University of Colorado - Boulder

ECEN 2703: Discrete Math for Computer Engineers
Analysis of Conway's Game of Life

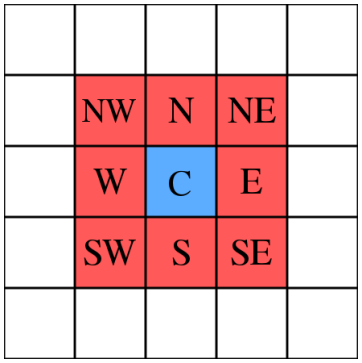
Zane McMorris

12/7/2022

Table of Contents

Overview of Cellular Automata

Cellular automata is generally described as a 2D grid of "cells" with various attributes that interact with each other using specific rules. These attributes can be as simple as a binary number representing either "alive" or "dead," a floating point number representing a weight, a color, or any other attribute. The rules that make up these systems must be strict but having few rules tends to constrict the system more than having an abundance of rules.



For this project, I explored Conway's Game of Life, a classic cellular automaton, it looks at each cell in a generation and decides whether or not it will live on to the next depending on the number of "alive" or "dead" neighbors it has. I used the "Moore's neighborhood" to define the cells to which the center cell would look to determine its next state. Moore's neighborhood looks at both the four sides of the center cell, as well as the cells that share vertices, giving eight total neighbors. The alternative to Moore's neighborhood would be Von Neumann's neighborhood, which only looks at the sides of the cell, giving four total neighbors. Conway's Game of Life usually looks at the Moore neighborhood because of the increased number of potential neighbors and, therefore, the increased interaction.

Conway's Game of Life is governed by 4 rules:

1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

These rules are evaluated once per cell per generation, meaning that the only changes that happen to the grid are from generation to generation and that if a cell were to die *next* generation, it would still count as alive for the current generation and allow neighbors to count it.

Creating Conway's Game of Life in Python

To create the game in Python, I had to decide on a few parameters before designing the system. First, I needed to decide what data structure I wanted to hold the cell's data. A 2D array of booleans would have worked for the general functionality. Still, it would be limited to only storing a single bit of information and couldn't hold any cell history or the cell's location in the grid. To make the cells as flexible as possible, I made them objects with several data members and functions that I could call in my application layer. To hold all these cells, I would need a 2D array to arrange the information simply. Still, I also wanted this grid to have extra functionality, such as history and member-specific functions. Ultimately, I decided that the Game of Life (GOL) object would hold: a 2D array of Cell objects, called a board, the starting board, the ending board, the ending condition, the number of generations to reach a steady state and an array to hold all previous boards, called board history. The final attribute to consider for these boards was whether to make them limited by the edges of the grid or if I should have edges wrap around to the opposite sides (left to right & top to bottom, and vice versa). This would allow the game to run more smoothly and without having to create a new rule to deal with edge conditions. I decided to have the grid wrap around because I felt it was the best approximation I could do with the Python knowledge I had and the scope of the project. This technically meant that the game was playing out on the surface of a torus.

Creating a GOL object in main would instantiate the first board randomly with either alive or dead cells. This random noise would be the foundation for all successive generations. The GOL object has an important function called "evaluate()" which would loop through the most recent board and apply the rules to each cell, and fill out a temporary board that would be the one in the next generation. This two-step process of filling out a new board and then assigning it to the current one separates the cell logic from generation to generation.

I implemented three primary end conditions to stop the game from continuing. The first was to see if every cell on the board was dead. The second was to check if the current board had a periodicity of two by simply checking it against the board history of two boards ago. If they were the same, then I knew that all subsequent boards would be the same, so I could end the process early. The final end condition was to end if the number of generations had exceeded 10,000. This ensured that the program generated the dataset efficiently and never got stuck on an infinite configuration that wasn't period two. Further exploration into this topic would include running these long-lasting configurations and looking at their long-term behaviors and seeing if they had interesting behavior. A modification was applied to the board history because the array size grew quickly as the game progressed. Each board was $gamescale^2$ bytes because each cell's living state was stored in a char '1' or '0'. The largest game scale I used was 128, meaning each board contained 16.3KB of information. The modification was that every 500 generations, the board history would clear to limit memory usage.

