicity of CA.

C. The Choice and Visualization

Because it had so many advantages, the 1-D binary non-totalistic cellular automata was the final choice. It had everything we wanted, even scalability, and was not going to cause us immense trouble in implementation. An added benefit of this CA choice was that we had a great possibility for visualization. In a few papers we found on the subject, 1-D CAs were shown as images, where a row in the image was a snapshot of the CA in time. The rows were organized so that as you scan down the image you can see the way the automata changes. This means that state number increases down the rows, and each row represents a complete state of the system. Therefore, it is very easy to understand at a glance what the automata did. We felt that this has immense benefits for our qualitative analysis of any given automata and enhances our ability to explain automata to other people.

V. The Genetic Algorithm used in this Project

A. What we Needed From our GA

For our binary cellular automata, we were exploring a sample space with 2^8 discrete solutions. This means the space was relatively small. Therefore, we had loose requirements for our Genetic Algorithm; essentially, it needed to converge relatively quickly on a solution and not be very computationally intensive to implement. We wanted our encoding to be such that it was at least an onto map to the search space. We spent a fair quantity of time discussing the possible algorithms and encodings we could use. Some very heated arguments occurred over the strengths and weaknesses of various encodings. We decided, given our inexperience in the field and inability to run time-consuming computer simulations, that for our problem a one-to-one correspondence would be easiest to implement without sacrificing too much.

B. Possible Ways to Meet our Needs

Much of the focus of us picking a GA was developing an encoding of a ruleset that allowed for easy or novel mutations without requiring excessive computational complexity or storage space. We evaluated possible evolution algorithms based on the encoding under consideration and its potential for either mutation- or crossover-based evolution.

1) DNA-Inspired Encodings: Our first hope was to implement an encoding that was DNA-inspired – one that used a minimum of symbols (like DNA's A, C, G, and T) in a variety of multi-symbol patterns to represent information. This could provide for evolution methods similar to those observed in the biological world.

Two solutions with high fitness values could be reproduced sexually, with crossing-over of genes in such a way that the two solutions could pass both coding and non-coding sections. There would be special "start" and "stop" symbol groupings (or "codons", as they are known in the biological world) that would allow us to code for specific rules, then other codons that would allow for exceptions or inert regions.

The allure of such a pattern would be to emulate the mutations that can take place in biological systems–the removal of a stop codon resulting in a very complicated rule, the removal of a single symbol skewing the following codons to create drastic changes, and that sort of thing. This sounded very exciting to us, and would require great concentration and effort on our part in formulating an interpreter adequate to the problem.

Ultimately, however, emulating the natural world was impractical. While compelling as a method to describe the rules applied in the complex 2-D CA we were initially considering, when working in a solution space of only 256 possible solutions, we decided it would be more trouble than it would be worth. Furthermore, it would be computationally impractical, as we anticipate that we would need a much larger number of fitness evaluations and mutations to develop an adequate solution, even in our sample space. Nonetheless, a DNA-inspired encoding could be a valuable and interesting area of investigation for complicated CA rulesets or for a variety of other problems.

2) Operator-Based Encodings: Similar to DNA-inspired encodings, an operator-based encoding has certain symbols that code for rules and exceptions – and would provide a convenient way to express all possible conditions via a symbol that codes for an "else" condition. For example, we could have a string like 'i001t0i011t1e0', which cold expand into an if-then-else structure that would read "If the neighborhood is 001, then the final bit is zero. If the neighborhood is 011, then the resulting bit is one. Otherwise, the resulting bit is zero."

This provides for the flexibility of a DNA-inspired encoding and similar scalability when straightforward string-based encoding becomes too large (as in the case of the 2D neighborhood of size 36), the difference being that single symbols, instead of codons, are used to represent a condition. This promises a more compact encoding at the expense of biological realism and an increased difficulty in implementing a sexual reproduction algorithm. Again, however, operator-based encodings suffered from the same issue of being too complicated for applying towards simpler CA problems, and we ultimately decided against them given the simpler CA we planned to work with.

C. Our GA Method

1) Encoding: The encoding we finally settled on was the conventional encoding mentioned in the introduction to cellular automata earlier in this paper. That gave us an encoding of only 2^3 or 8 bit length which fully expressed all possible options.

