Variations on Genetic Cellular Automata

Alice Durand
David Olson

Physics Department

amdurand@ucdavis.edu
daolson@ucdavis.edu

Abstract:

We investigated the properties of cellular automata with three or more possible states, represented by colors. For the simulation, the Pygame tool was used. Our hypothesis was that the system would be overall chaotic, with periods of stability, but we found that the system instead converges to a stable period-2 limit cycle, indicated by a checkerboard pattern in the simulation. Mutation can have significant effects on the traits that emerge dominant.

Introduction

Our project focuses on a variation of the cellular automaton, using three or more states instead of the regular two. Each of the states is represented by a "gene pair" with any combination of a red cell, a green cell, or a blue cell, and obeys a simple set of rules for producing the next "generation" of cells. Our motivation for this project is that it is relatively simple to simulate, but still opens up numerous possibilities for chaotic behavior. Even modeling a few short iterations by hand (i.e. a random arrangement of six colored cells) gives some very intriguing results – we were interested to see how the system would behave with many cells!

This project could also provide insight on how genes are expressed in animals or humans – for example, eye color (blue, green, and brown), or hair color (blonde, black, and red). We decided to use pairs of genes, as that is how the pairs of chromosomes exchange information in cellular reproduction. Naturally, a truly accurate simulation for such things would be exceedingly complex, and our project can only hope to scratch the surface of long-term gene expression.

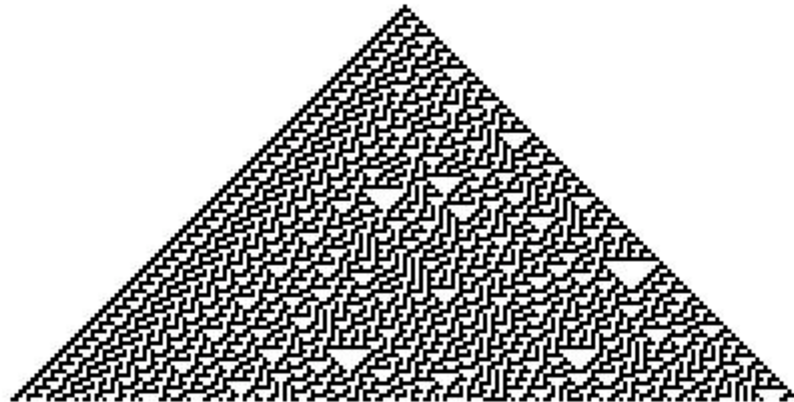
Our model for the many-state cellular automaton consists entirely of Python code and uses the tool Pygame for the visual simulations. The interface and code are very straightforward – the user gives an initial amount of cells to start with and the amount of generations to produce (the "step" size), while the program puts in a random combination of color pairs for the initial condition. We then use an algorithm that implements the reproduction rules and proceeds to display the next generation of colored cells. The simulation can continue on for quite a while, allowing the viewer to see if any patterns or colors dominate over others.

When writing the simulation for our project, we decided to expand on the original three colors, and introduced a "blending" algorithm that uses the hexadecimal codes of the colors (red, green, and blue having the simplest number representation) to "blend" together our initial set of colors. Besides creating an attractive, computer-background worthy visual, this gave us some unexpected results – namely that the colors often converge to one of the primary red, green, or blue colors after a relatively short time! We later on realized that this particular result was erroneous, and upon fixing the code found that the population would instead converge to a stable period-2 limit cycle (barring any random mutations). Another modification we introduced was the potential for cell "mutation" – that is, one cell randomly becoming a different color in a random number of generations. This is closely related to the gene mutations we see in DNA.

We found that through the simulation, the population eventually stabilizes to a period-2 limit cycle (indicated by a checkerboard pattern with two colors), and that mutation can strongly affect the outcome.

Background and Short History

The basic rules governing most cellular automata are such: a cell may only interact with its nearest neighbors (the ones on the left and right for a 1-D automaton, and those on the left, right, top, and bottom for a 3-D automaton), each iteration is discrete – for each "tick of the clock," a new generation is produced, and the particular reproduction rules for each automaton do not change over the course of one simulation. Many of the cellular automata studied so far involve only two possible states for a cell (likely related to the fact that most computers use binary), and simulations produce patterns that are largely chaotic, but do exhibit temporary stabilities. For example,



Here, the cellular automata starts from one cell and grows geometrically according to a particular rule.