The game is then working at this point, and the generation rate was limited purely by the computer's performance. My laptop could run two program instances at 10 generations / second each. My desktop computer could easily run four and also run at 10 gen/sec, the computer I used

to generate the datasets 24/7 for a week. The program was modified slightly once again to see if the end conditions were reached, and if so, save the ending board, number of generations, and reason for ending in a .txt file, which I would parse in the future and perform data analysis on.

Overall, the program created three datasets with total sizes described in the following table:

Scale	# Of games	Data.txt size	Data.csv size
32	14167	30.23 MB	87 MB
64	8636	71.3 MB	210MB
128	2320	75.5 MB	224 MB

Data Analysis

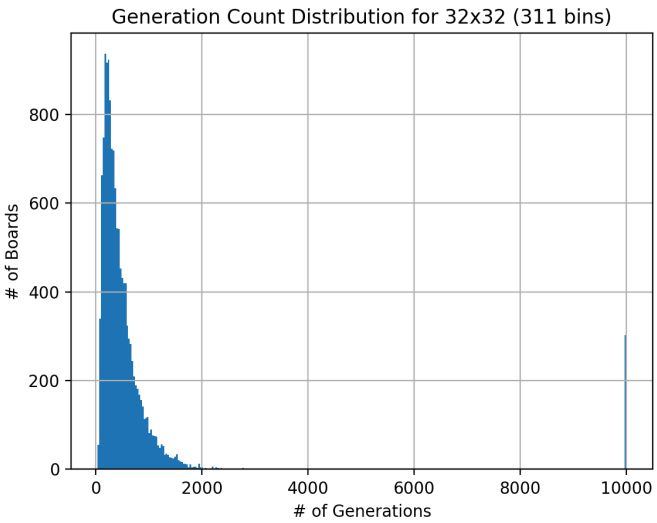
The first step to analyzing the data was to convert it into a usable format. I had some prior experience using the Pandas library in Python, so I knew what my end datatype would be, but the accurate parsing of the information took some time to get down. I created two 3D arrays to hold all the start boards and end boards, and two 1D arrays to hold the number of generations and end codes. I then created a data frame to store this information and used Pandas again to save it as a .csv for quicker reading in the future.

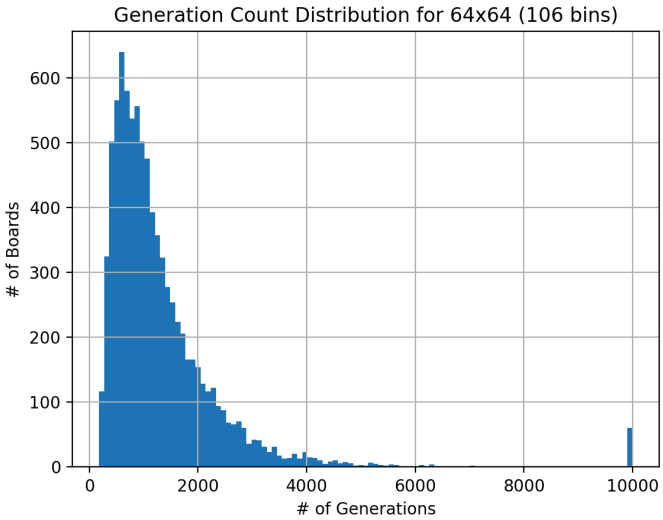
This means I only needed to parse my raw files once then I could use the .csv for repeated study. For the actual data analysis, I wanted to answer a few questions:

