"Looking for Life"

Finding Complexity in the CA Landscape

Artificial Life Project
EASY MSc
Department of Informatics
University of Sussex - Brighton - United Kingdom

1 Introduction

Complex emergent behaviour is seen throughout nature, and in many different guises. Bee colonies select nest sites and forage for food with an extremely high degree of efficiency [13, 14]. Ant colonies zone their graveyards, food stores and nests at a perfect equidistance from one-another [15]. Flocks of birds, schools of fish, swarms of insects and herds of mammals move with the grace of a coherent unit. There is no central commander in such systems. No conductor orchestrating the behaviour of each individual. The order emerges naturally from the dynamics of the underlying system.

How does such complex behaviour emerge and persist from the simple interaction of a system's components? Are there general properties that a system must possess in order for this "emergence" to become possible?

Cellular Automata (and other discrete dynamical networks) have become a central tool in the study of such questions. It is hoped that by examining the mathematics of complex dynamical systems we will be able to better understand these basic principles of emergence.

In general a dynamical system (and specifically a CA) will result in one of three types of behaviour. Chaotic, ordered (periodic / fixed) or complex. In studying von Neumann CAs (with 2 states and a 9 cell neighbourhood) we find that the VAST majority of the 10154 rulesets will result in either chaotic or ordered behaviour. In fact, only a tiny proportion of rulesets have been found which show any signs of complex behaviour at all.

A classic example is John Conway's "Game of Life" whose emergent behaviour is proven to have the computational power of a Universal Turing machine. But what is it that makes this ruleset stand out from the other less interesting ones? Are there any significant factors which may be used to define and navigate the vast CA rulespace in order to find other rulesets with similar emergent potency?

It is the aim of this paper to highlight two "factors" which the author believes fulfil these criteria.

A mutational GA which successfully traverses the CA landscape using these factors will be defined and the resulting "interesting" rulesets described.

Finally, a look at the complexity of these results shall be given along with a discussion of future projects which will be undertaken to further investigate the significance of these factors.

2 Current Methods

Firstly, it is important to examine "*some*" of the existing methods used for mapping and classifying the areas of complexity within the CA landscape. A detailed and thorough analysis of each approach is not given within this paper but may be found within the references provided for each.

Wolfram Classes

The first, quite general, approach in classifying CA rulesets came from Stephen Wolfram in 1984 [12]. He found that CAs displayed one of four classes of (dynamical) behaviour: fixed points (class I); limit cycles (class II); chaotic (class III); and "interesting" or "complex" behaviour (class IV). It is now understood that fixed points can be defined as limit cycles with a period of one, and so these classes can more generally defined as *order*, *chaos* and *complexity*.

λ Parameter

In the late 1980's Chris Langton defined the λ parameter [9]. This is simply defined as the "proportion of rules which result in a non quiescent state". In the Game of Life, 140 of the 512 rules lead to a non-ZERO state and so $\lambda = 0.2734$. It was thought that higher λ values would lead to chaotic behaviour, whilst lower λ values would result in more ordered behaviour. Norman Packard took this one step further by naming this critical λ value (λ_c) "*the edge of chaos*" (the transition zone between ordered and chaotic systems). However, it has since been shown that complex behaviour can be found many other λ values and so the critical "edge of chaos" shows no real significance in defining the regions of complexity (see Crutchfield, Mitchell et al [5, 6, 7]).

Z Parameter

Andy Wuensche, in 1996, performed an extensive study of the attractor basins found within all kinds of discrete dynamical networks [4]. Such networks are deterministic and dissipative (having an "arrow of time"); by this we mean that two trajectories through state-space may converge but will never diverge. Wuensche devised a method of, having found an attractor in state-space, stepping backwards in time to find all of the possible "pre images" (previous states) which may have lead to this attractor. By continuing this process until presented with only "garden of eden" states (states without any pre-images) a full transient tree can be drawn up, highlighting the full structure of the attractor basin.

Wuensche found that bushier transient trees (those with a higher in-degree of convergence) showed more ordered behaviour whilst sparser transient trees were more chaotic. Complexity lay somewhere in between. By defining a parameter (Z) as a measure of this "bushiness", it was possible to predict the expected class of a given CA.

Entropy Variance

A further measure outlined by Wuensche is Entropy Variance[1]. At each time-step every cell is in a particular "state" and makes a corresponding lookup from the CA rule-table. By recording the frequency distribution of these state lookups, we are able to measure the entropy of the system (equation 1).

(Equation 1) Shannon Entropy

$$S^{(t)} = -\sum_{i=1}^{2^k}\left(\left(Q_i^{(t)}/n\right)\times\log\left(Q_i^{(t)}/n\right)\right)$$

Where $Q_i^{(t)}$ is the lookup frequency of neighbourhood *i* at time *t*.

High entropy (a wide distribution of states found in the system) indicates a chaotic system; low entropy indicates an ordered system (perhaps cycling through just a handful of states). A fluctuation between periods of high and low entropy is an indication of a complex system.

[1] It is not clear where this measure first originated.

1/F Noise

Finally, a brief mention of a more recent development in defining the CA landscape. In 1998, Shigeru Ninagawa [2] showed that (for symmetric, outer-totalistic von Neumann CAs) the Game of Life is the only ruleset where the Fourier transformation of the time series of states exhibits 1/f fluctuation (sometimes referred to as "multiscale" entropy variance).

Subsequently Ninagawa has evolved other Cellular Automata (from within a wider rulespace) which match this property and which display similar complex emergent behaviour to that found in the Game of Life [3].

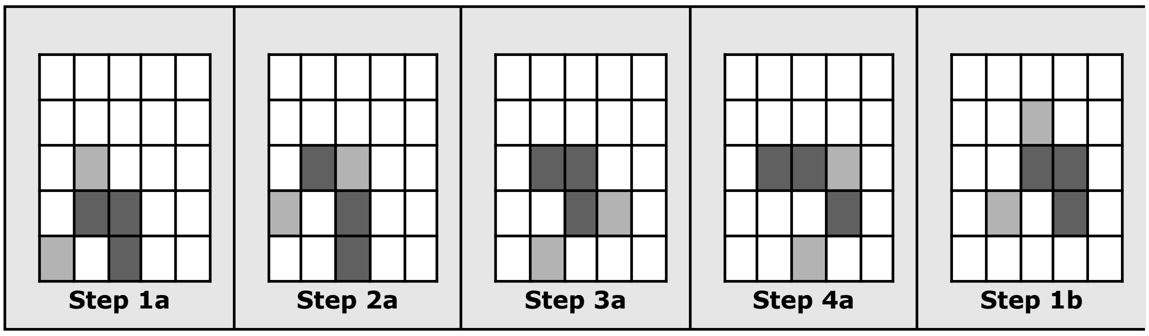
Each of the above methods aims to outline a "footprint" or "signature" that can be used in "looking for life" within the CA rulespace: either through specific λ values; through the bushiness of transient trees; through telltale entropy variance or through 1/f noise. The aim is shared. The remainder of this paper will outline and examine two further such "footprints"; the "*stay alive*" rate, and the "*population*" rate.

3 *Staying Alive and Population Rates*

Complex systems are made up of a large number of sub-components interacting in a non-linear fashion. Additionally, in vivisystems such as the human body, a bee colony, or a city, these sub-components are continually being replaced through the processes of birth and death. Somehow, the higher-level emergent system remains stable despite the continual death and regeneration of its constituent parts[2].