2) Mutations and Evolution: Our final GA was mutation-based. The first generation is based on a "seed" ruleset which is mutated eight times iteratively to produce a population of size 9, meaning that we mutate the seed, save the result, then mutate *that result* again, save the result, etc. This gives us an initial population with enough diversity to be meaningful but all still related to the initial seed such that specifying an initial seed which is close or far away from an expected solution can impact time required to generate that solution.

Between generations, as you can see in Figure 1, we select the three best ("most fit") rulesets, keep them in the next generation, and also mutate each of them twice to produce a new population still of size nine. The six less fit rulesets are discarded.

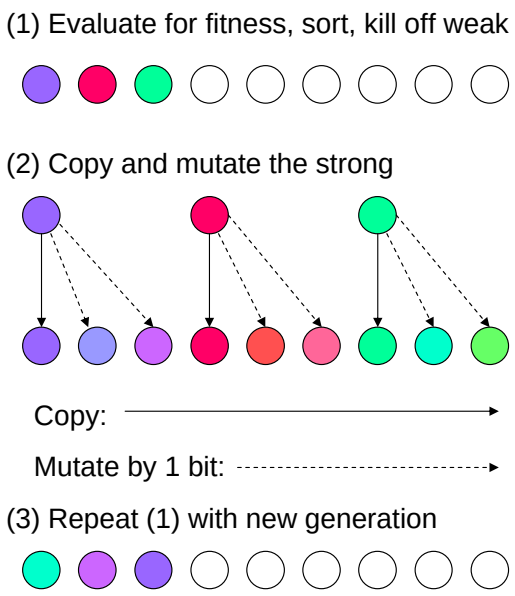

Fig. 1. Diagram of our algorithm's evolution process. Essentially, between generations the top three specimens from the previous generation are each copied and mutated twice to produce the new generation.

VI. Method of Implementation

We implemented our algorithms in PHP running on Jeff's server. Although PHP was not the ideal choice for computational speed, it afforded us a high ease of programming and more importantly, an easy web interface such that any of us or others could access the server and experiment with CA and GAs given only a web browser. This significantly enhanced our ability to demonstrate the program to others, and since demonstrating our work to classmates was a major goal, we thought the trade of speed for versatility was worthwhile.

PHP also provides object-oriented support, which we were able to effectively leverage to expedite development times. Our main class for running evolution, for example, is easily extensible with new definitions of fitness to allow for experimentation by changing our evolution goal with a minimal rewriting of code. Likewise, our ruleset class and environment class, the two of which combined define the behavior of a CA from one timestep to the next, are also fully extensible such that we could rewrite them to support more complex CAs supporting two-dimensions or ternary instead of binary states without requiring a rewrite of other code.

VII. Defining the Problems to Solve

A. Proof of Concept: Blackout

The first problem we attempted to solve using our GA was the simple one of finding a ruleset that would black out the final state of all cells given any initial conditions. To a human, the ruleset that will produce this result is obvious - '11111111' - which translates to no matter what the state of the neighborhood or current cell is, make it so the next state is black. We thought this was an appropriate problem for testing because it is very obvious by human inspection whether or not the test criteria is working or not.

B. Proof of Concept: Whiteout

The flip-side to blackout is whiteout, in which the final state of all cells is zero given any input condition. Again, the ideal solution to this is obvious - '00000000' - so it is valuable as another simple test.

C. Grayout

The first more interesting problem we decided to solve was the idea of creating as grey of an automata as possible. This problem was interesting because it was very easy to rank an automata state at a given time as having even distribution of black and white, while at the same time it is relatively difficult to come up with an ideal solution through inspection of the problem. However, despite the ruleset which results in this not being obvious, it is obvious to humans (and to our algorithm) whether or not it works in the final stage, making this a convenient test in which we anticipated non-trivial results.

D. Static

Another interesting case was the case of creating a static ruleset – one in which the final result is the same as the initial condition.

The ideal solution to this ruleset is one in which only the cell's current state is looked at and its neighbors do not influence future states. This way, the state of any given cell will remain static throughout time.

E. Tetris Pattern

The problem we attempted to solve here was not intentionally to find a geometric pattern as the name implies, but rather a particular binary string we often saw when running random rulesets with random initial conditions.

We seek to maximize the pattern '110100' in the final row of our CA tests, which coincidentally generally comes up when the geometric pattern formed by a 2D representation appears like L-shaped Tetris blocks.