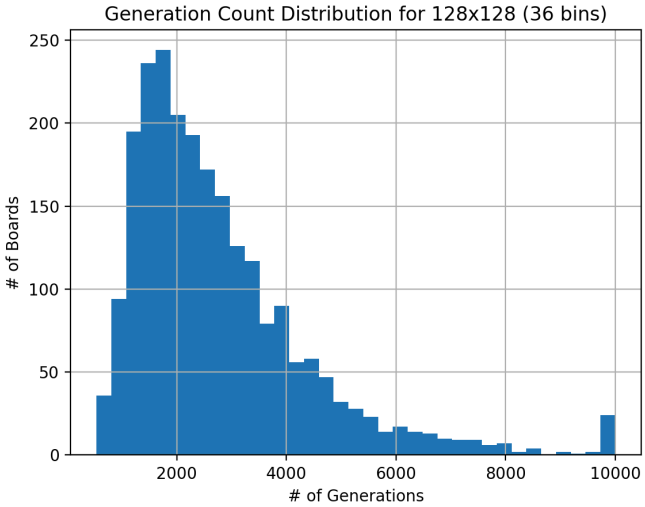
1. How does the time to steady state scale with the game scale?
2. What ending configurations are the most common, and does game scale affect this characteristic
3. What attributes do the “near-infinite” configurations have, and is it possible to predict their outcomes?

Part 1

With the data I have gathered, I can create histograms of each dataset and examine the distributions. Below are the histograms for 32, 64, and 128 game scales. An important feature is that each dataset’s bin count was determined using the Freedman Diaconis Rule and not arbitrarily chosen. To add transparency, the bin count is provided in the title of each graph.


Generation Count Distribution for 32x32 (311 bins)
# of Boards
0
200
400
600
800
0
2000
4000
6000
8000
10000
# of Generations


Generation Count Distribution for 64x64 (106 bins)
# of Boards
0
100
200
300
400
500
600
0
2000
4000
6000
8000
10000
# of Generations


Generation Count Distribution for 128x128 (36 bins)
# of Boards
0
50
100
150
200
250
2000
4000
6000
8000
10000
# of Generations

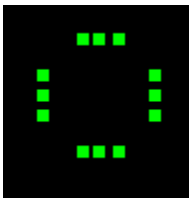
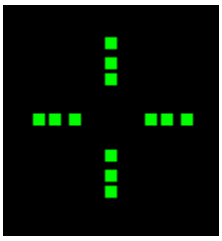
Described in the table below is the percentage of games simulated that exceeded the allowed generation limit, the average generation time excluding those that went beyond 10,000, and the most common distribution bin that the data fell into.

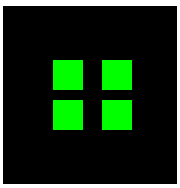
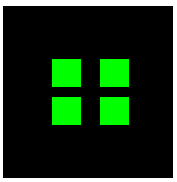
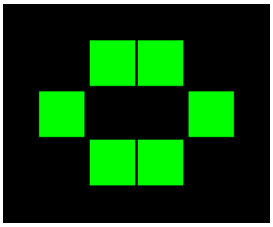
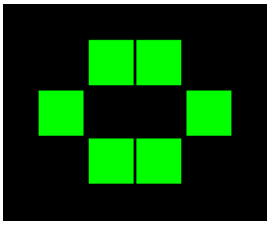
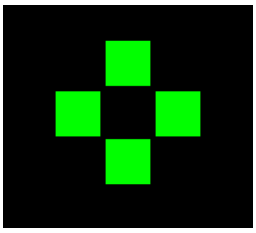
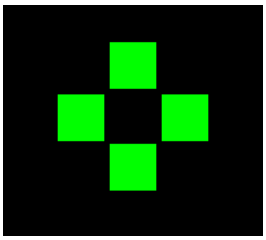
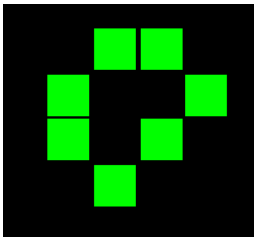
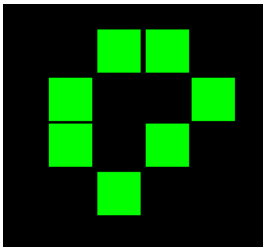
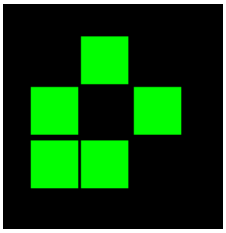
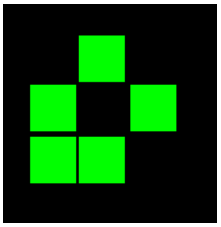
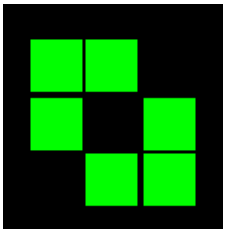
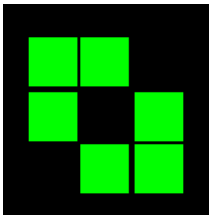
Game Scale	Average Generation Time	Percentage of whole	Mode Bin
32	457	2.13%	161 - 194
64	1251	0.69%	552 - 645
128	2703	**0.99%**	1616 - 1887

