Evolving Novel Cellular Automaton Seeds Using Computational Pattern Producing Networks (CPPN) *

Josh Wolper[1], George Abraham[2]

***Abstract*— The goal of this study is to evolve novel seeds for Conway's Game of Life Cellular Automaton with Computational Pattern Producing Networks (CPPNs). We evolved our CPPNs using both Objective Search (implemented with NeuroEvolution of Augmenting Topologies) and Novelty Search. Three variations of Objective Search were explored: rewarding solution lifetime, rewarding mass and lifetime, and rewarding lifetime while punishing mass. Objective Search quickly evolved game of life solutions that converged to trivial combinations of small-order oscillators and still life solutions. We did not find interesting symmetries and results until we ran Novelty Search, which produced complex high period oscillators such as the Pentadecathlon and Pulsar. Although these tests failed to produce novel solutions or generative structures, the evolutionary computation produced seeds that had a significantly higher lifetime than random seeds. Furthermore, the evolved solutions gave us insight into understanding the propagation of symmetries throughout the simulation, which demonstrates the potential for future use of CPPNs in studies of cellular automata or dynamical systems with inherent symmetries in the solution space.**

I. Introduction

Recent studies have suggested the importance of cellular automata in modeling large, or inherently stochastic data sets [6]. However, depending on the resolution of the cellular automaton being studied, machine learning algorithms are shown to out-perform traditional computational analysis methods [1]. Conway's Game of Life is a zero-player cellular automaton that has caught the interest of mathematicians, engineers, and computer scientists for decades. The Game of Life takes in an input set of dead and alive cells (called a "seed"), and performes a series of updates until the board is completely dead (i.e. empty). Given that a lot of sample solutions to this automaton tend to be symmetric, Computational Pattern Producing Networks (CPPNs) show promise in evolving novel Game of Life solutions due to the inherent symmetries of the CPPN basis fitness functions [7]. Although artificial neural networks have been applied to problems in cellular automaton before [1], this study is the first to apply CPPNs to a problem of this nature.

Our goal is to use Objective Search and Novelty Search to evolve novel seeds for the Game of Life cellular automaton. We hope to find seeds that produce complex, high-period oscillator solutions, or generative structures. Oscillators are solutions that, according to the update rules of the Game of Life, alternate between a fixed number of frames. Generative structures produce locomotive blocks that live forever in the Game of Life simulator. These types of solutions will be more precisely defined and explored later. We hope that use of these evolutionary computational methods will produce new solutions to the Game of Life simulator that give us insight onto the nature of the system as a whole.

*This work was conducted at Swarthmore College for an Adaptive Robotics Research Seminar in Fall of 2015.

[1] Swarthmore College Engineering Department. Swarthmore, PA 19081, USA. jwolper1@swarthmore.edu

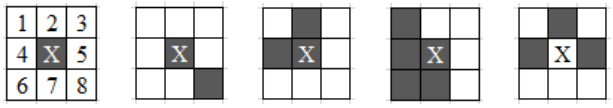
[2] Swarthmore College Engineering Department. Swarthmore, PA 19081, USA. gabraha1@swarthmore.edu

II. Background

A. Conway's Game of Life

Conway's Game of Life is a cellular automaton developed by John Conway in 1970 as an attempt to simplify the ideas presented by mathematician John von Neumann on a theoretical machine that could produce copies of itself [3]. This cellular automaton is a zero-player game, one that requires only a starting board state and a set of rules to play. The game seeks to model individual cells on a board living or dying based on a simple set of four rules. Each rule mimics a different aspect of life and death in a natural population, such as underpopulation, overpopulation, and reproduction. The rules are as follows (with examples provided in Fig. 1):

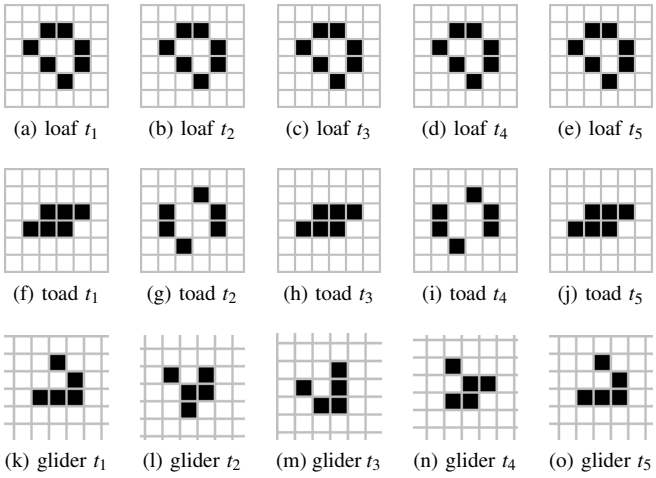
1) Any live cell with fewer than two live neighbors dies. (Underpopulation)
2) Any live cell with two or three live neighbors lives to the next generation.
3) Any live cell with more than three live neighbors dies. (Overpopulation)
4) Any dead cell with exactly three live neighbors becomes a live cell. (Reproduction)



(a) Neighbors (b) Rule 1 (c) Rule 2 (d) Rule 3 (e) Rule 4

Fig. 1: Depiction of how neighbors are defined and representative cases for each rule

There is a set of known organisms that can arise in the Game of Life: static still lifes (Fig. 2, row 1), dynamically changing but stationary oscillators (Fig. 2, row 2), and dynamically locomoting spaceships (Fig. 2, row 3). Each of these three types of organisms lives on forever, assuming they do not interact with one another. In addition, more complex organisms have been discovered since the game's conception: generative structures such as Gosper's Glider