F. Diagonal Pattern

This problem was similar to the Tetris pattern problem, except we are now instead looking for the patterns '0110' or '1001.' Again, this pattern does not necessarily imply diagonal lines in a 2D representation, but that happened to crop up during random tests.

G. Binary Result Attempt: Density

The most complicated behavior we attempted to create was to solve a density problem. We can define density of an automata state (ρ) as the number of cells at a given timestep whose value is one. We hoped to evolve CA rulesets that would allow us to ask the question "Is the density of the initial condition greater than x?"

If the answer was yes, and $\rho > x$, the final state of the CA would be all black (truth value 1), whereas otherwise the final state would be all white (truth value 0).

VIII. Developing Fitness

As stated before, genetic algorithms do not tell the system where to end up, but rather use an immediate feedback system to decide where to go. The typical fitness feedback system measures how well an evolved ruleset fits certain criteria on a 1-dimensional scale. The algorithm then uses this information to proceed. In order to simplify our algorithm, we decided to uniformly define our fitness to be a scale from 0 to 1.

A. Requirements of a Fitness

We required a fitness to be able to be expressed quantitatively and to be algorithmically determinable. Though it would be possible for us to act as the fitness measurers by hand, we decided this was impractical; furthermore, quantitative fitness analysis allows algorithmic determinations of strength. Since our goal was not to make "Genetic Algorithms: the Board Game", we decided that the more we could automate, the better.

B. Our Fitness Algorithms

The fitness algorithm depends on the problem we wish to solve with the CA ruleset. Each problem we tacked has a specific method we devised to determine how fit the final result is.

To describe these fitness-measuring algorithms, we will define $F(a)$ as the fitness of a final state, C as the total number of cells, and C_w as the number of white cells and C_b as the number of back cells. We will also define a cell at position x and time t as $c_{x,t}$

1) Fitness: Blackout/Whiteout: For the blackout or whiteout condition, the determination of fitness is simply $F(a) = \frac{c_b}{C}$ or $\frac{c_w}{C}$, respectively. This way, the most fit blackout or whiteout condition where $F(a) = 1$ is one in which all cells are entirely one color.

2) Fitness: Grayout: The grayout fitness test is to look for alternating white and black cells. A counter is incremented every time a cell after the current cell is a different color, (by checking if then the final value of this counter is divided by C.

3) Fitness: Static: The static condition simply checks the every cell c_{x,t_f} and compares it to the cell $c_{x,0}$. If they match, a counter is incremented and then the result is divided by C to normalize it to a value between zero and one.

4) Fitness: Patterns: The fitness test for patterns, either Tetris blocks or diagonals, is to test for the desired pattern beginning at every cell, and to increment a counter if it is found. In the end, divide the counter value by $\frac{C}{p}$, where p is the length of the pattern.

5) Fitness: Binary Result Density: The fitness test for density in the $\rho = 0.5$ case that we checked is to first determine the density of the initial condition, and determine if the end result should be a binary 1 or 0 (more or less than 0.5 density, respectively). It then performs the same test as used in the blackout or whiteout fitness tests to see how much of the final result is black or white, and then squares the result to increase the reward for better solutions.

IX. Results

A. Blackout (and Whiteout)

The blackout test was wildly successful. It has generated acceptable results in 4 generations or less almost every time. Usually the initial population seeding produces either a totally successful result or one so close that only one or two generations are needed to make it successful. However, the interesting thing about our results is that they very rarely produced the ideal solution. (Figure 2) The reason for this shortcoming is that when it comes to blackout, there are multiple imperfect solutions that work for almost any initial condition. The problem is that we have a limited set of initial conditions and if those do not include the special cases (like all white) then some rulesets (like rule 127) will not have the desired result.