The concept of cellular automata has been around since the 1950's, when several scientists began to study them for various applications in real life. Stanislaw Ulam was interested in cellular automatons because they were related to his current research on crystal growth in lattice networks. Indeed, the Ising model (which uses a two-dimensional square lattice) is connected to cellular automata, the main difference being that the rules governing the former are inherently random. Nonetheless, Ulam eventually met up with John von Neumann, who was working on a simulation for biological self-reproduction. In the words of another cellular automaton scientist (who apparently also collaborated with the great Professor Crutchfield on a paper!):

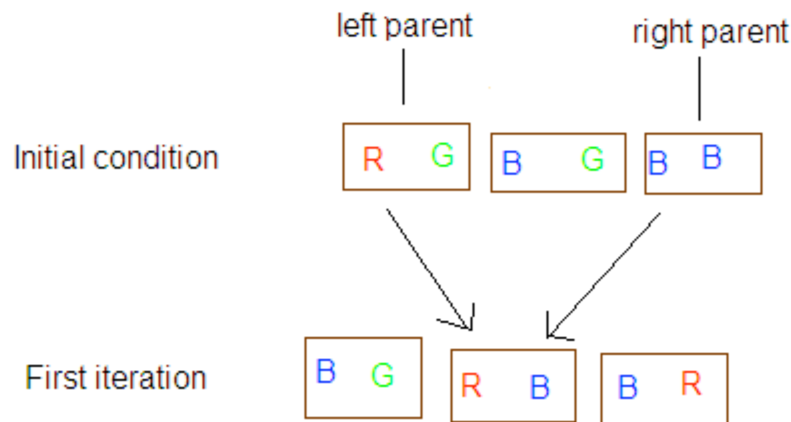
"CA were not invented, however, to be realistic models of Nature. They started with John von Neumann, who wanted to study self-reproduction, and decided that the first thing to do was

ignore everything biologists had learned about the way actually existing organisms reproduce themselves. This is known as *hubris,* and is especially galling when it works." – *Cosma Shalizi*

Despite John von Neumann's apparent ignorance of all things biological, at Ulam's prompting he nevertheless came up with a rather complicated version of a self-reproducing mathematical/biological system involving 29 different possible colors for a 200,000 cell lattice; this was the first cellular automaton. John Conway expanded on the idea in the 1970's with his famous "Game of Life," a 2-D computer simulation in which a cell lives or dies depending on the states of its four neighbors – today, this type of simulation is often used as a practice program in beginning programming classes. Later on, in the 1980's, Stephen Wolfram wrote numerous papers on "elementary cellular automata," a very simple, yet surprisingly complicated family of cellular automata. The graphic above is an example of one of the resulting systems.

Dynamical System

The particular system we have selected involves six possible states, or combinations of red, green and blue (a far cry from von Neumann's 29!). For the sake of simplicity, we have distinguished them with the colors red, green, and blue; these colors are randomly arranged in a finite line of cells. The main rule governing the new generation (before the blending algorithm) is perhaps best explained by a picture:



We have a simple, 3-cell initial condition with some random colors. In each iteration, a cell randomly takes one of the colors from each of its neighbors – the color from the left parent cell goes into the first spot of the new cell, and the one from the right parent cell goes into the second spot. Naturally, this system wraps around (old-school video game style).

Upon inputting the blending algorithm, in which the different-colored neighbors would blend to make a new color, we see that the system is at least somewhat biologically realistic. As with flower color, for example, two plants with the same color flowers will produce an offspring of that color,

whereas two different-colored plants will produce a plant with a blend of the two hues. Of course, we are not taking into account dominant and recessive traits (a feat for a more sophisticated algorithm).

The equations of motion for this system are relatively simple. For a system S_t^i *of n cells, we have the initial condition:*

$$S_t^i = [X^0, Y^0]_t, \ldots, [X^{i-2}, Y^{i-2}]_t, [X^{i-1}, Y^{i-1}]_t, [X^i, Y^i]_t, [X^{i+1}, Y^{i+1}]_t, [X^{i+2}, Y^{i+2}]_t, \ldots, [X^n, Y^n]_t$$

After one time step, we get:

$$S_{t+1}^i = [G^n, G^1]_{t+1}, \ldots, [G^{i-3}, G^{i-1}]_{t+1}, [G^{i-2}, G^i]_{t+1}, [G^{i-1}, G^{i+1}]_{t+1}, [G^i, G^{i+2}]_{t+1}, [G^{i+1}, G^{i+3}]_{t+1}, \ldots, [G^{n-1}, G^0]_{t+1}$$

