An Empirical Analysis of Life-like Cellular Automata

Justin Garrigus
Department of Computer Science
University of North Texas
Denton, U.S.A.
justingarrigus@my.unt.edu

***Abstract*—The study of cellular automata is an interesting topic in the research of complex emergent systems stemming from simple formal rule sets. In these areas, a small list of rules applied to a grid of cells can create structured objects, chaotic systems with unpredictable behavior, and even entire programs built from discrete logic gates. Cellular automata were an integral field of research in the development of computational theory, and is still an expanding topic found in areas like the Journal of Cellular Automata.**

The most popular form of cellular automata is Conway's Game of Life which defines the situations in which cells on a two-dimensional grid are enabled or disabled based on the state of neighboring cells. Different variations of this simple rule set are named *life-like automata*. While there are a large number of different rule sets, only a few of them have been studied in any detail. This paper will define a collection of variables present in grid-based automata and will use tools from empirical analysis to quantitatively describe correlations, causations, and similarities between the 11 most popular variations.

***Index Terms*—Cellular automata, emergent systems, chaos theory, empirical analysis**

I. Introduction

The concept of *emergent systems* has wide implications ranging from physics and chemistry to mathematical domains like complexity theory and chaos theory. The core idea is that simplistic, rule-based "axioms" that cannot be decomposed into any sub-concepts can form the basis of tremendously complex systems with wide variability and context-sensitivity. This is most easily seen in the context of subatomic particles, in which seemingly-simple rules define the position and momentum of quanta lead to the physical world that humans inhabit. Researchers can utilize these same concepts to build new systems with simple rules that can form the foundation of dynamic and expansive structures.

Cellular automata are a field of research that represents the core elements of these systems. An automata is a logic device consisting of *states* and *transitions*, in which the transitions define how one state can become another state. The most important kind of automata is the Turing machine [1], [2], which has been used as a tool to define the computability of algorithmic problems in terms of a formal system. The Turing machine gives an initial state, a list of steps to perform, and memory in the form of a linear "tape", and has been proven to be capable of represent any computable problem. Since its original definition in 1936, it has since been used as a minimum-power system that other computing devices like modern hardware computers must be defined in terms of in order to prove their ability to solve problems.

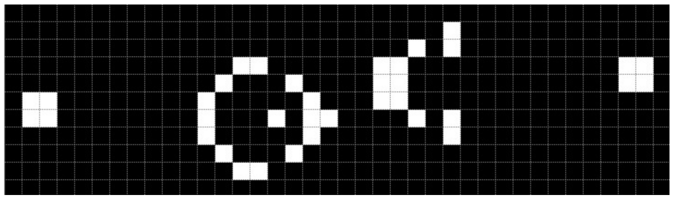


Fig. 1. A grid configuration of enabled (white) and disabled (black) cells that yields a "gun" producing one "glider" every 30 time steps.

Cellular automata extend this idea by representing each "cell" in a system as an independent automaton. The most popular form of cellular automata is Conway's Game of Life [3], [4] which defines the system in terms of a two-dimensional grid with only two possible states–*enabled* and *disabled*–where the state of a cell at the next time step is determined by (1) its state at the current time step, and (2) the current state of the eight surrounding cells in the cardinal and intermediate directions. The rules of the system can be defined as follows:

1) A cell which is currently disabled will become enabled at the next time step if there are exactly 3 enabled neighboring cells.
2) A cell which is currently enabled will become disabled at the next time step if there are not exactly 2 or 3 enabled neighboring cells.

The rules can be summarized with a concise syntax as "**B3S23**", or "**B**irth with 3 neighbors, **S**urvival with 2 or 3 neighbors".

While these rules may appear to be simple, they lead to a huge amount of complexity in the form of abstraction from emergent systems, where high-level structures appear from repeated patterns in the cell state. An example of this is the "glider gun" shown in figure 1, a structure that creates an infinite number of sub-structures at regular intervals that move across the grid on their own. The regular pattern of the input state leads to cells being activated and deactivated in such a way that causes an "object" to traverse the environment. This object could interact with others in future time steps in new and strange ways, bringing about more objects with higher levels of abstraction. An example of these increasing levels of abstraction is shown in a universal Turing machine [5], in which gliders combine with "stoppers" and "reflectors" to form logic gates, which can construct a finite state machine, which supplies input with a stack, which is proven to be capable of solving any computational problem.

While Conway's Game of Life is the most well-known cellular automata, there are a number of variations on the rule set **B3S23** that are shown to have similar properties. Rule sets which are based off of Life are known as life-like cellular automata, and can be expressed with the same syntax as before. Examples of this include "34 Life" (B34S34), "High-Life" (B36S23), and "Day & Night" (B3678S34678). Different aspects of these automata have already been studied including their tendency to quickly explode in complexity, form well-defined meta-structures, or interact to simulate events.

This paper will analyze these life-like cellular automata for a number of variables and attempt to find similarities between these variables by using methods in empirical analysis. We define five such variables that are of interest to us:

- `Count`: the number of enabled cells in the grid.
- `Area`: the area of the bounding box containing all points on the grid.
- `Density`: the `count` divided by the `area` of the grid.
- `Entropy`: the number of cells that are different between the previous state and the current state.
- `Clumpiness`: the average smallest distance between each enabled cell, in which a low value means values are clumped very closely together with few cells isolated with no close neighbors.

We create a robust simulator that is capable of quickly running individual tests of a rule set and writing the results of these variables to a log file which a separate program can parse and create graphs to visualize the output of. We design the programs such that users can easily replicate the results from our study with a simple set of commands.

We will first analyze these variables in different simulations of the automata and then determine if patterns exist between each variable by utilizing tools such as linear regression, resampling, cross validation, and t-test. The paper is organized as follows: section II gives additional background to the field of cellular automata; section III describes how the data set we analyze is generated; section IV gives an overview of each component part of our project; section V outlines how each portion of the project was implemented; section VI provides the results of the different experiments and analyzes the data and draws conclusions from any patterns we find; and section VII concludes the paper by detailing the work completed.