(a) loaf t_1 (b) loaf t_2 (c) loaf t_3 (d) loaf t_4 (e) loaf t_5

(f) toad t_1 (g) toad t_2 (h) toad t_3 (i) toad t_4 (j) toad t_5

(k) glider t_1 (l) glider t_2 (m) glider t_3 (n) glider t_4 (o) glider t_5

Fig. 2: Five timesteps for three different organisms: the still life "loaf", the oscillator "toad", and the spaceship "glider"

Gun perpetually create new organisms, gliders, that locomote across the board and live on forever. The glider gun harkens back to von Neumann's original theory of a machine that could self replicate, showing the Game of Life's ability to successfully implement this idea in a markedly simple fashion.

In this version of the Game of Life, the board is setup to have a toroidal topology ("wraps around"), meaning that grid locations at one edge of the board have adjacent neighbors on the opposite side of the board. This distinction is made so that a small workspace is isolated for any evolved organisms and behaviors. In addition, this decision makes sense computationally as to avoid keeping track of a board that may extend indefinitely.

B. ANNs and Computational Pattern Producing Networks

Artificial neural networks (ANNs) are collections of directed weighted edges and nodes connected to form various topologies. Most neural networks are considered feedforward indicating that their connections are exclusively nonrecurrent (i.e. all connections must be from low tier nodes to high tier nodes). Each ANN has an activation function (often the sigmoid function) associated with each node that is used to process data coming into the node and determine how much the node should be activated, similar to the way real neurons are either activated or deactivated. Thus ANNs are usually broken down into three regions of nodes: input nodes, hidden nodes, and output nodes. ANNs are typically used by allowing some type of system to input values into the ANN and then employ the outputs to do some function. ANNs can have any number of hidden layers and nodes, with more hidden layers indicating more complex system that can develop more sensitive and specialized functionality.

Computational pattern producing networks (CPPNs) are a subclass of ANNs that differ exclusively in that each node does not share the same activation function. Instead, in a CPPN the activation function for each node can be any one of a preset group of functions (as shown in Fig. 3) such as the linear function and the Gaussian function [7]. CPPNs are especially useful when used to process images or grids, where individual x and y values of each pixel of an image may be used as input, and an output is then determined by the CPPN.

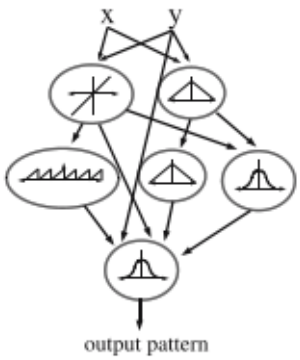


Fig. 3: Example CPPN depicting the variety of activation functions possible in each node [7]

C. NeuroEvolution of Augmenting Topologies

NeuroEvolution of Augmenting Topologies [8], known as NEAT, is a computational method for evolving populations of neural networks, based on a given goal fitness function, by altering network topology. The purpose of NEAT is to evolve ANNs - or, in our case, CPPNs - to solve tasks involving a complex sequences of decisions. NEAT works by starting with the simplest, most minimal network configuration, then gradually adding complexity by creating new nodes and links throughout the phases of evolution. NEAT saves genetic encodings of each member of the ANN population, and on each time step, may modify an existing connection weight or add new nodes and connections based on input probability parameters. NEAT then evaluates the fitness of each ANN member and with probability proportional to fitness selects members as parents for the next generation.

NEAT keeps track of the history of each member throughout successive generations through recording the number of times a new gene appears in the composition of an individual genome. This value, called the innovation number, allows NEAT to perform crossover without expensive topological calculations. These historical markings also allow NEAT to divide gene populations into separate species. Individuals are evolved based on comparison within their respective species, which allows topological changes throughout generations to be preserved.

Another useful feature of NEAT is that complexification happens in a minimal manner (i.e. final solutions are of minimal complexity). All of these features combine to make NEAT a reasonably fast, memory efficient algorithm for evolutionary computation. Previous experiments [2] have demonstrated the success and versatility of NEAT, specifically in the context of evolving CPPNs. Therefore, NEAT

is our primary learning algorithm for performing objective search in this study.

D. Novelty Search using NEAT

Although NEAT has demonstrated a lot of success as an evolutionary computation algorithm, a major criticism of NEAT is that it tends to converge to non-optimal solutions when confronted with deceptive tasks. In a previous study [5], when a robot was confronted with a deceptive maze problem that required the robot to initially travel away from the desired goal state, a computational mechanism known as Novelty Search outperformed NEAT by evolving a wide variety of solutions that broke out of a local maximum. Hence, we would like to use Novelty Search to evolve a diversity of different cellular automaton seeds, given that we predict NEAT will evolve mostly previously known solutions.

Novelty Search works similarly to NEAT, except, instead of having a fitness function geared towards a specific objective, the fitness function is geared towards evolving solutions that are the least similar to other solutions in the population. In other words, Novelty Search keeps track of the most novel behaviors in an archive, then performs evolution based on solutions most different from the archived solutions. Novelty amongst solutions is determined by a *sparseness* metric, which relies on some distance metric between two sets of solutions. This distance metric can be anything appropriate for comparing solutions depending on the task involved, but is often the Euclidean distance metric between points in the solution space. The sparseness of a solution, x, in the solution space with k solutions in the archive is given by:

$$\rho(x) = \frac{1}{k}\sum_{i=0}^{k} dist(x, \mu_i)$$

where k is an experimentally determined constant, and μ_i is the ith entry of the novelty archive. The candidates with the highest sparseness are added into the archive, and prioritized in the evolutionary process in the same way as high fitness individuals in NEAT. Thus the hope is that Novelty Search will find more interesting new solutions to the Game of Life cellular automaton.