Examination of a single glider

A single glider within the Game of Life can be looked upon in similar terms; its form remains despite the continual birth and death of its constituent cells. This is pictured in figure 1 where newly born cells are highlighted with a lighter shade.



[2] Stable is used here in the general sense as opposed to the more formal "stability" used to describe attractors within dynamical systems.

(Figure 1) The lifecycle of a glider (new components are highlighted with a lighter shade)

During each phase in the lifecycle; a proportion of cells remain alive, a proportion of cells die out and a proportion of new cells are born (see table 1).

	Births	**Deaths**	**Stay Alives**	**Population**
1a – 2a	2	2	3	5 of 25
2a – 3a	2	2	3	5 of 25
3a – 4a	2	2	3	5 of 25
4a – 1b	2	2	3	5 of 25

(Table 1) Births, deaths and "stay alives" of a glider

So, a glider maintains a constant population of cells with a consistent birth and death rate of 40% (or conversely a stay-alive rate of 60%).

Examination of CAs in General

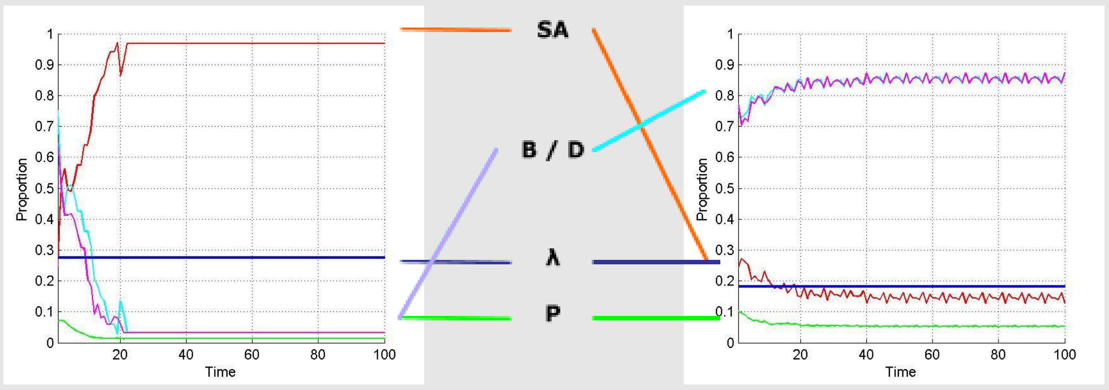
Before expanding this investigation to cover CAs in general, a more formal definition of terms needs to be given.

- **A "live" cell** is a cell in a non-quiescent state
- **A "dead" cell** is a cell in a quiescent state
- **The population (P)** is the % of "live" cells in the lattice at time t.
- **The birth rate (B)** is the % of p at time t which was "dead" at time t-1.
- **The death rate (D)** is the % of p at time t-1 which is "dead" at time t.
- **The stay alive rate (SA)** is the % of p at time t-1, which is also "live" at time t.

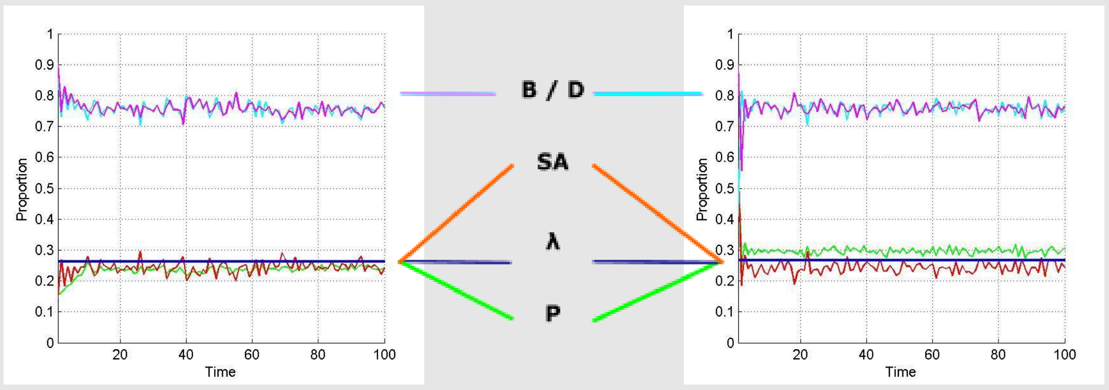
From this, it becomes possible to record these "rates" for any CA at any given time-step; and a natural progression is to examine the dynamics of how these rates change over time.

Figures 2a, 2b and 2c show these "rate-change graphs" for a number of different CAs. In each case, a CA is given a random initial state (with a population of approximately 10%), and is run for 100 time-steps. Each of the "rates" is recorded for each time-step and the data is plotted on a single graph along with the value of λ for that ruleset.

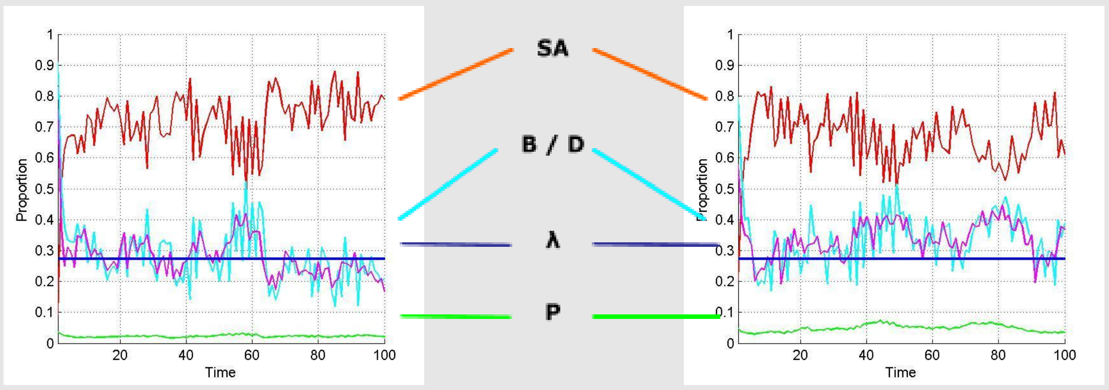
Figure 2a examines two arbitrarily chosen CAs which show **ORDERED** behaviour (quickly collapsing to fixed points and simple blinkers). Figure 2b examines two **CHAOTIC** CAs and 2c examines the **GAME OF LIFE**. All of the CAs have a λ value of approximately 0.27 (to try and compare like for like).