II. Related Work (Background)

The foundation of cellular automata has its roots in John von Neumann's research into self-replicating machines. The von Neumann cellular automaton was the first in the field, representing a positional rule-based system with independent "nodes" connected regularly to other nodes. In this design, there are three possible states and individual nodes are connected only in the cardinal directions; other automata which posses four-neighbor cardinal connections are thus said to have von Neumann neighborhoods. von Neumann was particularly interested in the relation between artificial discrete automata and the human neurological system where individual neurons are seen as switches that convey "all or nothing" information with varying degrees of high-level complexity [6]. von Neumann showed how a "universal constructor" could be designed that is capable of creating a machine according to some description, which is able to conceptualize biological genetics as a self-replicating machine [7].

Conway's Game of Life was proposed by John Conway and published in the October 1970 issue of Scientific American's *Mathematical Games* column written by Marin Gardner [4], which gives the ruleset of the game and various basic structures that are destroyed after some duration of time steps or repeat with a specific period. Shortly after in November, William Gosper submitted a design to the column containing a finite pattern of cells that yield an infinitely-growing pattern of enabled cells travelling in some direction down the grid. This led to the creation of another newsletter specifically aimed towards cellular automata named LIFELINE.

Since then, there have been a large amount of research (even in recent years) that study these systems and use their conceptual frameworks to design real-world physical and logical processes like text encryption [8], image processing [9], and traffic flow models [10]. Additional resources include *A New Kind of Science* by Stephen Wolfram [11] which describes the wide breadth that simplistic cellular automata have in diverse real-world domains, and the Journal of Cellular Automata which has published yearly volumes since 2006 continuing into the present day (2023).

III. Dataset

This project consists of simulations run on a collection of 11 life-like cellular automata and a subsequent analysis of the resulting data. Due to the simple rules used to produce the data, a program can be created which executes different simulations and writes their important variables to a file. For our project, data collection is performed by initializing a configuration with a two-dimensional grid size and an initial seed, and the simulation is allowed to execute until some time limit is reached. The seed is able to be provided randomly (which gives a random configuration of enabled and disabled cells) or with an inputted number (which can be used as a bit-mask to selectively enable some cells).

IV. Detailed Design of Features

This section describes a high-level overview of each component of the project and the goals each component seeks to achieve. There are two main elements that compose our design which will be discussed including the simulator and the parser.

A. *Simulator*

The simulator seeks to run a configuration for a cellular automaton given its rule set, grid size, execution count, and max duration allowed. It executes each iteration of the simulation, collecting the variables mentioned in section I and appending them to a log file. Each variable can be collected by measuring only the (1) state of each cell and (2) the position of each enabled cell on the grid, which can be performed with simple binary logic operations.

An important aspect of the simulator is the optional GIF creator that displays each state of a single simulation. Grids of enabled and disabled states can be interpreted as images of black and white pixels, and these images can be combined to form animated GIFs. This is helpful to visually see the implications of variable values; for example, the "Replicator" rule set has the highest entropy out of any of the 11 rule sets, but this is not immediately apparent when the rule description is shown (B1357S1357). Although, after viewing the generated GIF, we can see that the simulation given after using a randomly-generated initial state has a quick growth that fills the entire board and has nearly every cell blink between off and on states. In fact, viewing multiple GIFs shows that this particular rule set has the effect of replicating and expanding any initial state it is given.[1]

B. *Parser*

The parser is a tool to both (1) manage the automatic execution of the simulator to collect data and (2) read the collected data and analyze it with figures and statistical models. It is able to read multiple log files at once to compare the different variables with multi-dataset graphs, which are each given in section VI. The specific graphs that it supports are multi-boxplot comparisons, correlation heatmaps (with Pearson's coefficient), linear regression (with calculation of an R^2 and root mean-squared error metric), multi-histogram comparison with inclusion of a confidence interval of a statistic, and density graph. Additionally, operations on datasets are supported like removal of outliers, averaging of statistics, and bootstrapping.

V. Implementation

The simulator and the parser both have different needs that must be addressed, which makes their execution environments very different from each other. This section describes these execution environments as well as the specific implementation details that are important to know in order to understand how they operate.

A. *Simulator*

First, the simulator required high performance and fixed-width data types so that many different simulations could be run in a short amount of time so that the highest amount of data could be collected and analyzed; we will see in section VI that approximating a normal distribution of the variables is important and only possible if lots of data is collected.

[1]GIFs of each of the 11 rule sets can be viewed at https://youtu.be/jJTz-WpF_ZQ?si=evZDG2G4OA0dzmq9

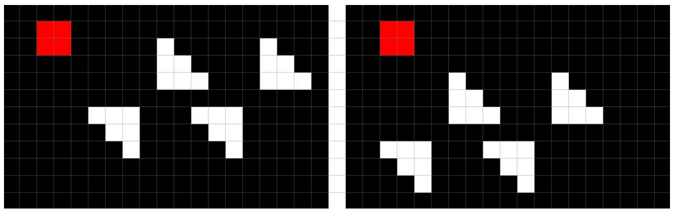


Fig. 2. An example of two identical initial states despite being located at different positions on the grid (relative to the red dot).

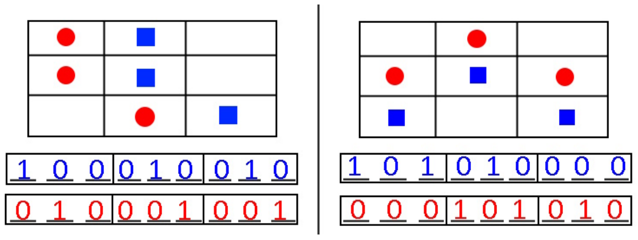


Fig. 3. Graphical demonstration of the state normalization process. The shapes represent enabled cells, with blue squares being the initial state and red circles being the normalized state. Here, the grid is 3×3 cells large, and the resulting hash value is a 9-bit value.