III. Experimental Setup

A. CPPN Setup

Since the starting seed patterns for the Game of Life are inherently an image or a grid, CPPNs are the perfect choice to try to develop seed patterns using different evolutionary algorithms. In order to allow for diverse behaviors 100 CPPNs are in the initial population and evolve over the course of 10 generations for 500 simulation steps each (See Appendix C for general experiment parameters and Appendix D for MultiNEAT evolution parameter details.) Since we are processing an image grid space by space, we first assign each grid space a coordinate pair value (Fig. 4). The center space is defined as (0,0), and each grid space is then assigned a coordinate pair with both x and y in the range [-1,1]. Thus the first three CPPN inputs are x coordinate, y coordinate, and bias. However, a fourth input was chosen to give the CPPN another metric to process. This fourth parameter was selected to be the Euclidean distance from the given grid space to a defined (0,0) position on the grid. The decision to include this input was informed by prior work done on CPPN-NEAT [2] in which it was shown that adding distance as a parameter to the evolving CPPNs introduced organic symmetries and repetitions to the evolved solutions— two features that are desirable when producing seed patterns for the Game of Life.

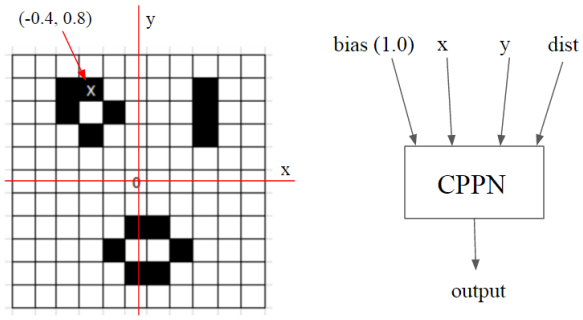


Fig. 4: Example Game of Life seed illustrating the underlying coordinate system as well as the inputs and outputs to the CPPN

The output received from the CPPN is a float in the range [0,1], and this was simply filtered into either alive or dead using the rule that if the output was less than 0.5, the cell was dead, otherwise the cell would be alive. In this way the CPPN could process each grid space and determine whether or not the given cell is dead or alive, producing a full Game of Life seed pattern.

B. Fitness Function Definitions

In order to formulate fitness functions that evaluate underlying characteristics of the seeds evolved by the CPPNs some important variables must be calculated during simulation. Two of these variables are related to different lifetimes of the solution throughout simulation time and are directly linked to the way that the simulation was written. During simulation three different boards are available in memory at any given time step: the board before the current time step (prevBoard), the currently displayed board (currentBoard), and the next board to be displayed (updateBoard) (Fig. 5). Two important variables are calculated by taking advantage of this system: $lifetimeUntilStillLife$ and $lifetimeUntilOscillator$. The first of these variables simply counts the number of simulation steps that occur before a purely still life solution is reached, or more simply whenever it is found that currentBoard = updateBoard. Alternatively, $lifetimeUntilOscillator$ counts the number of simulation steps until a pure period two oscillator solution is found by counting until prevBoard = updateBoard. This indicates that the solution is either all period two oscillators or a combination of period two oscillators and still lifes. Finally, the third important variable

is simply *endMass* or the total number of living cells at the end of the simulation runtime.

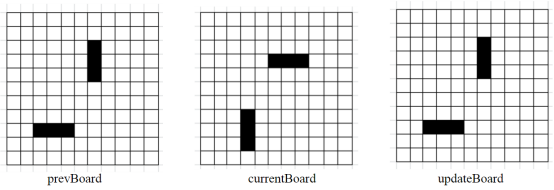


Fig. 5: Graphic illustrating the boards saved in memory during simulation

These three variables are normalized by two main variables that are native to our Game of Life class: $maxLifetime$ and $granularity$ where the first is simply the maximum number of simulation steps and the latter is the height (in number of cells) of the board. These five values together make up all three fitness functions defined for all experiments. If we define L_m to be our maximum lifetime, L_O to be our lifetime until an oscillator is evolved, L_S as our lifetime until still-life, g as the number of rows and columns in our board, and M as the mass of our board at the final time step, then our three defined fitness functions are as follows:

1)
$$f_1 = \frac{L_S}{L_m}$$

2)
$$f_2 = \frac{L_S}{2 * L_m} + \frac{M}{2 * g^2}$$

3)
$$f_3 = \frac{L_O}{L_m} - \frac{M}{2 * g^2}$$

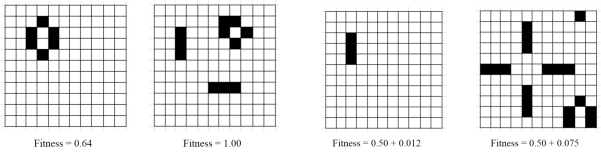
The first function rewards solutions that live on past still life solutions, this includes any solution with an oscillator or glider. The second function rewards a solution both for not converging to a simple still life solution but also for having a high mass at the end of the simulation, this ideally rewards larger oscillators and spaceships. Finally, the third function rewards solutions that live on past a simple period two oscillator configuration while also punishing solutions that have high mass at the end of the simulation. Fig. 6 shows some examples of board states and the fitnesses they would receive from each fitness function.