(**Figure 2a**) SA, B, D and P in **ORDERED** systems



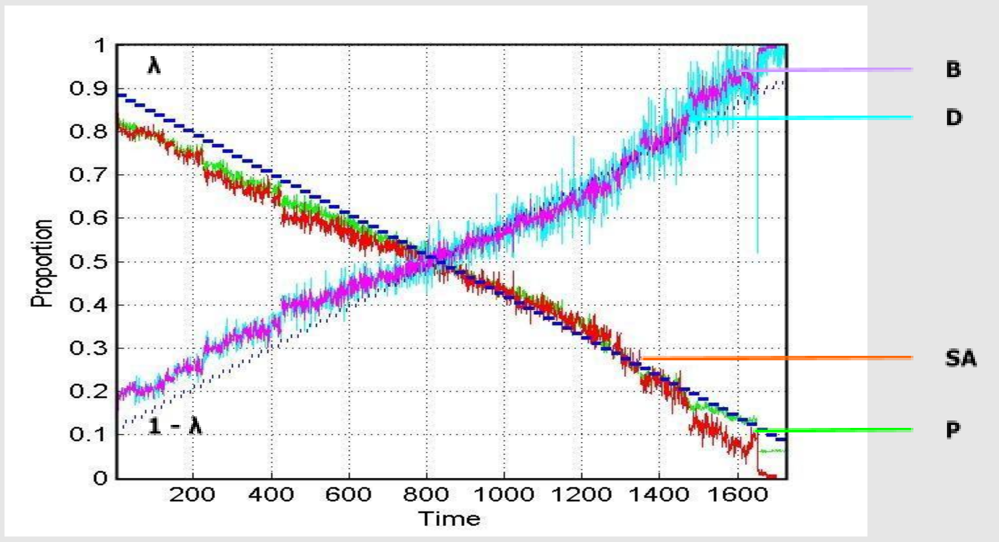
(**Figure 2b**) SA, B, D and P in **CHAOTIC** systems



(**Figure 2c**) SA, B, D and P in **GAME OF LIFE**

Although there are obvious differences between these three graphs, the primary distinction may not appear immediately significant. So, the next step is to examine how the "rates" vary for different values of λ.

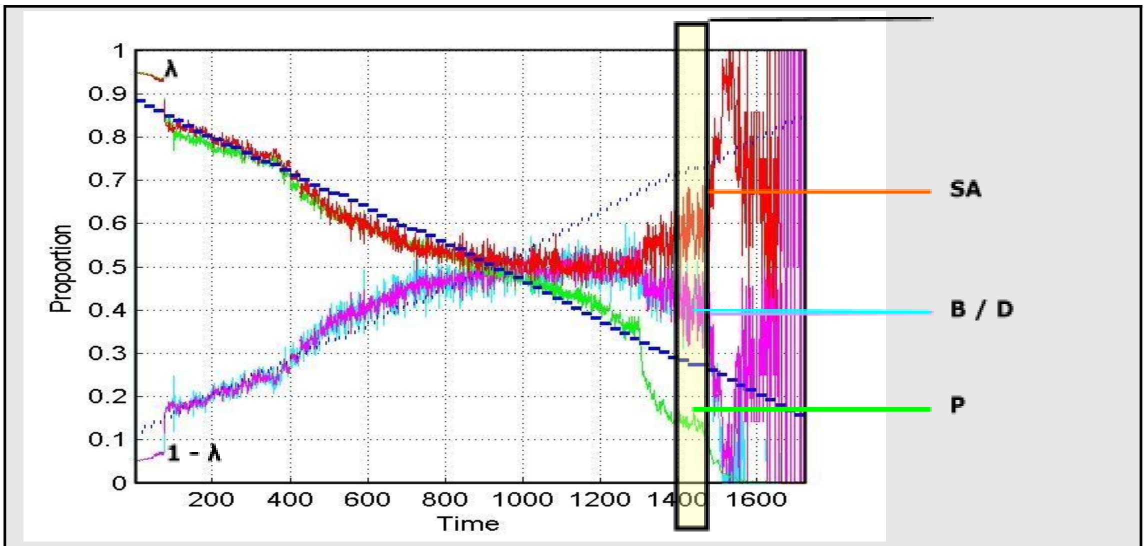
Starting with an initial ruleset with λ ≈ 0.9 a "rate-change graph" is plotted for 25 time-steps. The ruleset then has 1% of its rules "turned off" at random, (reducing λ by 0.1). The "rate-change graph" is then plotted for a further 25 time-steps. This process is continued until λ ≈ 0.1. The results are shown in figure 3a.



(**Figure 3a**) Walking through **CHAOS**

Next, the same process is performed, but this time, the "turning off" of rules is not done randomly. The rules are turned of in such a way as to bring the ruleset closer and closer to the Game of Life. The results for this are shown in figure 3b.

Game of Life Rules



(**Figure 3b**) Tuning in to the **GAME OF LIFE**

Observations

A number of possible observations can now be drawn from examining figures 2a, 2b, 2c, 3a and 3b.

For All Systems

1: In general the birth and death rates are approximately equal.

2: Death rate is intuitively 1-SA

For Chaotic Systems

3: The population and the SA rates closely follow the value of λ.

4: The birth and the death rates closely follow the value of 1-λ.

5: All rates maintain an obvious mean with a low, noisy variance.

For Ordered Systems

6: The population is significantly lower than λ.

7: The SA rates may be significantly different to λ.

8: The birth and the death rates may be significantly different to 1-λ.

9: All rates maintain a obvious mean with a low, ordered variance[3].

For Complex Systems

10: The population is significantly lower than λ.

11: The SA rates may be significantly different to λ.

[3] The more important factor here is the ORDERED nature of the variance – repeated patterns, or flatness. Mostly this is low in height, but order can be found with a higher change.

12: The <u>birth</u> and the <u>death</u> rates may be significantly different to <u>1-λ</u>.
13: The <u>mean</u> for each rate is not obvious with a high, <u>noisy variance</u>.

Through these observations, it's now possible to hypothesise what "footprints" may highlight complex rulesets.

A complex ruleset appears to show signs of both order and chaos. It maintains a disproportionately low population rate (signifying order) whilst also maintaining a high, noisy variance in SA, birth and death rate (signifying chaos). From observations 1 and 2, we can simplify this even further:-

"a complex system maintains a population rate disproportionately lower than λ whilst maintaining an high, noisy variance in SA rate[4]".

[4] Birth, Death or SA rate can be used equivalently. SA was chosen by preference.

4 Implementation of the CA Search

This chapter will describe a mutational GA that has been implemented to look for rulesets which display these "complex footprints".

MATLab

First, a quick look at the choice of programming environment. MATLAb is an excellent tool for the implementation of CAs. Its capacity for working with matrices is ideal for the storage and manipulation of the cell lattice, and rule-tables. Additionally, the strong graphing and imaging functionality allow for swift and simple result reporting; ideal for this report.

The final code is given in Appendix A, as best as possible the code is commented to highlight the key variables and functions.

CA Design

Having previously developed a genetically evolvable CA in MATLab, a large proportion of the CA design decisions have already been worked through and discussed (see [16] for a thorough breakdown of the implementation and encoding techniques).

In brief however; by limiting the search to 2-state CAs it is possible to make good and strong use of BINARY encoding techniques (a cell is either ALIVE=1 or DEAD=0). For a 9 cell neighbourhood (as in the Game of Life) each possible cell in the neighbourhood will be either 1 or 0. By mapping each position in the neighbourhood with a position in a 9 bit binary string we can represent each of the possible states with a single number ranging for 0 to 512 (see Figure 4). From here, an entire CA rule-table can now be represented by a single binary array of 512 bits.