The simplicity of the simulator meant that most of it could be designed in basic C with a few exceptions. Primarily, the GIF-creation capability was an integral part of the data-visualization process and necessary to properly understand how the variables should be interpreted, and a helpful minimal-size library was found that easily creates a GIF from a color (integer) buffer.[2].

The other library that was required was an implementation of a hash-set data structure in C.[3] Cellular automatas have an interesting property that their next state depends entirely on their current state: in other words, if two separate grid configurations reduce to the same initial state, then we can guarantee they will generate the same variables at every time step. An example of this is shown in figure 2, in which one initial grid configuration is exactly the same as another, but with a different position relative to a reference point. This shows how two states can be the same while having different grid values. Since cellular automata can be infinite-size in theory, it would be disingenuous to consider the two grid configurations as separate data points, so we need a method of hashing a state into a value that can be compared with another hash to determine equality.

The way this is determined is with bit-shifting rules shown in figure 3. In our design, the simulation grid is implemented as an array of bits (or, more exactly, an array of bytes that individual bits are extracted from) in row-major order, in which the most-significant bit is the lower-right cell in the grid and the least-significant bit is the top-left cell in the grid. In order to determine a hash for a state, the following steps are executed:

[2]https://github.com/lecram/gifenc
[3]https://github.com/avsej/hashset.c

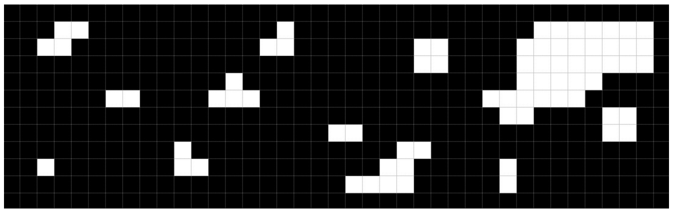


Fig. 4. An example of a grid with a clumpiness very close to (but not exactly equal to) 1.000.

1) A bit array is formed as a copy of the given state. The bit-array represents a row-major grid of cells.
2) A bounding box is created that surrounds each enabled bit of the entire structure. The width of this box is b_w and the height is b_h.
3) The cells are moved as far left as possible in the grid. We get the horizontal position of the leftmost cell in the structure by dividing the bit array into groups of b_w cells each, and finding the bit value that is furthest to the right in each group. Then, the leftmost position is the minimum value out of the positions of the rightmost enabled bit in each group. We right-shift the number by this amount.
4) The cells are moved as far up as possible in the grid. We repeat the above steps: we divide the bit array into groups of b_h cells each, in which the least-significant group is position 0, the next least-significant group is position 1, etc. Finally, we right-shift the number by $b_h \times position$ bits.
5) The resulting bit array is the hash value.

Each run of the simulator may execute many dozens or hundreds of simulations. To prevent repeated data from being logged, we create a hash value according to the steps shown above and compare it with values already logged previously. If the hash-set already contains the hash-value, then the simulation is skipped; otherwise, it continues and its hash-value is added to the hash-set.

Each of the five variables stated in section I can be easily implemented by calculating bounding boxes and counting enabled bits. The simulation is double-buffered so both the current state and the previous state are stored for the purposes of finding the entropy of the simulation. The only variable which is not entirely intuitive is the *clumpiness*: this is found by finding the minimum distance from a point p_1 to any point p_2. An example of this is shown in figure 4, in which almost every enabled cell in the grid is a neighbor with another cell. Although, there is one cell in the bottom-left corner which has no immediate neighbors, with its closest neighbor being 3 cells away in the diagonal direction. Since there are 73 enabled cells on the grid, this would lead to a clumpiness of $\frac{(1\times3)+(72\times1)}{73} = 1.027$.

Finally, the log files need to store each of the five variables for each simulation so the parser can create graphs from it. These log files take the following format shown in figure 5. The constant `NUM_ITERATIONS` is shared between the simulator and the parser so it does not need to be stored, but all other values are generated by the simulator itself.

```
simulations: uint64
Repeat simulations times:
    seed: uint64
    clumpiness: double
    count: uint64
    Repeat NUM_ITERATIONS times:
        area: uint64
        entropy: uint64
        density: double
```

Fig. 5. File structure of simulation logs. All values are 8-bytes long and are determined by the automata rules being applied to the seed (except for `NUM_ITERATIONS`, which is a global constant value).

For this project, we interpret the variables `clumpiness` and `count` as independent variables measured only in the initial state of a simulation, while `area`, `entropy`, and `density` are dependent variables that are measured each time step.[4] We will attempt to predict these dependent variables by using the independent variables in section VI.

B. Parser

While the simulator required high performance and a precise design, the parser could afford to be slower and more abstract, but it had a requirement of being easier to program in to encourage quick prototyping of data analysis tools. The parser would be used to generate figures describing the data–along with managing the execution of the simulator if those log files do not exist at parse-time–so it should only need to be run once as opposed to the simulator which would run potentially millions of times. In addition, the parser requires lots of external libraries to be included featuring data-processing and statistical libraries. All of these qualities combine to make Python an ideal fit for this program.

The most important part of the parser is the ability to generate figures from the simulation logs. The logs are parsed using the same format described in figure 5 and converted into lists of numbers, but the popular libraries `matplotlib` and `seaborn` are used to visualize the data with graphics (box plots, heatmaps, scatter plots, histograms, and density plots), and `scipy` and `sklearn` are used to simplify the implementation of statistical tools (T-tests and linear regression).

Each rule set and human-readable name (e.g., "B3S012345678" and "Life without Death") for the 11 life-like automatas are included directly in the file next to execution flags, so the user can easily configure their execution environment to generate the types of data they require.