Here we are using Gs to denote a random choosing of X or Y from the left and right parent cells. In technical terms:

$[G^n, G^1]_{t+1}$ = *any of four combinations of Xs and Ys from the cells* $[X^1, Y^1]_t$ *and* $[X^n, Y^n]_t$

In particular, for the blending algorithm, we obtain the next time step by performing an averaging operation with the hexadecimal digits:

$$(G^{i-1} + G^{i+1})_{t+1} / 2$$

Methods

Our program takes in a population size, a step size, a seed number (usually set to 0), and a mutation coefficient from the user. The result is a map of however many iterations the user wanted – from this we can see patterns and chaotic behavior. The program has two parts: one is the class file which details an object that holds the states for every cell in the population, iterates the initial condition, and blends the genes together; the other is the program file, which calls the class file, and updates the screen, but consists mainly of the user interface code (using Tk).

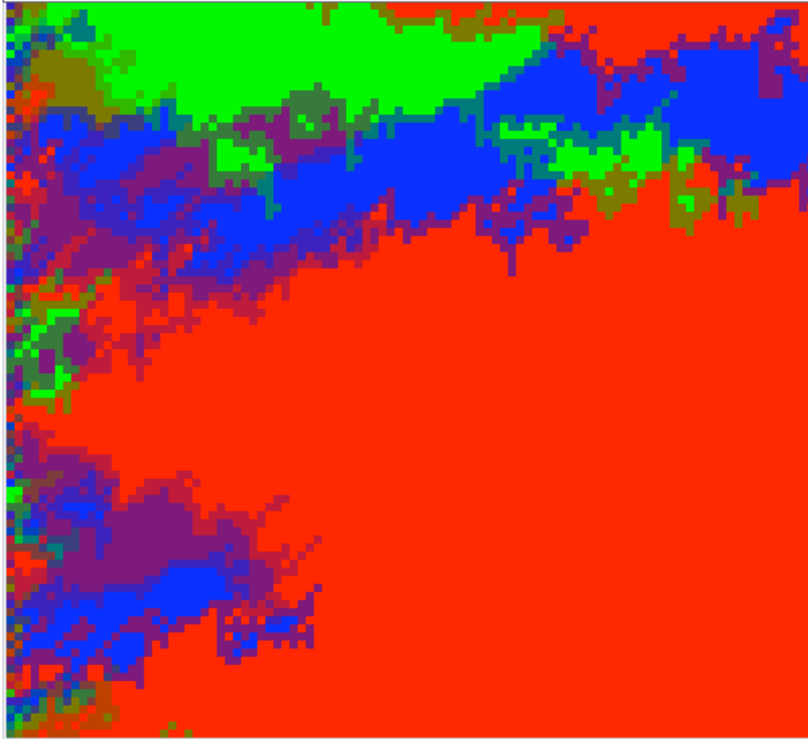
The program file passes in a "gene map," which is a dictionary file where 'r', 'g', 'b', and so on (other colors such as cyan, magenta, and yellow may be used) are the keys passed in. Each color has an associated array with its hex values (e.g. 'r' is connected to the array "array([256,0,0])"). The algorithm itself uses only the keys 'r', 'g', and 'b' in the iterations, and the values are only converted into hex when displaying the resulting generation. The population is generated randomly via either user input (using the input seed), or a random seed as determined by the computer's clock. A basic

chain of events following user commands would look like this: the program looks at the initial state → iterates the state → converts the results into hexadecimal color values → stores the information in a "state matrix" that holds all of the information to pass to the screen. The user can also input a mutation coefficient from 0 to 1. This represents the probability of a mutation happening in one of the genes and is usually on the order of 10^{-4} to 10^{-6} for an actual eukaryotic cell. Note that here, we are mutating entire gene pairs, as opposed to single genes.