Fig. 2. A successful but non-ideal blackout. Rule # 159

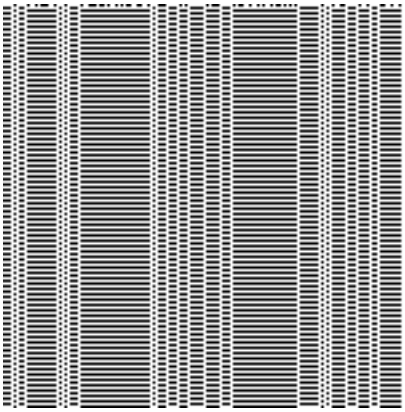

Fig. 3. The local maximum rule in our algorithm for blackout. Rule #128

Another anomaly that was discovered on one test of the algorithm was when we started with a group of rulesets among which the best result was rule 128. This is an imperfect whiteout rule with drastic failure because all white neighborhoods result in a black cell. This means that the final result for most configurations is about 50/50. Thus this has a fitness of roughly 50% for blackout. Unfortunately, for most initial conditions, most 1-bit mutations perform significantly worse than 50% for blackout. Thus, when the GA ran, and 128 (Figure 3) was the best rule in the first generation, it became stuck at that local maximum, because mutations were not great enough to find a better solution. This is not the only case where we found false maxima, but it is the best example of a case that is so far from the ideal solution and with such a clear limit to its effectiveness.

Whiteout went almost identically to blackout, in a probabilistically likely way, except that it was, of course, in reverse.

B. Grayout

Grayout was our second major success. Though a perfect solution doesn't likely exist, we found good solutions within 10 generations on most trials. One particular solution (Figure 4) works by cutting long sections of solid color at the ends and replacing the ends with alternating color. Another good solution (Figure 5) cuts off at only one side of long sections and breaks it down so that eventually the image is composed of all white columns, all black columns, and alternating black-white columns. These result in some remarkable images.

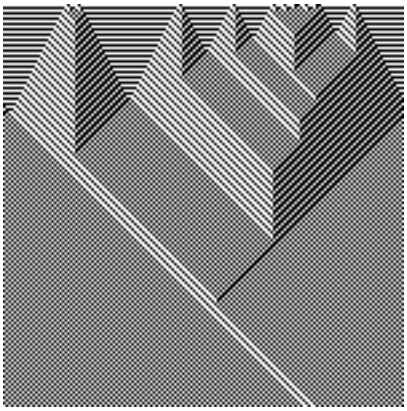

Fig. 4. This grayout rule effectively brings even highly uneven images very near gray. Rule #156

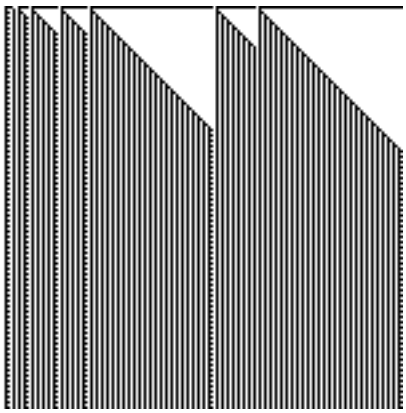

Fig. 5. This grayout rule brings highly uneven images toward gray, and settles into a pattern. Rule #56

C. Static

Static was also successful. Since there is an ideal solution, we frequently evolved to that. (Figure 7) However, we found some solutions we didn't anticipate. The first one of these

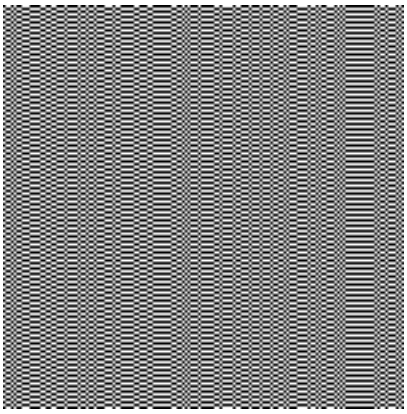

Fig. 6. The alternating solution to the static problem. Rule #204

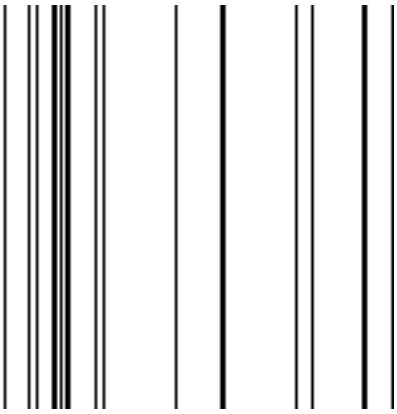

Fig. 7. The ideal solution to the static problem. Rule #51

took advantage of our even number of time steps, and simply alternated the cells every time step. That is, all white cells turned black and all black, white. Another unexpected group of results happened because of our design choices. Since we decided on a circular environment and our width of our environment was equal to its length (time steps = number of cells), a rule which simply shifted all cells to the left every time or another which did the same to the right would show up as a solution. A variant on this was alternating black and white as it did this. (Figure 6) To suppress these results, we simply changed the dimensions, and then we got the one which doesn't change each time.