[4]The decision to choose this division of independent/dependent variables was arbitrary. Future work can see if interesting effects arise from a new division.

VI. Results and Analysis

This section will describe the results received after running the project. The same results shown in this paper can be easily received by running the command `python3 parse.py`, which will queue up each simulation, GIF-creator, log-generator, and data-parser to generate the same figures shown here. Every simulation will run for 32 iterations with 65536 simulations executed per rule set (with duplicate configurations removed as described in section V) in a grid size of 48×48 cells. Each rule set was run two times: once with randomized seeds (ranging from 0 to $2^{16 \times 16}$) and once with seeds incrementing from 0 to 65536.

A. Random Large Seeds

First, we will compare the averages of the three main statistics (area, entropy, and density) between the different rule sets. These logs were generated from random seeds, so the `count` had the potential to range from 0 to $16 \times 16 - 1$, the `clumpiness` from 0.00 to 15.00, the `area` from 0 to 1, the `entropy` from 0 to $16 \times 16 - 1$, and the `density` from 0.00 to 1.00. The results are shown in figure 6.

Immediately, we can notice several obvious correlations between the statistic averages and the visual representations shown in GIFs of the rule sets. For starters, Replicator (B1357S1357) has the highest area and entropy, which is apparent due to its tendency to rapidly fill the entire grid and its quick blinking due to having a birth and survival neighbor count equal to each other (B = S = 1357). Alternatively, Anneal (B4678S35678) has the smallest entropy with a widely inconsistent (but still very low on average) area, due to it becoming too small and vanishing very quickly. In our environment, we consider a simulation which ends early–before the entire 32 time steps are complete–to contain 0-padded values for the area, entropy, and density, which would reduce the averages considerably if one rule set tended to end very quickly. As such, we would likely see different results for Anneal if the iteration size was smaller than 32. Anneal does have the highest average density though due to the fact cells stay alive with a large number of neighbors (B = 4678, S = 35678), unlike most other rule sets which experience an "overpopulation" effect where they die with too many neighbors (e.g., Life with the rule set B3S23).

Next, we can see how the initial starting state variables `count` and `clumpiness` correlates with the average `area`, `entropy`, and `density` for a specific execution of a simulation. This step is extremely dependent on the maximum number of simulated time steps since it favors rule sets which feature "stagnation" and an overall consistent and predictable population more than they do chaotic systems that change very frequently. This is shown in figure 7.

The correlation graphs initially show promising data for predicting average qualities from an initial value. The correlation values are calculated from a Pearson's coefficient describing the correlation between two lists of equal-size data, in which $+1$ indicates a positive correlation, -1 indicates a negative correlation, and 0 indicates no correlation. In particular, Diamoeba (B35678S5678) has average `area` negatively correlate with initial simulation `count`, and Replicator (B1357S1357) has average `density` positively correlate with `count` as well. Clumpiness does not share this same predictability for any statistic likely due to the fact it is a chaotic statistic that is likely to change almost immediately in even the next iteration of the simulation. In fact, clumpiness should probably be used in predicting short-term statistics rather than long-term ones like those used in this paper. We can plot a linear regression model from the datasets used previously (e.g., relating initial `count` and average `entropy` for Diamoeba, and initial `count` and average `density` for Replicator),

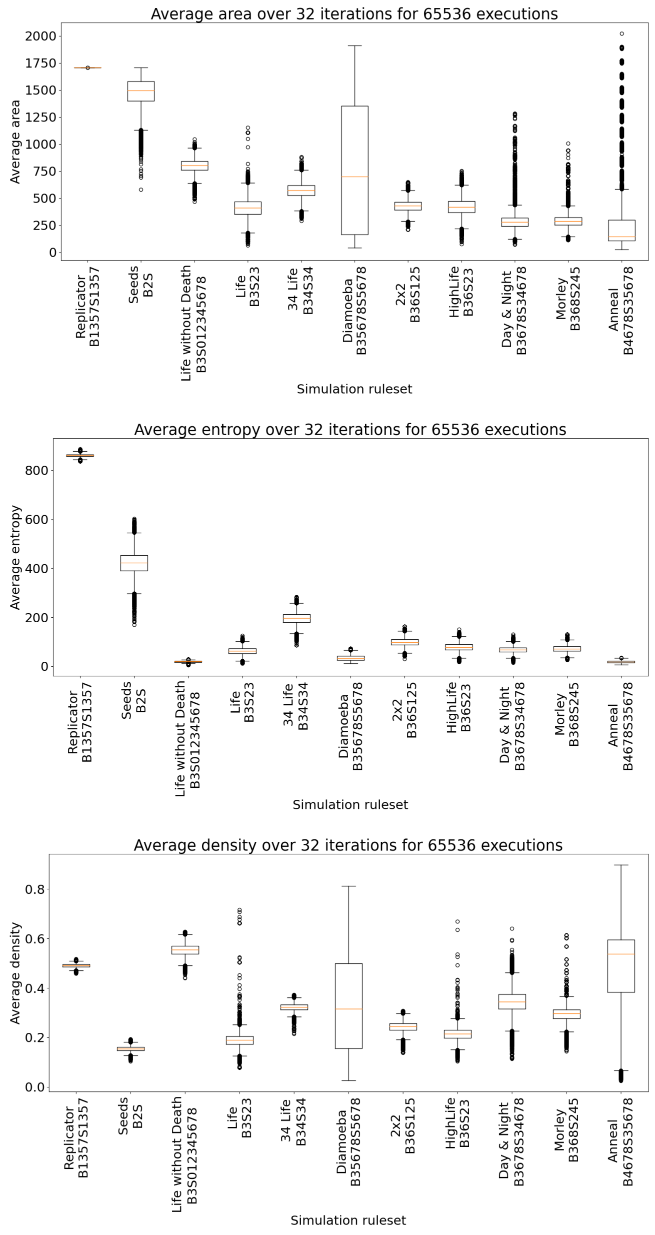


Fig. 6. The average area, entropy, and density for random initial starting states in each of the 11 rule sets.