Value = 0

32	2	64
16	1	4
256	8	128

Value = 391

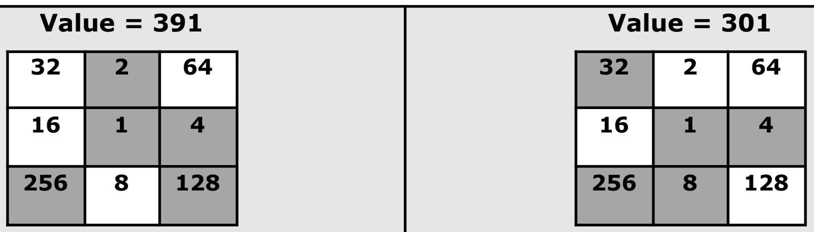
32	▨ 2	64
16	▨ 1	▨ 4
▨ 256	8	▨ 128

(Figure 4) Mapping neighbourhood to bit value in a binary string

Adding Symmetry

Symmetry is a major aspect of the study of CAs, in particular to the study of complexity within CAs. The Universal Computational power of the Game of Life could not exist without the ability of gliders to move in all directions and so collide (a feature made possible through symmetry of rules). As such, it is felt that the CA design should have the option of "enforced symmetry".

Figure 5 shows the rotation of a glider through 90 degrees and thus provides a mapping of neighbourhood positions (391 maps to 301). A similar mapping can also be given for 180 and 270 degree rotations.



Mapping

	1	2	4	8	16	32	64	128	256
90°	1	4	8	16	2	64	128	256	32

(Figure 5) Mapping neighbourhoods in rotational symmetry

By building up data arrays of these mappings it becomes a trivial exercise of table-lookup to determine the symmetries of any given CA state.

GA Design

This method of binary encoding enables an entire CA rule-table to be stored as an array of 512 bits; an ideal genotype for a genetic algorithm (GA). The next step is to define the style of algorithm which will best search the phenotypes of behaviour.

Looking back to figure 3b, it can be seen that as a ruleset mutates towards the Game of Life, the "footprint" (or fitness) increases. It was therefore decided that a purely mutational GA would be developed with the aim of slowly stepping towards greater and greater levels of complexity[5].

[5] This decision warrants a great deal more investigation and is discussed further within section 6

In brief, the aim is to find reasonable peaks in the fitness landscape and to randomly examine the local terrain looking for higher points. Once a higher peak is found the population should converge around it and begin its meandering again from here.

The basic mechanisms are provided in Algorithm 1. A population of random rulesets is created; the fitness of each is measured; a handful of the fittest parents are copied into the next population and the remaining children created by through their mutation.

```
Generate a POPULATION of random rulesets
FOR loop = 1 TO generation-time
    FOR EACH ruleset in POPULATION
        Calculate Fitness Score
    NEXT ruleset
    Sort POPULATION by fitness
    Empty NEW-POP
    FOR x = 1 TO no-of-parents-to-keep-alive
        NEW-POP = NEW-POP+ POPULATION(x)
        FOR y = 1 TO no-of-children-per-parent
            NEW-POP = NEW-POP + RANDMUTATE(POPULATION(x))
        NEXT y
    NEXT x
    POPULATION = NEW -POP
NEXT loop
```

Algorithm 1: The basic GA

Population Size and Mutation Rates

The choice of population size and of how large a "handful" to keep alive in each generation really warrants an in depth investigation beyond the scope of this report. However, through a process of trial and error it was found that a "keep alive rate" of around 10% worked particularly well, with a population of around 60-100[6].

Child rulesets are generated by copying a parent ruleset and randomly flipping a number of rules (whilst also ensuring that symmetry was maintained). The number of these mutations (the mutation rate) can potentially range from just a single symmetrical flip, to a full negation of the parent. Again, through trial and error, it was found that changing 10%-20% of the rules seemed to result in the greatest overall performance[7].

[6] Too high a population would result in too many CA fitness evaluations; slowing down the process dramatically as this is the primary computational cost. Too small a population would simple not move quickly enough through the landscape, again causing slow performance.

[7] This is an extraordinarily high mutation rate and is discussed within the conclusions of section 6

To allow for a greater range of mutations (and therefore introducing variability into the width of meanders) the mutation rate was made stochastic within certain limits. This meant that some children would mutate with a low rate of 1-2%, whilst other children would mutate with a high rate of 20-30%.

Measuring Fitness

Calculating the fitness of a ruleset has been outlined earlier. A fit ruleset maintains a population rate disproportionately lower than λ whilst maintaining an high, noisy variance in SA rate.

A function (runCA) has been implemented which steps through the CA state-space and returns a vector of SA and P values for each time-step. From this:-

Fitness of Mean is measured by splitting each vector in half and comparing the mean of each half with some *desired value* (this ensures that a stable mean is maintained – punishing linear increases or decreases).

Fitness of Variance is simply the variance of the vector. However, an additional constraint is that the change-distribution should be roughly Gaussian[8] to ensure that this variance is "noisy".

Finally, **λ** can also be calculated and compared to a *desired value*.

These fitness measurements and comparisons are then added together to form the overall score for the ruleset.

By using "desired values", it becomes possible to further examine the CA landscape. Setting desired values such that $P \approx 0.1$, $SA \approx 0.65$ and $\lambda \approx 0.2734$ will look for rulesets with similar rates to the Game of Life. However, it is equally possible to look complexity elsewhere.

Other Parameter Considerations

Finally, there are a few more parameters which need to be taken into consideration.

The initial state-space; It was decided that for the purpose of these experiments that a random initial state-space with a population rate close to the desired population rate would be used.

[8] See conclusions in section 6

The size of the CA lattice; with larger lattices, the CA state-space is open to display a much wider collection of behaviours. However, larger lattices take a lot longer to evaluate. For these experiments a CA of 50x50 cells was used.

CA Run Time; longer runs will highlight behaviours far more clearly, however there is a huge computational overhead. For these experiments each CA was run for 100 time steps.

5 Results

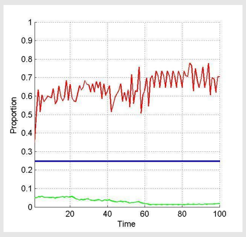
Using the described program the CA rulespace was searched, and the final rulesets examined. This section aims to highlight just a few of the rulesets found, their behaviours and their corresponding SA and P rates.

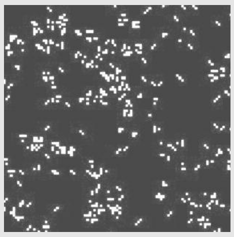
Example 1: SA = 0.65 : P = 0.05 : λ= 0.27

Very close to Game of Life

Similar SA, P and λ values to Game of Life.

Includes gliders, fixed points and chaotic explosions.



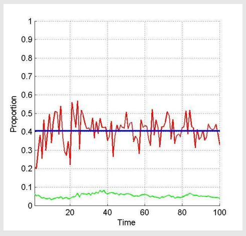


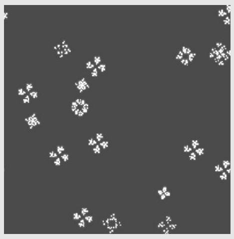
Example 2: SA = 0.40 : P = 0.05 : λ= 0.40

Glowing Snowflakes

SA ≈ λ

Regions of chaos quickly disappear to leave lovely looking long-period (about 35 state) cyclical attractors.



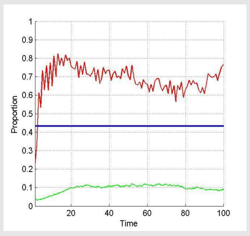


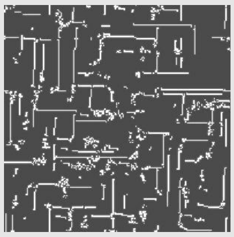
Example 3: SA = 0.70 : P = 0.10 : λ= 0.43

Lines and gliders

Lots of growing straight lines that, when hit with regions chaos, burn like a fuse.