C. Novelty Search Setup

In order to encourage diverse solutions Novelty Search is also explored as an alternative to objective search. In this setup this simply indicates evolving the CPPNs using a sparseness metric instead of a fitness metric. This sparseness seeks to measure how different one solution is from the solutions most similar to it. In this setup, sparseness is calculated using a system of "behaviors" that are compared to one another. Each seed has a behavior associated with it and is simply a tuple with a length equal to the number of grid spaces in our seed patterns. Each element of the behavior tuple is either (0,0) indicating that the given grid space has a dead cell, or it has the coordinate pair associated with the grid space to indicate a living cell. We then compare the k nearest neighbors by finding the Euclidean distance between each pair of behavior elements for each pairing with the nearby neighbors.

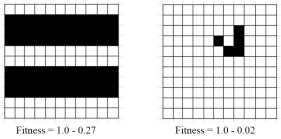
It is important to note that the behavior tuples are formed from the initial seed pattern generated by the CPPN, rather than the final state of the game board after the game has been played. This decision was made to avoid a subtle pitfall of our experimental setup. More specifically, a solution that has identical end behavior to another solution may receive a high sparseness metric if it is ever so slightly out of phase from the other solution. Thus any number of identical end solutions would all have a high sparseness metric given that they are all out of phase from one another when the simulation ends. More simply, an assumption of our setup is that we cannot distinguish between identical or even similar solutions at the end of the simulation because solutions may evolve at any point in the simulations, and often do evolve out of phase from one another.

When the CPPNs are evolved using Novelty Search and the sparseness metric, the most objectively fit chromosome is saved in each generation. This allows not only the most novel, but also the most fit end behaviors to emerge from this search. As such, a fitness function is still necessary for Novelty Search. See Appendix E for more details on the Novelty Search parameter setups.



(a) Fitness Function 1

(b) Fitness Function 2



(c) Fitness Function 3

Fig. 6: Example game boards and associated fitnesses for each fitness function. Note the fitnesses are broken down into their terms for clarity.

For details on the code used to implement these experiments see Appendix A.

IV. Results

A. Fitness Functions

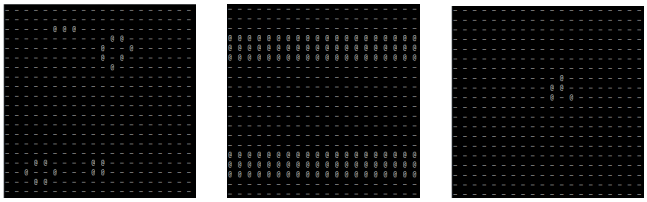
Before exploring the notable differences between evolving the CPPNs using objective search and evolving them using Novelty Search, it is important to discuss the common solutions found by each of our fitness function, and discuss why one function in particular was chosen to focus on.

Despite our expectation, many of our fitness functions led to extremely similar end behavior. Our first fitness function rewarded the seed pattern's lifetime until a still life solution, thus we expected it to leave behind seed patterns that would converge to pure still life solutions. This expectation was correct, however this fitness function tended to produce similar and somewhat trivial end behavior in every generation: simple patterns of small still lifes and occasionally between one and four oscillators of period two called "blinkers" (Fig. 7a).

Our second fitness function both rewarded lifetime until a pure still life solution as well as the total mass of end behavior solution. We expected this would produce more connected end behavior, as opposed to the discrete organisms formed from rewarding still lifetime alone. Unfortunately, this fitness function scarcely ever produced anything save for high period oscillators we deemed "wave oscillators" as they have bands that move up and down the board in wave like patterns (Fig. 7b). We marked this as a trivial solution despite its high period since it is only possible due to the toroidal nature of the board and would not exist otherwise.

Finally, our third fitness function gives some desired results by specifically rewarding seeds that live longer before converging to a period two oscillator solution and punishing high end state mass. The reward seeks to avoid the solutions common in our first fitness function such as the still life and blinker combinations, while the punishment seeks to eschew the wave oscillator solution so prevalent in our second fitness function as this solution had consistently high end state masses. Instead of converging to our previous, more trivial solutions, this fitness function predominantly evolves gliders (Fig. 7c).

The end behaviors from the third fitness function were considered to be the closest to a desired solution as these gliders are inherently formed out of a dynamic mass of cells as the game progresses which is a feature of Gosper's glider gun. Thus, all attention was focused on experimenting with fitness function three as it showed the most promise in finding generative structures.



(a) Still lifes, Blinker (b) Wave Oscillator (c) Glider

Fig. 7: Examples of solutions generated by fitness functions 1, 2, and 3 respectively

B. Objective Search vs. Novelty Search

In order to explore the different effects of evolving the CPPNs using objective search versus with Novelty Search 10 trials of 10 generations of 100 CPPNs were run, five trials for objective search and five for Novelty Search. The fitness function used for both objective and Novelty Search was the fitness function rewarding lifetime until a period two oscillator solution as well as punishing high end state mass.