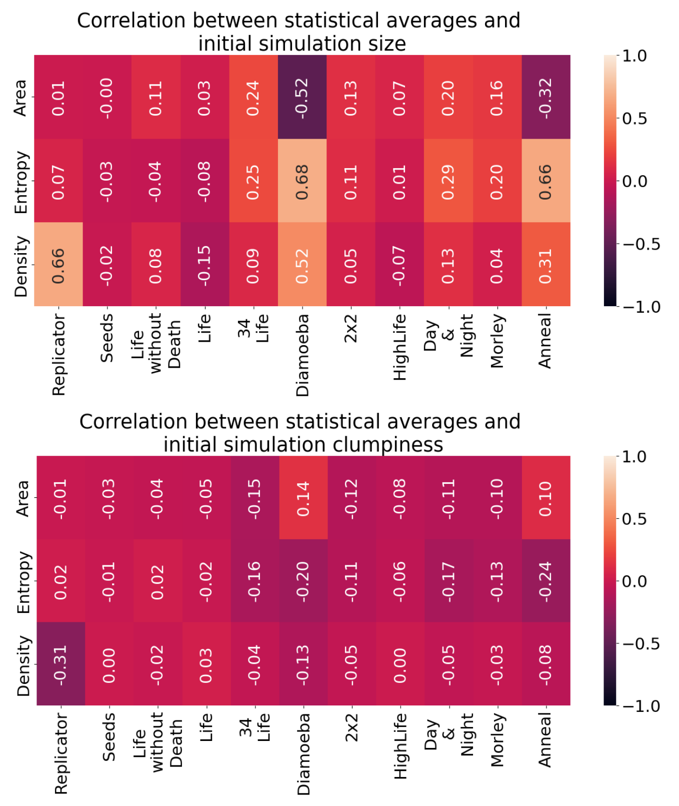


Fig. 7. The correlation between the initial count and clumpiness with the average area, entropy, and density.

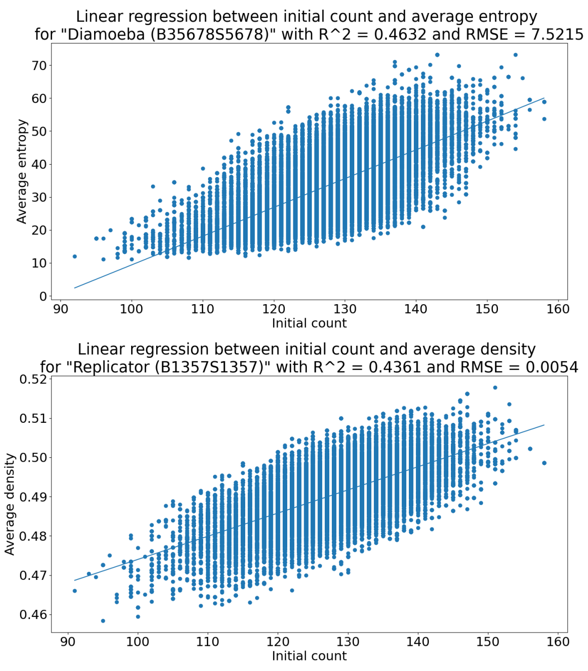


Fig. 8. Linear regression modelling between the two most correlated statistics shown in figure 7.

shown in figure 8.

The model is created from a train-test split of 67%/33% and evaluated with an R^2 and root mean-squared error $RMSE$ metric obtained from comparing the fit of the line to the training data and testing data, respectively. We find that Diamoeba has an $R^2 = 0.463$ and an $RMSE = 7.522$, while Replicator has an $R^2 = 0.436$ and an $RMSE = 0.0054$. Both R^2 values fit moderately well to the data (and, as can be seen, they adequately capture a clear trend of the plot points), so we can be capable of utilizing this linear regression plot to estimate stats of the data given the initial state.

Although, we can see how much of an effect outliers have on the dataset by repeating the previous three figure sets with outliers removed. In our case, we have three dependent variables (`area`, `entropy`, and `density`) and two independent variables (`count` and `clumpiness`) so we interpret an "outlier" to be a single dependent variable which is more than one standard deviations away from the mean for a given statistic. The point is only added to the reduced dataset if all three dependent variables pass this test. The new average statistics are depicted in figure 9.

The biggest difference between figure 9 and figure 6 is the variability in the data. The upper and lower quartiles in the box plot are significantly more compressed around the mean, except for the average area and density of Diamoeba (B35678S5678) and Anneal (B4678S35678), which still experience relatively large variability likely due to the wide distribution of values in figure 6.

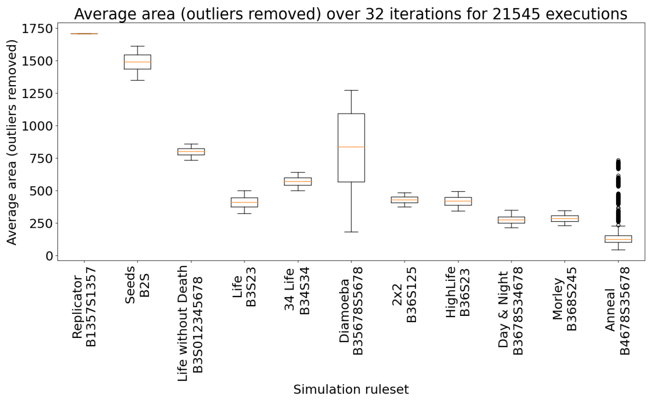
Figure 10 repeats the correlation comparison between each independent and dependent variable. It is clear that the removal of outliers in the dataset reduced the overall correlation between each variable, with Pearson's coefficient dampening closer to zero in all cases. We can compare the same two variables we did previously in the linear regression test next.