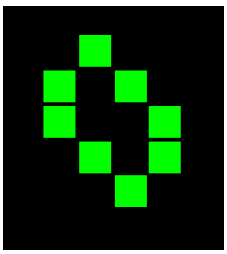
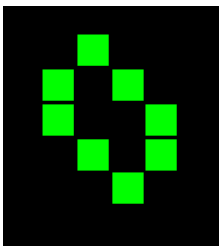
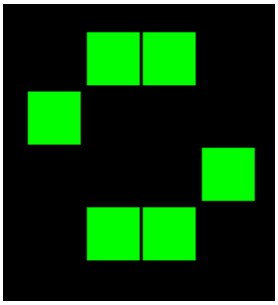
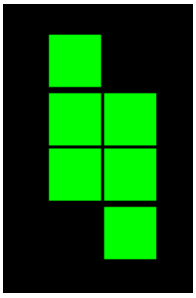
Using this table, we can see that both the mode and mean generation time increase with the increased grid size. This is likely because similar complex shapes are being created in different sections of the board shortly after creation. The larger grids allow these shapes to be in greater abundance and interact with each other over time. This causes the game to run for longer, and we can observe that the percentage of long-lasting boards should decrease with the game scale. Still, because the cutoff was arbitrarily assigned to be 10,000, the 128 grid was disproportionately affected. The rest of the data shows the relationship between grid size and generation time. Applying a linear regression to these 3 points gives us a near-exact fit, with an R^2 value of 0.994, showing that the relationship between game scale and average generation length is linear.

Part 2

Described in the table below is a list of some of the most common shapes and configurations found because they were period two. This list was compiled by examining the ending boards; no statistical analysis was performed.

First State	Second State	Explanation
		Sets of 3 cells in a row are stable and have period 2. Standalone rows or columns of 3 were exceedingly common.

		Sets of 2x2 cells are stable and do not oscillate at all because they sit perfectly on the living conditions for each cell.
		This configuration is also totally stable and occurs often
		Stable
		Stable
		Stable
		Stable

		Stable
		Period 2

This table is not exhaustive, and I'm sure there are many more configurations that my program would have halted or otherwise found uninteresting. This list, however, is representative of the most common causes for the program to halt due to the detection of a period-2 or entirely stable configuration.

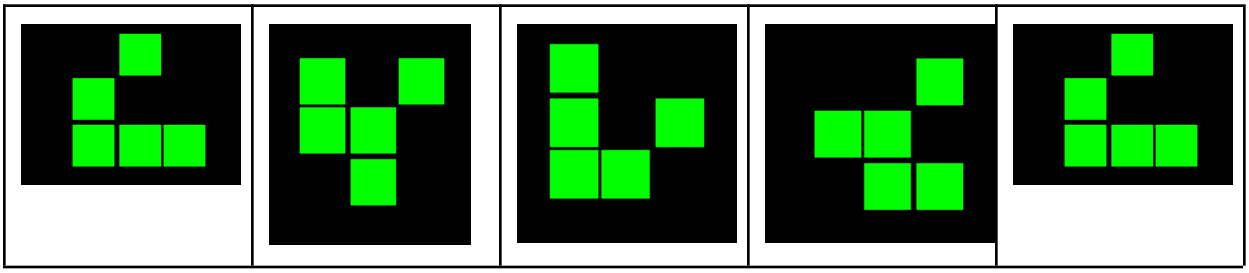
Gamescale didn't seem to have much effect on the variety of ending configurations, and this is likely because these stem from the initial state. By starting with random noise, the board quickly devolves into a chaotic system where groups of live cells start to affect residues of others. The random state of the initial board seems to have much more effect on the ending configuration, which matches our preconception of how these cellular automata behave.