When evolving the CPPNs using objective search, it was found that every trial performed almost identically to each other. An immediate, high fitness solution was found within one to two generations, and then propagated throughout the following generations. Thus, when evolving the CPPNs using objective fitness it was found that only two types of behaviors emerged, and they are markedly similar to one another: a solo glider solution, and a glider with still life solution (Fig. 9a). However, it is important to note that the glider with still life solution is less fit than the solo glider solution (higher mass) and therefore is "bred" out of the population as quickly as possible. As such, 47 out of 50 total generations converged to a lone glider as their end behavior.

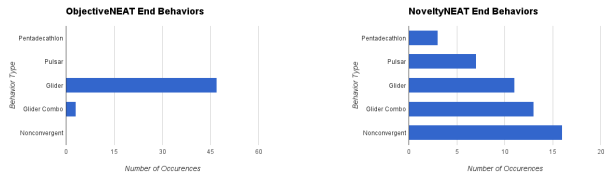
Conversely, when evolving the CPPNs using Novelty Search and the sparseness metric, five different types of end behaviors emerged, three of them unseen by objective search. In addition to the glider and glider with still life end behaviors, Novelty Search also evolved two unique high period oscillators: one of period three named the "pulsar" (Fig. 8a, and see Appendix G for individual stills) and one of period 15 named the "pentadecathlon" (Fig. 8b, and see Appendix H for individual stills). Finally, the final new behavior was simply non-convergence- essentially indicating that the behavior needed a longer maximum lifetime to fully converge. Overall, Novelty Search produced much more diverse behaviors as well as evolving some highly complex oscillators that no previous trial could produce. In fact, a complex oscillator was evolved as the highest fitness solution in 10 out of 50 total generations using Novelty Search (Fig. 9b).

(a) Pulsar, period 3 (b) Pentadecathlon, period 15

Fig. 8: Complex, high period oscillators moved to the center of the board for ease of viewing (these complex oscillators often formed around the edges of the board, making them difficult to view in their original form)

C. Random Encoding vs. Generative Encoding

Given that most of our trials in experiments above seemed to converge to optimal solutions within the first generation, this begs the question of whether or not our Evolutionary Computation methods can produce better results than random

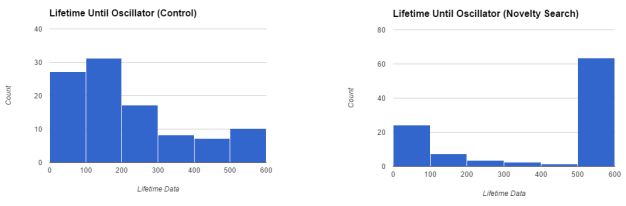


(a) Objective NEAT Behavior (b) Novelty NEAT Behavior

Fig. 9: Comparison of the frequency of different types of end behavior for ObjectiveNEAT and NoveltyNEAT

seeds. Therefore, we ran a test that compared the lifetimes of 100 randomly generated seed patterns to lifetime data obtained from the entire population of the 10th generation of our most successful evolutionary experiment (using the third objective fitness function with Novelty Search). Fig. 10 displays the data obtained after 100 runs of both experiments in histograms. The random data appears to be more uniformly spread with a slight skew right, while the Novelty Search data seems to take on an inherently beta distribution shape. Although both methods appear to produce oscillator solutions, it is important to note that most of the oscillators produced in the random seeds were trivial, low-period solutions while Novelty Search evolved complex oscillators such as the Pulsar (see Appendix G) and Pentadecathlon (see Appendix H).

Nevertheless, attempts were made to normalize both data sets and compare them using a standard t-test. However, as seen in Fig. 11, even after an inverse cumulative normal distribution transformation was applied to both data sets, the Novelty Search data still preserved its strong beta shape, making a t-test inappropriate to apply. Therefore, to compare the two data sets, we applied the Kruskal-Walis H-test [4]. This test applies because our two data sets are independent, and they have the same sample size. Given our two data sets ($g = 2$) with 100 observations ($N = 100$), we see that our Kruskal-Wallis test rejects the null hypothesis that our random data and Novelty Search data come from the same underlying distribution with $p = 2.927 \times 10^{-6}$ (see Appendix F for more detail regarding the computation of our Kruskal-Wallis test statistic).



(a) Random Data (b) Novelty Search Data

Fig. 10: These histograms display the lifetime until oscillator data for 100 random seeds and 100 seeds evolved for 10 generations with Novelty Search.

See Appendix B for links to videos showcasing results

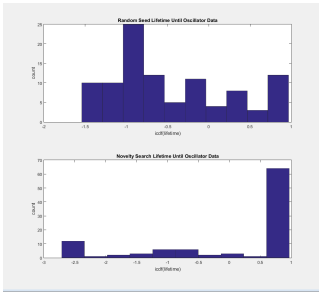


Fig. 11: These histograms display the inverse normal cdf applied to the lifetime until oscillator data for 100 random seeds and 100 seeds evolved for 10 generations with Novelty Search. This attempt to normalize our data did not work for the Novelty Search data (even after normalization, it has an inherent beta shape).

such as typical runs for each fitness function and a visualization of both the pulsar and the pentadecathlon.

V. Discussion

Although generative structures were never found in these experiments quite a bit was concretely found about evolving cellular automaton seeds using CPPNs. First, CPPNs are extremely proficient at developing solutions that follow a fitness function correctly, so well that the fitness often jumps to the max fitness within only a few generations. As such, we may say that given the correct fitness function, most desired behaviors may be isolated using CPPN evolution. For example, it was simple to isolate the glider end behavior from the wave oscillator behavior by simply punishing higher end state masses.