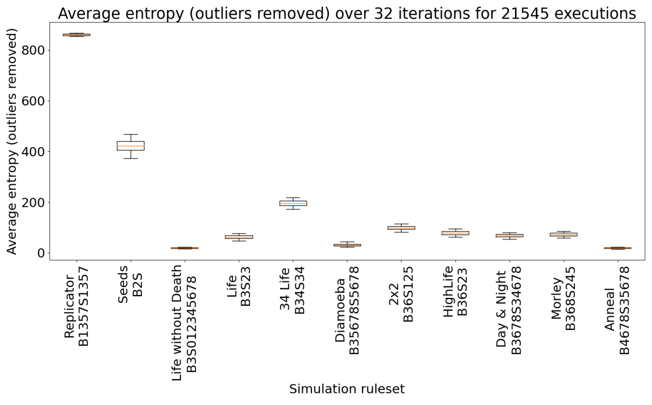
Linear regression shown in figure 11 shows a reduced R^2 score, but an improved $RMSE$ metric in both cases. This means the ability of linear regression to fit to training data deteriorated, but its predictive ability to generalize to unseen data has improved. This is arguably a more important metric to optimize, so removing outliers can be seen to have a positive effect on the model.

B. *Incremental Small Seeds*

As opposed to the previous section, this section will run simulations on an exhaustive search of the initial starting states from 0 (no enabled cells on the grid) to 2^16 (a 4×4 grid of enabled cells). Ideally, we would use a much larger initial grid size, but this becomes extremely expensive and infeasible to run. Our goal in this section is to see how well we can generalize small initial states to large states, and how well two rule sets compare when their exact input states are equivalent as opposed to the previous section which had very different starting states in each simulation run.

Figure 12 shows a comparison between each of the independent variables, as we have shown previously for the random (figure 6) and random-no-outlier (figure 9) datasets. This time, we can observe new curious artifacts emerge from the data.





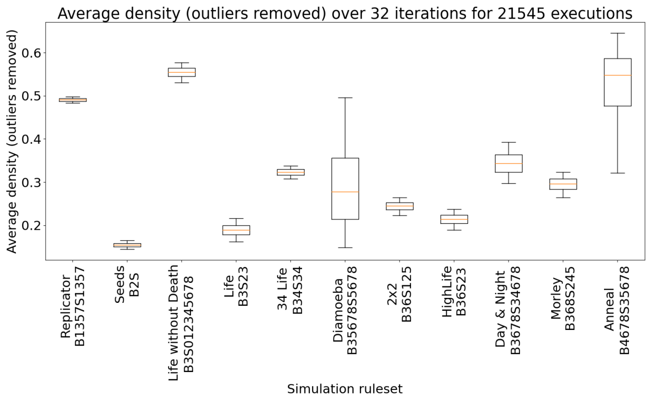
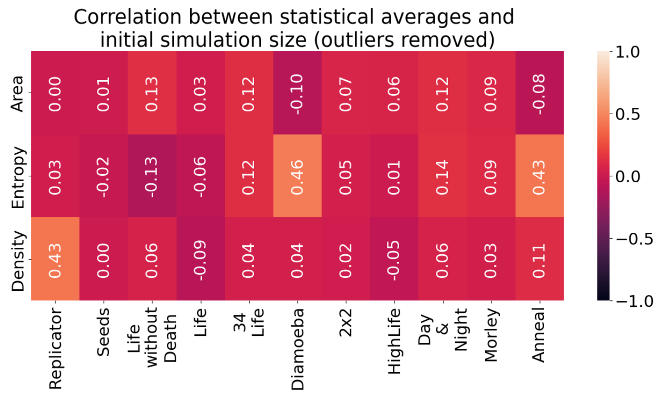


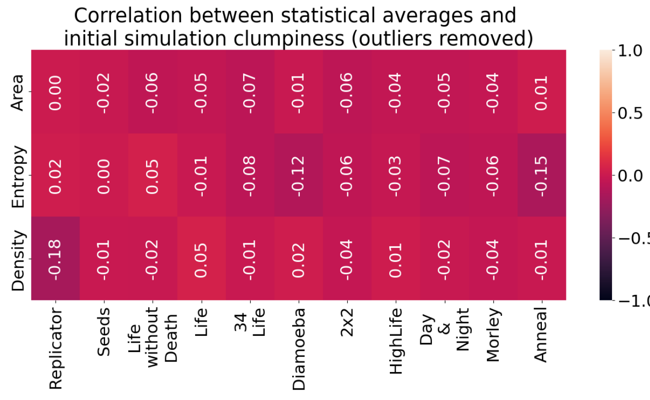
Fig. 9. The average area, entropy, and density for random initial starting states but with outlier values removed.

In particular, we can see that the variation in the area for each rule set increased compared to the random initial states. This may indicate that area is highly dependent on the number of enabled cells, especially for a small-duration limited-size grid like the one shown in our simulator. Despite this, entropy remains almost the same as before. This is probably due to the fact the change in cell state is consistent regardless the size of the grid; most grid configurations will have very similar changes in their cell status.