Plus high rate of gliders.

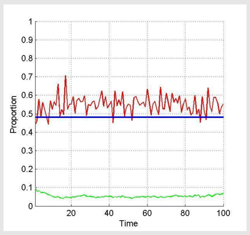


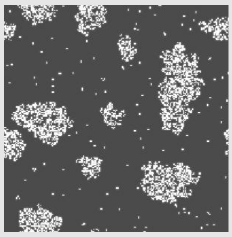


Example 4: SA = 0.55 : P = 0.08 : λ= 0.50

Big blobs

Larges areas of throbbing chaotic mass which show signs of stability (they do not decay to full order or chaos)

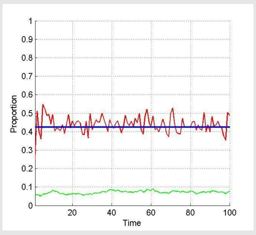


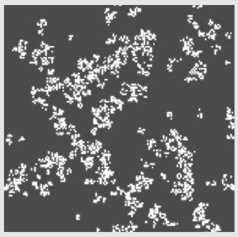


Example 5: SA = 0.42 : P = 0.08 : λ= 0.42

Big blobs and gliders

Large areas of chaotic mass in an otherwise empty environment except for a number of gliders flying about.



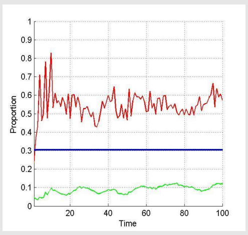


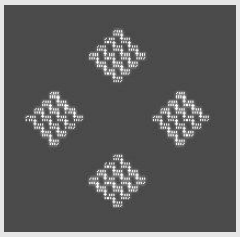
Example 6: SA = 0.55 : P = 0.10 : λ= 0.30

Minimal Replicator

Fractular

Given a start of 2 blocks, the CA grows endlessly in a fractular pattern.



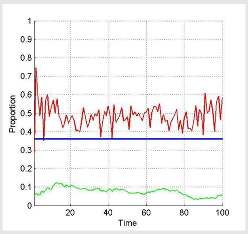


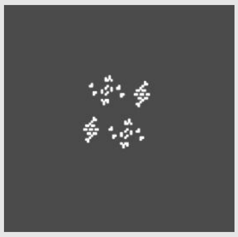
Example 7: SA = 0.48 : P = 0.08 : λ= 0.37

Minimal Replicator

Complex Lifecycle

Minimal replicator with a exceptionally long but finite, non-repeating lifecycle resulting in a couple of blinkers.



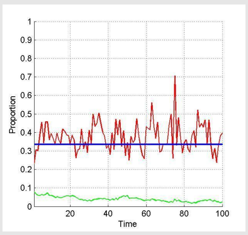


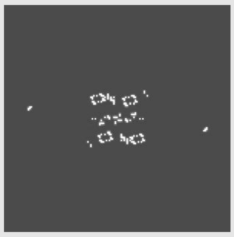
Example 8: SA = 0.35 : P = 0.05 : λ= 0.32

Minimal Replicator

Glider gun

Minimal replicator with a finite lifecycle. During its lifecycle one single glider is fired in each direction.



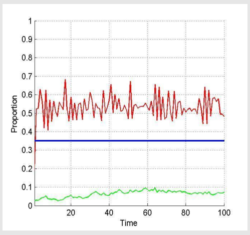


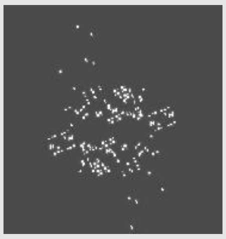
Example 9: SA = 0.52 : P = 0.08 : λ= 0.35

Minimal Replicator

Clouds of Gliders

A minimal replicator which after a while starts emitting gliders in all directions. Gliders can react to create new replicators.





There are many, many more examples that could have been listed here. Each run of the programme produced a variety of interesting and complex behaviours; including all sorts of gliders, cyclical objects of varying shape and size, giant flowing blobs of chaos and minimal replicators.

6 CONCLUSIONS

This paper aimed to highlight and describe a new method of finding complex rulesets within the CA rulespace. Section 3 pointed to the significant difference of "stay alive" and "population" rates in the Game of Life compared with other, less complex, rulesets. Section 4 described a GA which could search the CA landscape for other rulesets with the same "footprint". Section 5 outlined a handful of the rulesets found and describes the behaviour of each. The final question, therefore, is whether or not the evolved rulesets are "*complex*", and so whether the main aim of this paper has been met.

On Complexity

Unfortunately this question doesn't have so simple an answer. As with so much in the field of A-Life and AI, the term lacks precise definition and therefore becomes somewhat open to interpretation.

Using Wolframs classes, behaviour which is neither ordered (fixed/cyclical) or chaotic (unpredictable and hence without pattern or form) is defined as "complex" or "interesting".

More recently however, with a shift in focus towards the commercial use of CAs, CA researchers have begun to redefine complexity in terms of the emergent computational capacity of a system.

Further still, Ninagawa attempts to formally restate complexity as a dynamical system which displays 1/f fluctuations in a changing state-space.

The results shown certainly display behaviours which fit within Wolfram's criteria of Class IV CAs, and many also show promising signs of emergent computation capabilities[9]. On this basis, the author feels able to conclude that "complex" rulesets have indeed been found according to these definitions. Other definitions, however, would require further investigation.

Further Investigations

There are a number of assumptions, parameters and choices within this paper which would greatly benefit from further investigation and discussion.

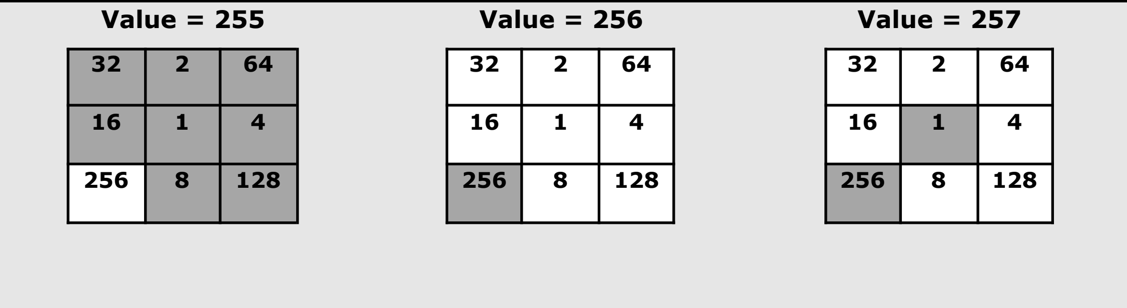
Firstly, the mutational GA developed certainly served its primary purpose of finding complex rulesets fitting the highlighted "significant factors". However, its initial aim of slowly mutating rulesets towards greater and greater complexity was not met. The most

[9] Glider collisions have been briefly examined although not outlined within the body of this paper.

successful mutation rate tended to be an extremely high 10-20% - making this a rather crude example of a GA.

A detailed examination of alternative techniques may result it a greater understanding of the CA landscape. For example, a more formal neutral network algorithm may assist in achieving the initial "step by step" goal and so enable the mapping of the CA landscape.

The genotype encoding itself has a crucial feature which may affect the style of GA that can be used successfully. Figure 6 highlights the ruggedness in state transition between three consecutive rules. A smoother encoding which slowly changes the "alive-ness" of states may allow recombination to play a greater role.