Second, and perhaps more notably, high period oscillators arise from evolving the CPPNs using Novelty Search. Since these solutions are so inherently symmetrical it seems to indicate that Novelty Search opens the doors to CPPNs producing symmetrical and repeating patterns. Symmetry and repetition seemed oddly absent in most trials during simulation despite CPPN's reputation for evolving those exact traits, especially when processing images. Perhaps the addition of novelty allowed these symmetrical solutions to dominate some generations by sheer uniqueness, as they would not perform particularly well in fitness function 3, especially compared to a lone glider which clearly has a much lower end mass.

A. *Significance of Results*

Although we did not evolve new oscillators or generative structures, our results demonstrate the ability of evolutionary computational methods to adapt and produce optimal results. Objective search evolved mostly trivial (but high-lifetime) solutions while Novelty Search produced a wide variety of solutions with different axes of symmetry. Our results confirm that CPPNs with Novelty Search may be used to solve problems with inherent symmetries, which is an important feature of many different cellular automata. In general, our CPPN evolution trials helped us gain insight onto the nature

of the Game of Life solutions, which, in some sense, is equally as valuable as more solutions. Newer applications of cellular automata include using them to analyze patterns in large data sets such as land usage [1]. As a computational method that is inherently designed to understand symmetry, CPPNs pose a lot of potential in furthering our understanding of cellular automata.

More generally, despite that Genetic Algorithms and Artificial Neural Networks have been used in research involving dynamical systems (cellular automata are a subset of discrete dynamical systems), there has not been much research involving CPPNs with cellular automata or dynamical systems. Depending on the symmetries of the solution space, CPPNs may be useful in evolving solutions to dynamical systems with complex notions of frequency and oscillation, as demonstrated in the ability to evolve high-order oscillators here. Again, despite that we were not able to find new analytic solutions to our specific system, CPPNs may be useful in solving other systems, or at least, helping systems researchers gain intuition on a complex system.

B. Criticism of Approach

That being said, although our evolutionary computational methods appeared to give us significantly better results than pure random seeds, the results of our study still beg the question of whether or not evolutionary computation is even appropriate to apply to this Game of Life scenario. Our results do not appear to be ”intelligent” in any way; in fact, most of our fitness functions were edited to converge to specific results we wanted to see. The computer did not act intelligently to find new solutions in any way, and despite our results being interesting, they do not show evidence that these computational methods function as a more general purpose artificial intelligence.

As for our approach, there was one assumption made in our setup that could potentially have led to issues- this is our toroidal topology. This topology was chosen for reasons of minimizing computational intensity and confusion regarding edge neighbors. But generative structures produce new organisms indefinitely, and as such, the mass of the system is always increasing. As such, a toroidal topology in which newly formed organisms naturally wrap around the board might cause problems because these organisms may then destroy the very structures that brought them into existence. Thus, it is highly possible that the toroidal topology was the main reason generative structures were not evolved by either objective search or by Novelty Search.

C. Future Study

Given these criticisms of our study, there are a lot of changes we can make in future experiments. One of the major disappointments in this project was that we did not evolve generative structures in any of our trials. This happened, partially, due to the toroidal topology of our game of life implementation; the well-known glider gun generative structure, for example, does not work on a toroidal game board because the gliders generated travel back around and cancel the blocks forming the generative structure. If we had tried implementing an unbounded board (isomorphic to $\mathbb{Z} \times \mathbb{Z}$), then we may have evolved generative solutions. Our main concern regarding implementation of our unbounded board, however, was the amount of memory such an implementation would take up. Once the game proceeds to off the edge of the board, we must make subsequent game updates based off of cellular data off of our main game board. An implementation idea for this is to apply our entire game of life simulation to a board of size $T_{max} \times T_{max}$, and only considering the middle portion of the board (the normal granularity) in update decisions, lifetime data, and simulation videos. This, however, would potentially harm the runtime of our program.

Aside from generative structures, another goal of our project was to evolve more complex oscillator solutions. While we had instances of Pulsars and the Pentadecatholon, we did not find any novel oscillators. Most oscillator solutions evolved were combinations of blinkers - a very trivial oscillator of period 2 (Fig. 12). If we concatenate 2 blinkers, we see that we can form a toad, an oscillator with the same period, but more complex behavior as a whole. Furthermore, we can also compose the pulsar (a period 3 oscillator) out of concatenations of blinkers too, by this same logic. Hence, we propose that another interesting experiment may involve setting up 2 layers of CPPNs: one which evolves some sort of basis configuration of blocks (for example, the blinker), and one which composes a board based off of concatenations of this basis block. As opposed to composing a board by individual blocks, this second CPPN will evolve to find complex concatenations of given basis blocks, and this may cause more complex oscillators to evolve.

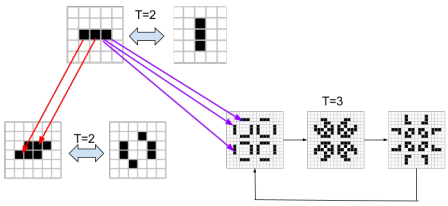


Fig. 12: This shows how the toad (left) and pulsar (right) can be made up of a combination of blinker oscillator structures (top), and how the overlap in blocks between a base oscillator can produce a more complex, higher-period oscillator, and an oscillator of the same period. This is the motivation for the experiment outlined in Fig. 13.