D. Tetris Pattern

The Tetris pattern was fairly successful, though we don't believe a perfect solution exists. Solutions results in a pattern which has a diagonal flow of L-shaped blocks (Figure 8). We found the time-based patterns which result from end-game fitness requirements remarkable, as it allowed us to produce fascinating designs in the environment whilst controlling only the ruleset.

E. Diagonal Pattern

Like the Tetris pattern, we found that the results of this test resulted in time-based patterns – again, they were diagonals for this one (Figure 9). Our solutions were fairly good, but the all resulted in the pattern of diagonals.

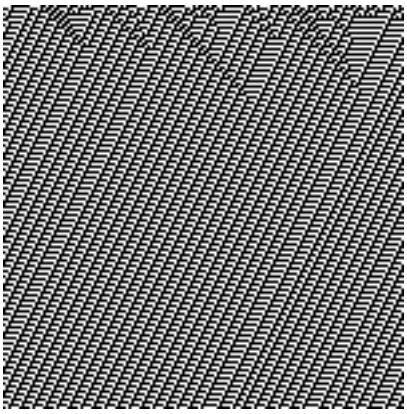

Fig. 8. A good solution to the Tetris pattern. Rule #188

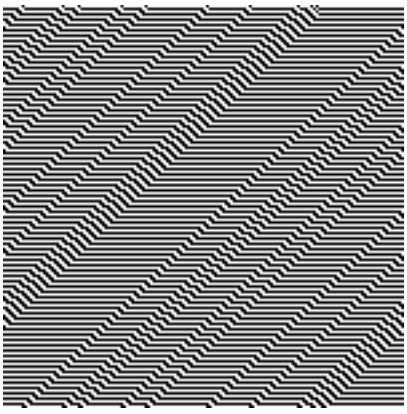

Fig. 9. A strong solution to the Diagonal pattern. Rule #142

F. Density

With density, our attempts to make a CA answer a question failed. Unfortunately, our tests and others [1] have shown that with small, simple rulesets this is apparently impossible. With this question, however, we realized that the density of our random environments was normally distributed with a mean of 50%; the environment was random, but the density is the mean of 50 random bits. Therefore, density was being poorly tested; however, a redesign using uniformly random density has yielded the same results as before.

X. Conclusion

From our strongly positive results, we believe that it is true that the steady state of the environment is determined nearly entirely by the ruleset and is only minimally affected by the initial environment. We found that genetic algorithms are effective tools in finding good solutions to the problems we posed, though we found with density that they cannot solve problems which are unsolvable within the search space. Since the initial environment has so little effect on many of the rulesets we evolved, we can conclude that many of them would be ill-suited for answering questions about the initial environment.

A. Possible Further Investigations

The topic of using GAs to evolve CA rulesets is very broad, and this field certainly is one in which there could be any

number of interesting and worthwhile investigations.

Expanding on the work we completed in this project to include more cell states or larger neighborhoods is an obvious extension that could yield extremely interesting results at the cost of increased computational complexity. As it stands now, given that our solution space of 256 possible rulesets is very small, the advantages of using GA as a search tool for ideal solutions is debatable, since obtaining results from a random or exhaustive search of the entire space is not computationally infeasible nor difficult to implement.

Adding another dimension and expanding our same project to two dimensions would also have interesting implications and would greatly expand the search space. Also interesting would be to attempt to evolve a CA rule (probably one in two dimensions, since we doubt one dimension could have the needed complexity) capable of carrying out basic computation tasks (addition, subtraction, multiplication), although this is considerably more complex in scope, specifically when it comes to determining the fitness of solutions.

In any of these cases, we would most likely want to rework our software implementation as PHP is not meant for large-scale computations and can become difficult to work with when it is required to process large amounts of data.

REFERENCES

[1] J. Crutchfield, M. Mitchell, "The Evolution of Emergent Computation," [Online Document] Proceedings of the National Academy of Sciences 1995, [2005 Dec], Available at HTTP: http://www.santafe.edu/projects/evca/Papers/EvEmComp.pdf

[2] University of Illinois at Urbana-Champaign Genetic Algorithms Lab, [Online Directory] Available at HTTP: http://www-illigal.ge.uiuc.edu/index.php3