(Figure 6) 3 numerically consecutive, but vastly different, states

The primary questions posed by this report however, should pertain to the significance of the population and SA rates in determining complexity.

The measurement of Gaussian distribution in SA rate variance, for example, was primarily implemented as a method of drowning out unwanted ordered variance (regular waves) but it is felt to have a more important significance that that. It may have similar implications of "multi-scale" variance similar to Ninagawa's 1/f noise.

Similarly, the population is shown to closely follow λ in chaotic systems and an increase in order appears to have the effect of pulling it away from this norm. Is this relationship proportional? How does it relate to other measures of complexity, such as entropy variance? These questions certainly warrant a deeper investigation and work has already been begun to fully understand these relationships.

Finally, it is exceedingly important to examine the portability of these observations to larger state or larger dimension CAs, and also to other discrete dynamical systems such as Random Boolean Networks. For these factors to be established as proper measures of complexity, they must be demonstrated to have a wider scope than the 2-state, 2D CAs looked at within this report.

Propositions

During this process of "Looking for Life", a large number of complex worlds have been peered into and observed. In the vastness of computational space there are many forms of glider, an exquisite array of replicators and some beautiful displays of cyclical attraction.

A lot of these worlds, including the Game of Life, provide temporary displays of complex and interesting behaviour; but they eventually run out of energy and decay to simpler worlds of ordered structure.

In others, the decay runs the other way; with chaotic fires burning away ordered structures until the system is left in a state of high energy and chaos.

However, a rare few kept energetic decay at bay. In these worlds, emergent complex behaviours were able to persist.

Maybe complexity is simply the measure of a systems capacity for maintaining itself far from the states of chaos and order; a state of energetic non-equilibrium kept alive through replication, translation and form. To this extent, maybe complexity and life have more in common than was initially considered.

7 REFERENCES

[1] **Ganguly N., et al.** (2003) "*A Survey on Cellular Automata*", Technical Report Centre for High Performance Computing, Dresden University of Technology

[2] **Ninagawa S.** (1998) "*1/f fluctuation in the Game of Life*", Physica D vol. 118, pp. 49-52

[3] **Ninagawa S.** (2005) "*Evolving Cellular Automata by 1/f Noise*", proceedings of ECAL 2005

[4] **Wuensche A.** (1996) "*Attractor Basins of Discrete Networks*", D.Phil, University of Sussex

[5] **Mitchell M., Crutchfield J.P., Das R.** (1997) "*Evolving Cellular Automata to Perform Computations*"

[6] **Crutchfield J.P., Mitchell M.** (1995) "*The Evolution of Emergent Computation*", Proceedings of the National Academy of Sciences, USA 92:23 10742-10746.

[7] **Mitchell M., Hraber P.T., Crutchfield J.P.** (1993) "*Revisiting the Edge of Chaos: Evolving Cellular Automata to Perform Computations*", Complex Systems 7, pp. 89-130.

[8] **Packard N. H**. (1988), "*Adaptation toward the edge of chaos*", Dynamic Patterns in Complex Systems, pp. 293-301.

[9] **Langton C.G.** (1990) "*Computation at the edge of chaos: Phase transitions and emergent computation*", Physica D vol. 42, pp. 12-37.

[10] **Park K.**, (webpage) "*Stochastic Cellular Automata*"
http://www.cs.purdue.edu/homes/park/interest-ca.html

[11] **Hanson J.E., Cruthfield J.P.** (1994) "*The attractor-basin portrait of a cellular automaton*".

[12] **Wolfram S.** (1984) "*Universality and complexity in cellular automata*", Physica D vol. 10

[13] **Seeley T.D.** (1991) "*Collective decision making in honey bees: how colonies choose among nectar sources*" Behav. Ecol. Sociobiol. vol 28 pp 277-290.

[14] **Britton N.F., Pratt S.C., Franks N.R., Seeley T.D.** (2002) "*Deciding on a new home: how do honey-bees agree?*" Proceedings of the Royal Society of London B 269:1383-1388.

[15] **Johnson S.** (2001) "*Emergence*", Penguin Books, p 32-33

[16] **James S.** (2004) "*Evolutionary CA and the Edge of Chaos*", Unpublished, University of Sussex MSc Paper