As a final extension to this study, it would be compelling to explore the effects of board resolution or more specifically the board granularity value on solutions that are evolved. For example, if the board were extended to a 100 by 100 grid,

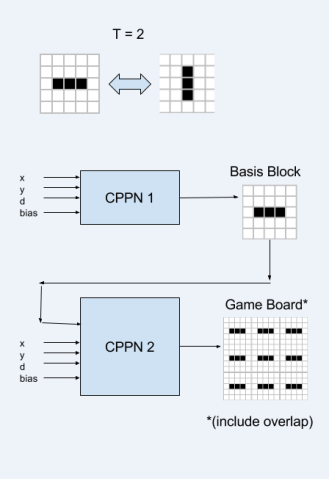


Fig. 13: This shows a proposed experimental setup with 2 CPPNs used to evolve a game board composed of basis blocks (basis shown here is the blinker oscillator). This setup may allow us to find more complex, high-period oscillators.

this may potentially eschew the complications that arise from the toroidal topology. Even further, this may allow for higher mass, higher period oscillators that may not be well defined or known yet.

VI. Conclusion

In this work we explore evolving cellular automaton seeds using CPPNs, expanding on extensive work done on finding organic symmetries and repetitions using CPPNs for image grid processing by Cheney, MacCurdy, and Clune et. al [2]. In addition this explores an extension to Conway's classic Game of Life to see whether well known structures such as Gosper's glider gun may be evolved using an evolutionary algorithm such as NEAT. Although generative structures such as the glider gun were never found in this study, unique high mass high period oscillators were in fact evolved, and this is impressive in the sense that this stems from a machine learning extension of a game that already attempts to model organic phenomena. In fact, this result succinctly comes full circle on John von Neumann's original dream of a self replicating machine- NoveltyNEAT has evolved CPPNs that can organically produce long living organic solutions to the Game of Life. With more time and further study, there is no telling where deeper exploration of combining evolutionary algorithms with cellular automata will lead.

VII. Acknowledgments

Thanks to Kenneth O. Stanley for fielding our inquiries about the usage of MultiNEAT and directing us to Peter Chervenski for further help on the library. Thanks to Peter Chervenski for supplying us with MultiNEAT starter code. Thanks also to Lisa Meeden for advising us throughout the stages of this work and providing us with the framework necessary to make it possible.

References

[1] Omar Charif, Reine-Maria Basse, Hichem Omrani, and Philippe Trigano, *Cellular automata model based on machine learning methods for simulating land use change*, Winter Simulation Conference (Oliver Rose and Adelinde M. Uhrmacher, eds.), WSC, 2012, pp. 163:1–163:12.

[2] Nick Cheney, Robert MacCurdy, Jeff Clune, and Hod Lipson, *Unshackling evolution: Evolving soft robots with multiple materials and a powerful generative encoding*, SIGEVOlution **7** (2014), no. 1, 11–23.

[3] Martin Gardner, *Mathematical games: The fantastic combinations of john conway's new solitaire game "life"*, Scientific American (1970), no. 223, 120–123.

[4] WH Kruskal and WA Wallis, *Use of ranks in one-criterion variance analysis*, Journal of the American Statistical Association (1952), 583–621.

[5] Joel Lehman and Kenneth O. Stanley, *Abandoning objectives: Evolution through the search for novelty alone*, Evol. Comput. **19** (2011), no. 2, 189–223.

[6] Sushil J. Louis and Gary L. Raines, *Genetic algorithm calibration of probabilistic cellular automata for modeling mining permit activity*, ICTAI, IEEE Computer Society, 2003, pp. 515–519.

[7] Kenneth O. Stanley, *Compositional pattern producing networks: A novel abstraction of development*, Genetic Programming and Evolvable Machines **8** (2007), no. 2, 131–162.

[8] Kenneth O. Stanley and Risto Miikkulainen, *Competitive coevolution through evolutionary complexification*, Journal of Artificial Intelligence Research **21** (2004), 63–100.

VIII. Appendix

A. Outline of Code Utilized and Implemented

To implement the desired experimental setup a few pieces of previously written software were utilized. For the CPPN-NEAT implementation Peter Chervenski's MultiNEAT library was used, as well as starter code kindly provided by Chervenski himself. For all visualizations of the simulations Josh Wolper and Amelia Erskine's Pthreaded Game of Life simulator written in C was utilized to quickly run simulations and read in text files describing seed patterns. All other code was original, including an implementation of the Game of Life written in Python so that the simulations could be run without a text file input during evolution as well as so that various new variables may be calculated at simulation run time.

B. Video Links

Video	Link
Fitness Function 1 Example	https://goo.gl/2Ouzw2
Fitness Function 2 Example	https://goo.gl/YDAiGl
Fitness Function 3 Example	https://goo.gl/ipta7L
Actual Pulsar Run	https://goo.gl/P3ooPH
Actual Pentadecathlon Run	https://goo.gl/KtkxXM
Normalized Pulsar	https://goo.gl/5HW3hU
Normalized Pentadecathlon Run	https://goo.gl/uPJZM9

TABLE I: Video Links

C. General Parameters

Parameter	Value
granularity	20
maxLifetime	100
livingThreshold	0.5
genNum	10

TABLE II: General Parameters

D. MultiNEAT Parameters

Parameter	Value
pop size	100
mutate activation prob	0.0
activation mutation max power	0.5
min activation	0.05
max activation	6.0
mutate neuron activation type prob	0.03
act. function signed sigmoid prob	1.0
act. function unsigned sigmoid prob	1.0
act. function tanh prob	1.0
act. function tanh cubic prob	1.0
act. function signed step prob	1.0
act. function unsigned step prob	1.0
act. function signed gaussian prob	1.0
act. function unsigned gaussian prob	1.0
act. function absolute value prob	1.0
act. function signed sine prob	1.0
act. function unsigned sine prob	1.0
act. function linear prob	1.0