Next, we take a look more closely at the Life (B3S23) and HighLife (B36S23) rule sets to observe how they differ from each other. We can see that the three variables shown in figure

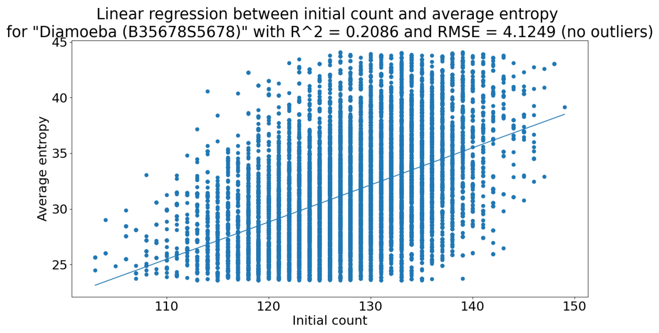


	Replicator	Seeds	Life without Death	Life	34 Life	Diamoeba	2x2	HighLife	Day & Night	Morley	Anneal
Area	0.00	0.01	0.13	0.03	0.12	-0.10	0.07	0.06	0.12	0.09	-0.08
Entropy	0.03	-0.02	-0.13	-0.06	0.12	0.46	0.05	0.01	0.14	0.09	0.43
Density	0.43	0.00	0.06	-0.09	0.04	0.04	0.02	-0.05	0.06	0.03	0.11



	Replicator	Seeds	Life without Death	Life	34 Life	Diamoeba	2x2	HighLife	Day & Night	Morley	Anneal
Area	0.00	-0.02	-0.06	-0.05	-0.07	-0.01	-0.06	-0.04	-0.05	-0.04	0.01
Entropy	0.02	0.00	0.05	-0.01	-0.08	-0.12	-0.06	-0.03	-0.07	-0.06	-0.15
Density	-0.18	-0.01	-0.02	0.05	-0.01	0.02	-0.04	0.01	-0.02	-0.04	-0.01

Fig. 10. The correlation between the initial count and clumpiness (independent variables) with the average area, entropy, and density (dependent variables), where outlier dependent variables are removed from the correlation test.



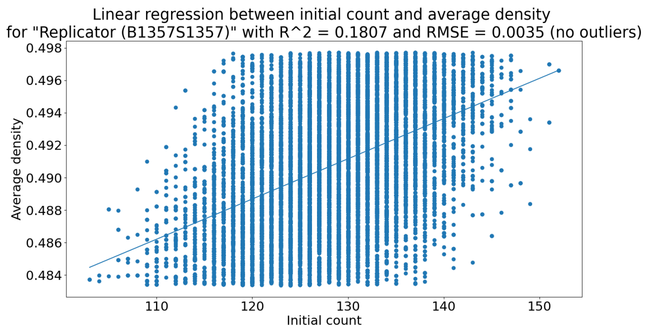


Fig. 11. Linear regression modelling between the two most correlated statistics shown in figure 10, with outliers removed from the original dataset.

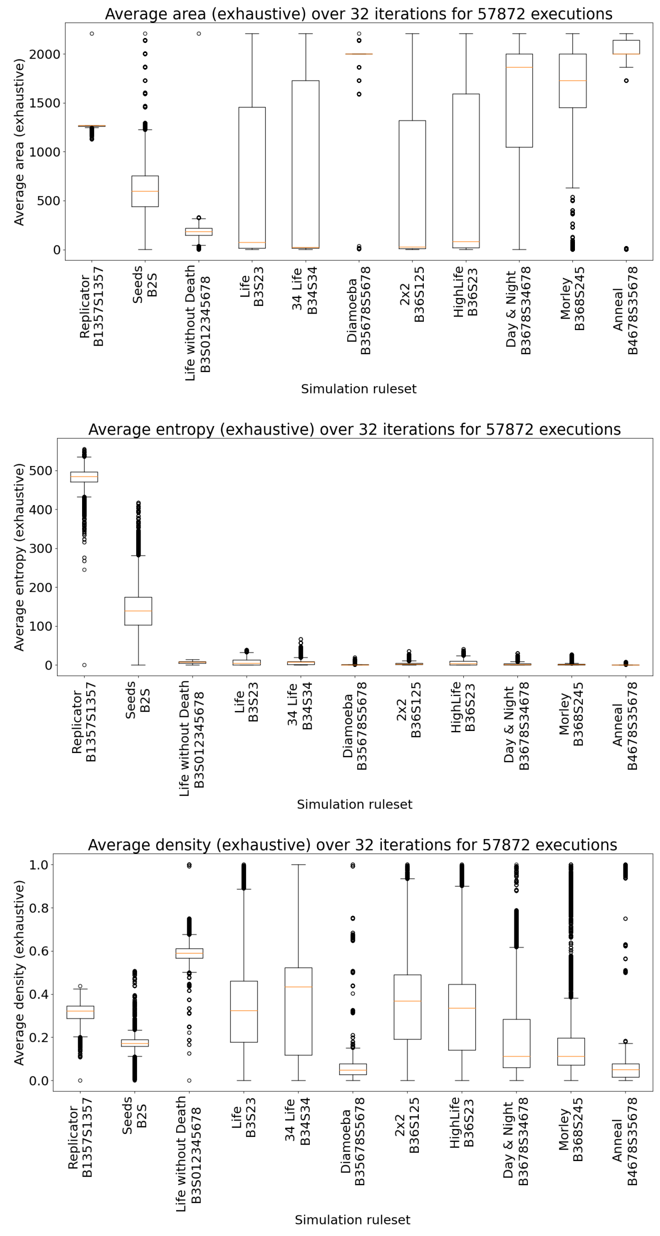


Fig. 12. The average area, entropy, and density for regular initial starting states.

12 have values that are nearly identical to each other, so we can create a hypothesis that the two rule sets do not differ in their `area`, `entropy`, and `density`, at least for the given simulator configuration. Framing the problem in this way allows us to perform a t-test on each individual statistic; if all three t-tests value p-values greater than 0.05 then we fail to reject the null hypothesis, meaning we are unable to tell if the points came from the same population (or rather, we are not confident that the three variables are significantly different based only on the rule set). The results of these tests and a comparison of the values is given in figure 13.