8 Appendix A –MATLab Code

The following is the final programme code for the GA developed to search for complex systems. It was written and run within MATLab version 7.0.1.15

```
function[] = LL(filename)
   %-------------------------------------------------
   % GLOBAL SETUP
   %-------------------------------------------------


   global h_figure1; global h_figure2;
   global fld; global staterot; global dieifalone; global stochastic;
   global xx; global xplus; global xminus; global gridcount;

   h_figure1 = figure(1); h_figure2 = figure(2);

   set(h_figure1,'Name','SA and Population Rates','WindowStyle','Docked')
   set(h_figure2,'Name','CA Animation','WindowStyle','Docked')

   fld = 'C:\Looking for Life\';



   %-------------------------------------------------
   % Set Up Constants
   %-------------------------------------------------
   % --- CA SETUP ---
   gridsize        = 50;                  % --- how many cols/rows should the CA have
   maxt            = 100;                 % --- the number of timesteps
   symmetry        = 1;                   % --- 1 = include symmetry ; 0 = exclude
   dieifalone      = 1;                   % --- 1 = always result in death for a single cell

   % --- GA SETUP---
   gentime         = 100;                 % --- how many generations in GA
   popsize         = 60;                  % --- how big a population in GA
   GAkeepalives    = 10;                  % --- how many parents to keep unmutated
   muteprob        = 0.4;                 % --- maximum mutation rate

   % --- LAMBDA ---
   initlambda      = 0.27;                % --- start population with roughly this Lambda
   targetlambda    = 0.5;                 % --- the DESIRED VALUE for lambda

   % --- SA AND POPULATION ---
   targetsa        = 0.5;                 % --- the DESIRED VALUE for SA
   initpop         = 0.1;                 % --- the initial population density
   targetpop       = 0.1;                 % --- the DESIRED VALUE for P

   % --- General ---
   nbhood          = 9;                   % --- the neighbourhood size of the CA
   nostates        = 2;                   % --- the state size of the CA
   rulesize        = (nostates^nbhood);   % --- the number of rules in the CA ruletable
   bestscore       = -1000000;            % --- initialising the best GA score
   bestscorecount  = 1;                   % --- a counter used for counting new BEST SCORES
   startpop        = 1;                   % --- counter used within code
   xx              = 1:gridsize;          % --- counter used – array of X cell positions
   xplus           = [2:gridsize,1];      % --- counter used – array of X + 1 cell positions
   xminus          = [gridsize,1:gridsize-1]; % --- counter used – array of X – 1 cell positions
   gridcount       = gridsize^2;          % --- counter used – number of cells in the lattice
   muterate        = zeros(1,popsize);    % --- counter used within code

   [gauss gaussix] = hist(randn(1, 10000),21);     % --- array storing a GAUSSIAN set of data
   %-------------------------------------------------
```

```
%-------------------------------------------------
%-------------------------------------------------
% Set Up Initial Rules
%-------------------------------------------------
%-------------------------------------------------

% Set Up Rotation Matrix
for i = 1:1:rulesize
    [y1 y2 y3] = staterotate(i);
    staterot(i, 1:3) = [y1 y2 y3];
end

if (symmetry == 0)
         if (initlambda == 0)
            % ------------------------
            % --- GAME OF LIFE
            % ------------------------
            rulesets        = 0:(rulesize-1);
            alive           = bitand(rulesets,1);
            nbours          = bitcount(rulesets,nbhood)-alive;
            rules(1,:)      = (((nbours<4) .* (nbours>1) .* alive) + (nbours==3))>=1;
        else
            % ------------------------
            % --- BASED ON LAMBDA
            % ------------------------
            rules = rand(popsize,rulesize)<initlambda;
        end
else
    % ------------------------
    % --- SYMMETRICAL
    % ------------------------
    rules = zeros(popsize,rulesize);
    for i = 1:1:popsize
       for j = 1:1:ceil((rulesize*initlambda)/4)
            tmprule = ceil(rand() * (rulesize-1));
            while(rules(i,tmprule) == 1) tmprule = ceil(rand() * (rulesize-1)); end
            rules(i,tmprule) = 1;
            rules(i,staterot(tmprule,1)) = 1;
            rules(i,staterot(tmprule,2)) = 1;
            rules(i,staterot(tmprule,3)) = 1;
       end
    end
end
% ------------------------
% --- DIE IF ALONE
% ------------------------
if (dieifalone == 1) rules(:,2) = 0; end

% ------------------------
% --- LOAD FROM FILE
% ------------------------
if (strcmp(filename,'') == 0)
    gentime = 1; popsize = 1; symmetry = 0; dieifalone = 0; GAkeepalives = 0;
    load(cat(2,fld,filename));
    rules = bestrules;
end

%----------------------------------------------
```

```
%--------------------------------------------------
% RUN NEUTRAL NETWORK
%--------------------------------------------------
for generation = 1:1:gentime

   %---------------------------
   % CALCULATE GENERATION SCORES
   %---------------------------
   for i = startpop:1:popsize
      % --- INITIALISE CA GRID ---
      initcells = ones(gridsize,gridsize);
      while (abs(sum(sum(initcells)/gridcount) - initpop) > 0.005)
         initcells      = (rand(gridsize,gridsize))<initpop;
      end

      % --- CALCULATE LAMBDA ---
      if (stochastic == 0)
         lambda = sum(rules(i,:) ==1)/rulesize;
       end

      % --- RUN CA ---
      [finalcells noIters, stats] = ...
                 runCA(initcells,rules(i,:),gridsize,maxt,nbhood,0,0,nondeterminism);

      % --- CALCULATE SCORES ---
      fullp = stats(3,:);
      fullsa = stats(1,:);

      p = fullp(ceil(noIters/5):noIters-1);
      sa = fullsa(ceil(noIters/5):noIters-1);

      meanp = mean(p);
      meansa = mean(sa);

      meanp1 = mean(fullp(1:ceil(noIters/2)));
      meansa1 = mean(fullsa(1:ceil(noIters/2)));

      meanp2 = mean(fullp(ceil(noIters/2):noIters-1));
      meansa2 = mean(fullsa(ceil(noIters/2):noIters-1));

      dista = 0; distb = 0; distc = 0; distd = 0; diste = 0; distf = 0;
      varsa = 0; varsa2 = 0;

      % Check for gaussian variance
      distrsa = sa(1:1:size(sa,2)-1) - sa(2:1:size(sa,2));
      varsa2 = (((max(distrsa) - min(distrsa)) * 10) - 10);
      distrsa = distrsa .* ((max(gaussix)-min(gaussix)) / (max(distrsa) - min(distrsa)));
      gausssa = hist(distrsa,21);
      gausssa = gausssa .* (sum(gauss)/sum(gausssa));
      varsa = - ((sum(abs(gauss - gausssa))/4000));
      if (varsa < -2) || isnan(varsa)
         dista = -50;
      else
         dista = varsa2;
      end

      if (targetlambda > 0)
         distb = 0 - abs(lambda-targetlambda)*25;
      end

      if (targetsa > 0)
         distc = distc - abs(meansa2-targetsa)*5;
         distc = distc - abs(meansa1-targetsa)*5;
      end



      if (targetpop > 0)
         diste = diste - abs(meanp2-targetpop)*25;
```

```
            diste = diste - abs(meanp1-targetpop)*25;
            diste = diste - abs(meanp2-meanp1)*50;
        end

        if (max(sa) > 0.99) distf = -50; end
        if (min(sa) < 0.01) distf = -50; end
        if (min(p) < 0.01) distf = -50; end

        genscores(i) = dista + distb + distc + distd + diste + distf;

        if isnan(genscores(i))
            genscores(i) = -100;
        end

        % --- BEST SCORE TEST & GRAPHING ---
        if (genscores(i) > bestscore)
            bestscore = genscores(i);
            bestrules = rules(i,:);

            figure(h_figure1); hold off; clf; grid on; axis xy; hold on;
            xlabel('Time');
            ylabel('Proportion');
            axis([1 noIters 0 1]);
            plot(1:1:noIters,fullp,'g-','LineWidth',1);
            plot(1:1:noIters,fullsa,'r-','LineWidth',1);
            plot(1:1:noIters,fulld,'c-','LineWidth',1);
            plot(1:1:noIters,fullb,'m-','LineWidth',1);
            line([0 noIters],[lambda lambda],'Color','b','LineWidth',2);

            figure(h_figure2); hold off; clf; grid on; axis xy; hold on;