TABLE III: MultiNEAT Parameters

E. Novelty Parameters

Parameter	Value
k	5
limit	30
threshold	0.2
pointLength	2
behaviorLength	$granularity^2$
maxDistance	$2 * \sqrt{2} * behaviorLength$

TABLE IV: Novelty Parameters

F. Kruskal-Wallis Test Computations

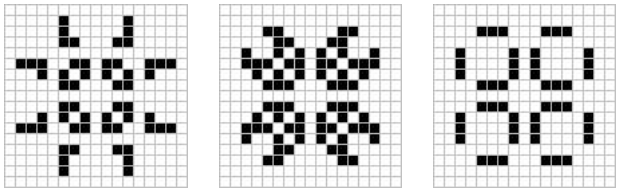
Given g groups of data (2, in our case) and N observations in each group (100), we can compute the H statistic by:

$$H = (N-1)\frac{\sum_{i=1}^{g} n_i(\bar{r}_i - \bar{r})^2}{\sum_{i=1}^{g}\sum_{j=1}^{n_i}(r_{ij} - \bar{r})^2}$$

where n_i is the number of observations in group i, r_{ij} is the rank of observation j in group i, $\bar{r}_i$ is the average rank of all observations in group i, and $\bar{r} = \frac{1}{2}(N+1)$.

We input our data sets into MATLAB and used the kruskalwallis command, which gave us our result based on computation of the test statistic above. See the figure below for the exact MATLAB output of our system.

G. Pulsar Stills



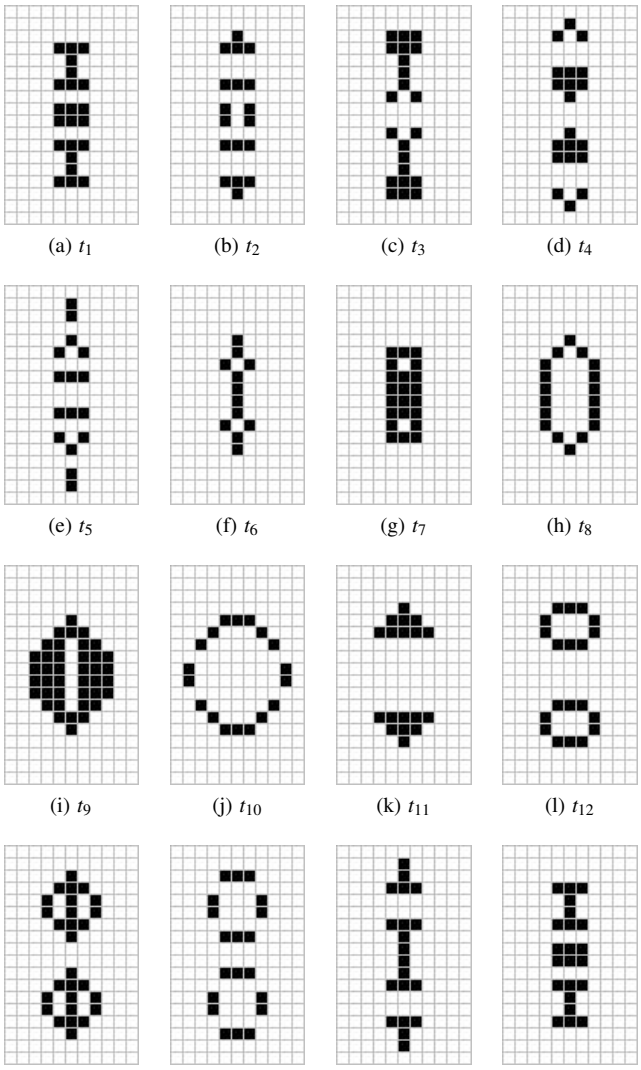
(a) t_1 (b) t_2 (c) t_3

Fig. 15: All 3 states of the period 3 oscillator, the pulsar

```
>> [p,tbl,stats] = kruskalwallis([CD0 ED0],[],'off')

p =

   2.9277e-06


tbl =

    'Source'     'SS'           'df'     'MS'           'Chi-sq'     'Prob>Chi-sq'
    'Columns'    [6.9676e+04]   [  1]   [6.9676e+04]   [21.8634]    [ 2.9277e-06]
    'Error'      [5.6452e+05]   [198]   [2.8511e+03]          []              []
    'Total'      [    634193]   [199]             []          []              []


stats =

       gnames: [2x1 char]
            n: [100 100]
       source: 'kruskalwallis'
    meanranks: [81.8350 119.1650]
        sumt: 389484
```

Fig. 14: This shows the numerical output of our Kruskal-Wallis test performed on our random seed and Novelty Search data in MATLAB.

H. Pentadecathlon Stills



(a) t_1 (b) t_2 (c) t_3 (d) t_4
(e) t_5 (f) t_6 (g) t_7 (h) t_8
(i) t_9 (j) t_{10} (k) t_{11} (l) t_{12}
(m) t_{13} (n) t_{14} (o) t_{15} (p) t_{16}

Fig. 16: All 15 states of the period 15 oscillator, the pentadecathlon