Part 3

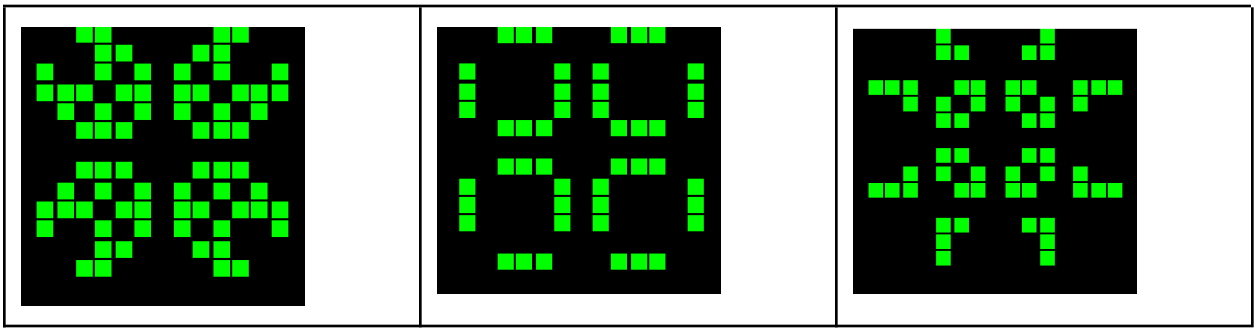
Next, I would like to briefly discuss the causes of the long-lasting configurations. The table below briefly describes how often and exactly how many games reached 10,000 generations.

Game Size	# of occurrences	% of total
32	23	2.15 %
64	60	0.69 %
128	23	0.99 %

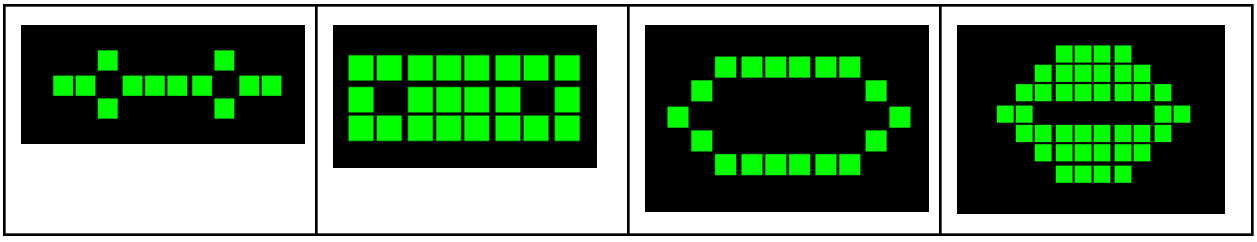
The most prominent cause of these was the config commonly known as a “glider”. Gliders traverse the board diagonally and are made of just 5 cells, arranged to nudge themselves forward by 1 cell every 4 cycles. The table below looks at the life cycle of the glider. These were the most common cause because their shape is very simple and requires very few cells. The only requirement for them to exist forever is a clear diagonal line free of any obstacles.