            hist(distrsa,21);

            drawnow;
            save(cat(2,fld,num2str(bestscorecount),' - RULES.mat'),'bestrules');
            saveas(h_figure1,cat(2,fld,num2str(bestscorecount),' - SA POP.jpg'))

            bestscorecount = bestscorecount + 1;
        end
        % ------------------------------------------

    end
    % --- END CALCULATE POPULATION SCORES ---


    %---------------------------
    % NEUTRALITY WANDERING
    %---------------------------
    %startpop = GAkeepalives;
    [genscore ix] = sort(genscores,'descend');
    for a = 1:1:popsize
        ruleid = ix(a);
        % --- KEEP GOOD PARENTS ---
        if a <= GAkeepalives
            newrules(a,:) = rules(ruleid,:);
        % --- CREATE NEW CHILDREN ---
        else
            %parentrule = ceil(rand*GAkeepalives);
            parentrule = mod(a,GAkeepalives) + 1;
            newrules(a,:) = newrules(parentrule,:);
            if (symmetry == 0)
                    % --- NORMAL ---
                    for j=1:1:rulesize
                        if(rand()<muteprob) newrules(a,j) = abs(1-newrules(a,j)); end
                    end
            else
                % --- SYMMETRICAL ---
                if (muteprob >= 1)
                    noofmutations = muteprob;
                else
                    muterate(1,a) = rand * muteprob;
                    noofmutations = ceil((rulesize*muterate(1,a))/4);
                end
                for j = 1:1:noofmutations
                    tmprule = ceil(rand() * (rulesize-1));
                    newrules(a,tmprule) = abs(1-newrules(a,tmprule));
                    newrules(a,staterot(tmprule,1)) = newrules(a,tmprule);
                    newrules(a,staterot(tmprule,2)) = newrules(a,tmprule);
                    newrules(a,staterot(tmprule,3)) = newrules(a,tmprule);
                end
            end
        end
    end
    rules = newrules;
    % --- DIE IF ALONE ---
    if (dieifalone == 1) rules(:,2) = 0; end
    % --- END NEUTRALITY WANDERING ---

    output = [generation mean(genscores) max(genscores)]
end
% --- END NEUTRAL NETWORK ---
%--------------------------------------------------


%--------------------------------------------------
% ALLOW USER TO RUN FINAL RESULT
%--------------------------------------------------
g=input('Enter 1 to run (or 2 to save video) and press enter\n');
while (g >= 1)
    gridsize       = 100;
    maxt            = 155;
    drawSpeed       = 0.008;

    g1 = 0; g1=input('Enter gridsize\n');
    if (g1 >= 1) gridsize = g1; end

    g1 = 0; g1=input('Enter maxt\n');
    if (g1 >= 1)  maxt = g1; end

    xx              = 1:gridsize;
    xplus           = [2:gridsize,1];
    xminus          = [gridsize,1:gridsize-1];
    gridcount       = gridsize^2;

    initcells = ones(gridsize,gridsize);
    while (abs(sum(sum(initcells)/gridcount) - initpop) > 0.005)
        initcells       = (rand(gridsize,gridsize))<initpop;
    end

    [finalcells noIters, stats] =
runCA(initcells,bestrules,gridsize,maxt,nbhood,g,drawSpeed,nondeterminism);

    g=input('Enter 1 to run (or 2 to save video) and press enter\n');
end
%--------------------------------------------------
end
```

```
%---------------
% runCA function
%
% Given an inital state, a set of rules and details
% of the CA, run a the CA for maxt iterations.
%---------------
function[cells noOfIters, stats]     =
runCA(cells,rules,gridsize,maxt,nbhood,drawIt,drawSpeed,nondeterminism)
    global h_figure2;
    global fld;
    global xx; global xplus; global xminus; global gridcount;

    warning off all
    noOfIters       = 1;
    noAs            = sum(sum(cells));
    stats           = zeros(5,maxt);
    noBs = 0; noDs = 0; noSAs = 0; S = 0;

    if (drawIt >= 1)
        if(drawIt == 2) mov = …
             avifile(cat(2,fld,'output.avi'),'fps',15,'quality',100,'compression','CinePak'); end;
        emptygrid       = zeros(gridsize,gridsize);
        IMH             = image(cat(3,emptygrid,cells,emptygrid));
        axis equal;
        axis tight;
        pause(drawSpeed);
    end

     % --- Perform STATE Changes ---
    while ((noOfIters <= maxt))
        oldcells = cells;
        oldAs    = noAs;
        states(xx,xx) = 1 + cells(xx,xx) + ...
                      (2 * cells(xplus,xx)) + ...
                      (4 * cells(xx,xplus)) + ...
                      (8 * cells(xminus,xx)) + ...
                      (16 * cells(xx,xminus)) + ...
                      (32 * cells(xplus,xminus)) + ...
                      (64 * cells(xplus,xplus)) + ...
                      (128 * cells(xminus,xplus)) + ...
                      (256 * cells(xminus,xminus));
       cells(xx,xx)      = rules(states(xx,xx));


        if(drawIt >= 1)
            set(IMH, 'cdata', cat(3,emptygrid,cells,emptygrid) );
            if(drawIt == 2) F = getframe(gcf); mov = addframe(mov,F); end;
            drawnow
            pause(drawSpeed);
        end

        noAs    = sum(sum(cells));
        noSAs   = sum(sum(oldcells&cells)) / oldAs;
        stats(3,noOfIters)  = (noAs/(gridsize*gridsize));
        stats(1,noOfIters) = noSAs;
        noOfIters           = noOfIters + 1;
    end

    if(drawIt>=1)
        set(IMH, 'cdata', cat(3,emptygrid,cells,emptygrid) );
        if(drawIt == 2) F = getframe(gcf); mov = addframe(mov,F); end;
        drawnow; pause(drawSpeed);
        if(drawIt == 2) mov = close(mov); end;
    end
    noOfIters       = noOfIters - 1;
end
```

```
%-------------------------------------
% STATE ROTATE Function
%
% Given a STATE, produce 3 rotations
%-------------------------------------
function[y1 y2 y3] = staterotate(x)
   y = x - 1;
   y1 =    bitand(y,1)/1 * 1 + ...
         bitand(y,2)/2 * 4 + ...
         bitand(y,4)/4 * 8 + ...
         bitand(y,8)/8 * 16 + ...
         bitand(y,16)/16 * 2 + ...
         bitand(y,32)/32 * 64 + ...
         bitand(y,64)/64 * 128 + ...
         bitand(y,128)/128 * 256 + ...
         bitand(y,256)/256 * 32;
   y2 =    bitand(y1,1)/1 * 1 + ...
         bitand(y1,2)/2 * 4 + ...
         bitand(y1,4)/4 * 8 + ...
         bitand(y1,8)/8 * 16 + ...
         bitand(y1,16)/16 * 2 + ...
         bitand(y1,32)/32 * 64 + ...
         bitand(y1,64)/64 * 128 + ...
         bitand(y1,128)/128 * 256 + ...
         bitand(y1,256)/256 * 32;
   y3 =    bitand(y2,1)/1 * 1 + ...
         bitand(y2,2)/2 * 4 + ...
         bitand(y2,4)/4 * 8 + ...
         bitand(y2,8)/8 * 16 + ...
         bitand(y2,16)/16 * 2 + ...
         bitand(y2,32)/32 * 64 + ...
         bitand(y2,64)/64 * 128 + ...
         bitand(y2,128)/128 * 256 + ...
         bitand(y2,256)/256 * 32;
   y1 = y1 + 1;
   y2 = y2 + 1;
   y3 = y3 + 1;
end


%-------------------------------------
% BIT COUNT Function
%
% Given a number, how many bits are on?
%-------------------------------------
function[y] = bitcount(x,size)
   y = bitand(x,1)/1 + ...
      bitand(x,2)/2 + ...
      bitand(x,4)/4 + ...
      bitand(x,8)/8 + ...
      bitand(x,16)/16;
   if (size == 9)
      y = y + ...
         bitand(x,32)/32 + ...
         bitand(x,64)/64 + ...
         bitand(x,128)/128 + ...
         bitand(x,256)/256 + ...
         bitand(x,512)/512;
   end
end
```

```
%-------------------------------------
% STATE FLIP V
%
% Given a state, flip it vertically
%-------------------------------------
function[y] = stateflipv(x)
   x = x - 1;
   y = bitand(x,1)/1 * 1 + ...
      bitand(x,2)/2 * 2 + ...
      bitand(x,4)/4 * 16 + ...
      bitand(x,8)/8 * 8 + ...
      bitand(x,16)/16 * 4 + ...
      bitand(x,32)/32 * 64 + ...
      bitand(x,64)/64 * 32 + ...
      bitand(x,128)/128 * 256 + ...
      bitand(x,256)/256 * 128;
   y = y + 1;
end


%-------------------------------------
% STATE FLIP H
%
% Given a state, flip it horizontally
%-------------------------------------
function[y] = statefliph(x)
   x = x - 1;
   y = bitand(x,1)/1 * 1 + ...
      bitand(x,2)/2 * 8 + ...
      bitand(x,4)/4 * 4 + ...
      bitand(x,8)/8 * 2 + ...
      bitand(x,16)/16 * 4 + ...
      bitand(x,32)/32 * 256 + ...
      bitand(x,64)/64 * 128 + ...
      bitand(x,128)/128 * 64 + ...
      bitand(x,256)/256 * 256;
   y = y + 1;
end
```