Results

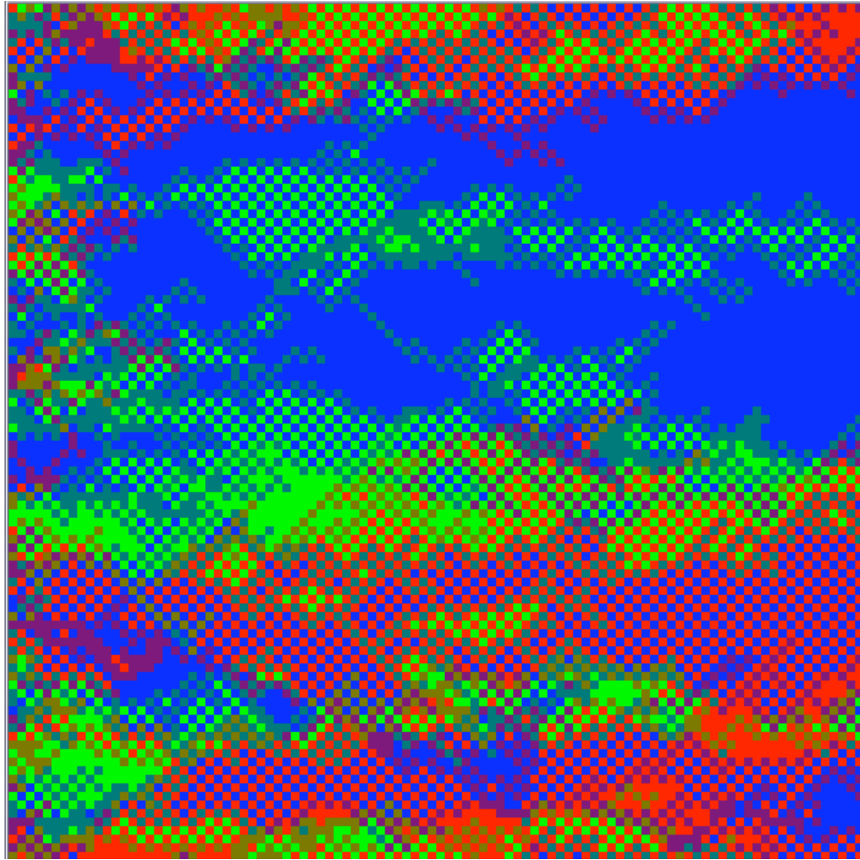
Our first coding attempt for this program had a bug in it that caused the iteration to proceed site by site, rather than generation by generation. The first generation would iterate as normal - in the following iterations, however, the first gene would get the information from its parents, then REPLACE the other first gene, so the second gene would use the already changed first gene (instead of its true parent) to determine it's color. This problem is not without its merits, however. In this case, the program is simply modeling more continuous results, with children becoming parents within the iteration. The results we got from this initial program were a rapid converging of traits – as in the graphic below, one color would begin to dominate after 1000 iterations or less. Mutations in this program would cause slower divergence, as one strategically placed mutated cell could throw off the entire outcome. Biologically, this could represent "inbreeding" of sorts – the traits converge very quickly, characteristic of inbreeding in animals or plants. Overall, much genetic diversity is lost in a relatively short amount of time.

Graphic (with the old program):
Population: 100, Steps: 100, Seed: 1, Mutation: 0.0001



After some coding adjustments, we came up with the correct program. This modified code gives a population that remains diverse for a lot longer – even when it does converge, it almost always converges to two traits (given no mutation). Here, the checkerboard pattern is much more prevalent, much unlike the solid colors we were seeing in the previous program. This makes sense, as one would expect a child cell with two green parents to be green itself. As time goes on, the population usually converges to a period-2 limit cycle of two colors in a checkerboard pattern – it is possible there is a case which converges to one trait, but these cases are few and difficult to find (100 Steps, 100 Population, 0165421 seed, 0.00001 Mutation Coefficient is an example of such a case, and even then it is only after a great many generations). Mutation has more of an effect when the code is correct; for example, if a pure blue or green cell randomly appears in a pattern of blue/green checkers, the pure color can blow up rather quickly. Even with one part in 10^{-4}*, the mutation can cause some impressive trait dominance – the color patterns can switch completely*

Graphic (with the new program):
Population: 100, Steps: 100, Seed: 1, Mutation: 0.0001



Conclusion

In the first program, we notice an exponential drop-off in trait diversity, though this is less noticeable in the second program. The second program eventually converges to two alternating traits – in the real world, this is a bit unrealistic, as many traits do have some form of dominance or recessiveness. For example, if we had a population of equally blue-eyed humans and brown-eyed humans, the brown-eyed trait would eventually win out due to dominance (though the blue-eyed trait would probably not disappear completely). Overall, this program is a primitive model of biological genetic evolution that can nonetheless provide some interesting insights into the effects of inbreeding and mutations on genetic traits.

We had predicted that the cellular automaton simulation would give a mostly chaotic, periodically stable population, much like the binary cellular automatons. Instead, we got an initially chaotic, but eventually stable period-2 attractor. Mutation can occasionally skew the results into becoming a different period-2 attractor (for example, a red and green checkerboard pattern instead of a blue and green one).