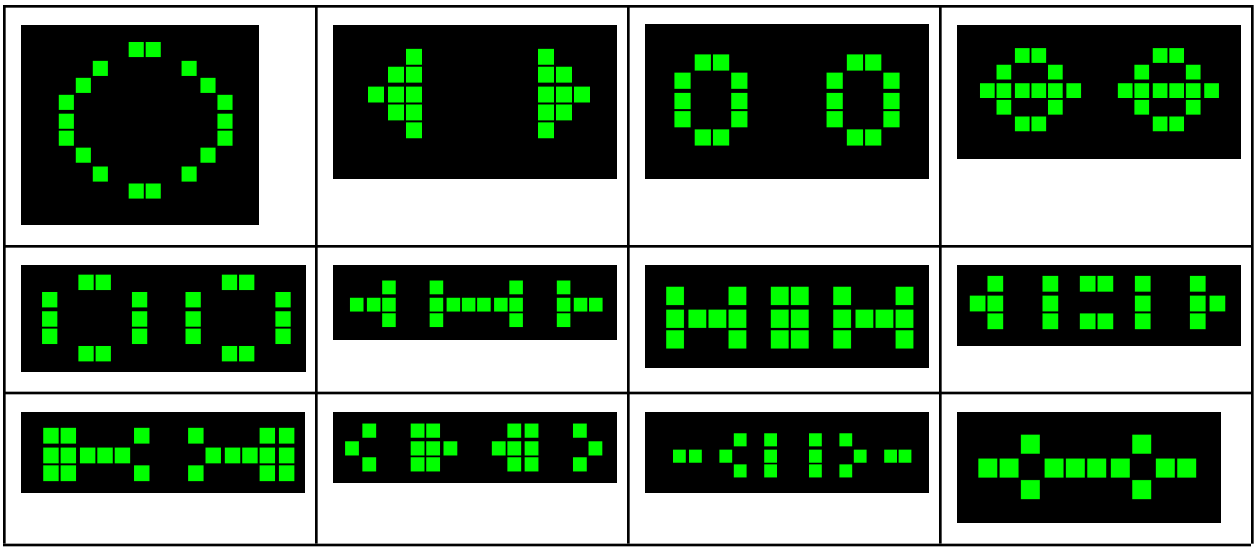


Gliders were the most common cause of these long-lasting systems, but occasionally some non-period two shapes would appear. The following two tables show the two non-period two configurations I found in my dataset just by visually inspecting the boards with the exit code 3.



This configuration occurred only once across the entire set. It had a period of 15 and occurred during a 64x64 game. This needed a relatively large amount of free space, which is likely why it only occurred once.





Given more time, I would have liked to make a script that would look through each board with over 10,000 generations and look for the shapes outlined above. This would then hopefully provide an estimate of exactly when the stable configuration was reached. Once enough data about stable configurations had been gathered, it would have been really interesting to try and use machine learning to create a model that could predict the behavior of any given board state. Further statistical analysis could have also been in the form of counting the individual shapes in the ending boards and finding the occurrences of each throughout the entire dataset to see if there was any correlation between game scale and what shapes came out of the noise.

Conclusion

Creating the implementation of Conway's Game of Life took much less time than I originally anticipated so I was able to quickly begin creating a GUI and generating data for my later analysis. The actual process of parsing my raw data files and turning them into a Pandas dataframe, the library I had used once prior, took more time than expected because of my unfamiliarity with some Python conventions. The creation of graphs, tables, and numerical analysis taught me how to do simple things in Python and how to draft programs quickly in the language. I'm glad that the language had to be Python for this project because it brought me out of my comfort zone, and gave me access to the powerful libraries that others have already written Python.

The data analysis showed that the generation time to reach steady-state depended on the game scale by a linear factor. End state configurations didn't depend heavily on the board size because the most common shapes were smaller than 3x3. Small shapes like this easily fit on the board and could coexist with others. In the rare instances where large shapes occurred, it can be

seen that very few other shapes exist on the board, likely because they were absorbed into the large shape previously in the game's lifespan.

More work can always be done on programming projects like this but knowing when to stop is important. A stretch goal I've been thinking about for my programming projects has always been to incorporate the NVIDIA CUDA library to run computations on my graphics cards, thereby accelerating computation time. I'd estimate that I spent 48 continuous hours running multiple instances of the generation program, and I think that using all the hardware in my PC could have made the process quicker or generated more data to analyze.