From this figure, we can tell that the data distributions are

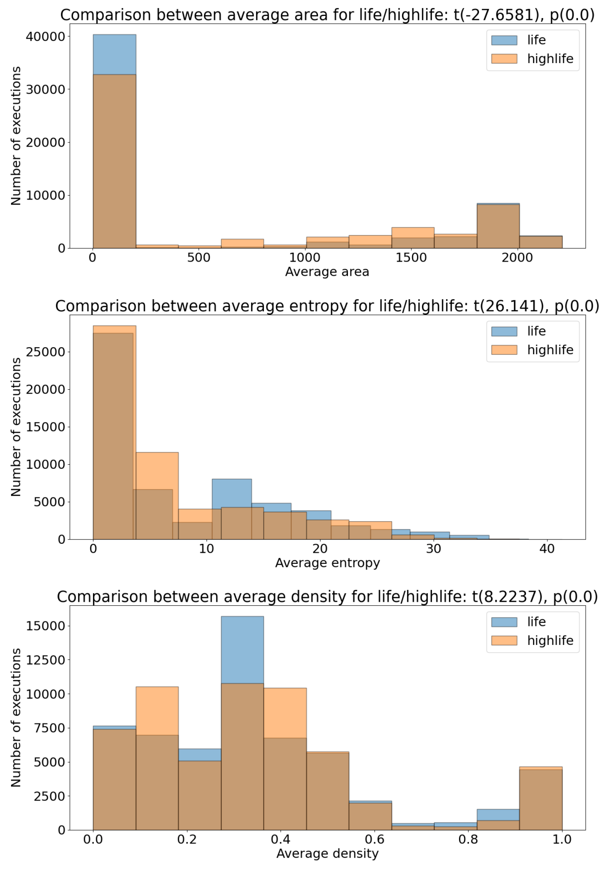


Fig. 13. Comparison between the average area, entropy, and density for life (B3S23) and HighLife (B36S23).

both very similar to each other. The overall shapes of the distributions are nearly identical, but the p-values in each case is very close to 0.000. This means that despite the similarity between the datasets and the closeness in their shapes, a statistical model would still clearly consider them coming from different populations. As such, we reject the null hypothesis.

The final figure to display is a prediction of the 95% confidence interval of a dataset. In our case, we chose to predict the mean of life (B3S23). Ideally, the exhaustive search of the statistics will generalize to larger dataset sizes with more randomization, which is tested in the next set of figures. The distribution of `area` values for an exhaustive search of small initial seeds is given in figure 14.

Clearly, small seed values lead to a very large variance in values. The mean `area` is at around 500, but the 95% confidence interval is approximately in the range [0, 2200]. This could indicate a small dataset size, but it is not overall very helpful in narrowing the search for the true population mean.

Bootstrapping is a helpful technique which can possibly assist us in this goal. With bootstrapping, we repeatedly create sub-samples with replacement from the sample shown in figure 14, then create a new confidence interval for the mean `area` based on it. This distribution is very likely to contain areas

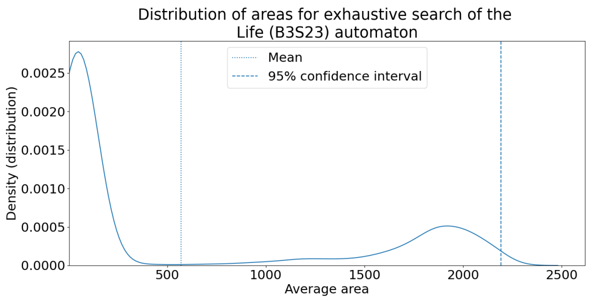


Fig. 14. 95% confidence intervals for the mean of the regular life (B3S23) automaton areas.

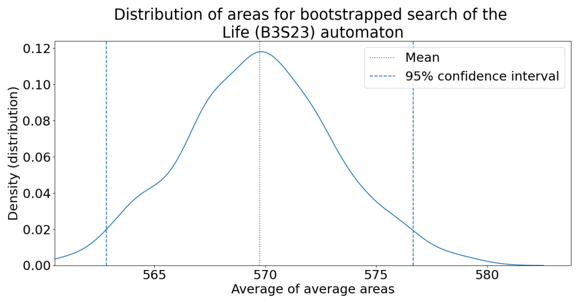


Fig. 15. 95% confidence intervals for the mean of the bootstrapped regular life (B3S23) automaton areas.

close to the grand mean in the original sample, but it should have a variance that somewhat reflects the hidden distribution representing the population. The bootstrapped mean distribution is given in figure 15. The fact this distribution makes a normal-looking curve is promising.

The sample we want to predict the mean of is given in figure 16, representing the average areas of a simulation initialized with a random seed value. Although, we can compare the true mean of that distribution with the 95% confidence generated in figure 15 and see that they are very different from each other. This most likely indicates that the overall tiny range of initialization values from the exhaustive search (between 0 and $2^{4\times4}$) is still far too incapable of predicting the same kind of information shown in the initialization with random seed values (between 0 and $2^{16\times16}$. This does not mean that bootstrapping is an unreliable technique, nor does it imply that

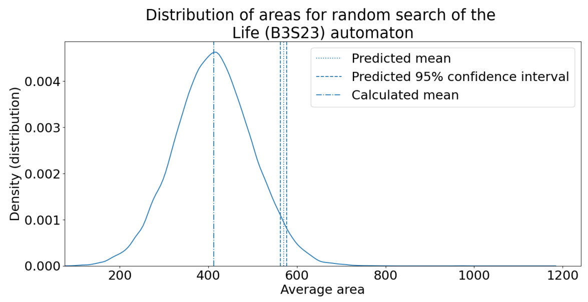


Fig. 16. Distribution of the random life (B3S23) automaton areas with the 95% confidence intervals from figure 15

the variables are completely uncorrelated, but rather that more information should be gathered from a wider distribution of data before it should be used as a predictive tool.

VII. Project Management

This project (besides the `gifenc`, `hashset.c`, and Python statistical libraries included in the source code or project directory) was implemented entirely by Justin Garrigus. Source code can be found at https://github.com/justinmgarrigus/empirical-analysis.git.

References

[1] J. V. Leeuwen and J. Wiedermann, “The turing machine paradigm in contemporary computing,” in *Mathematics Unlimited–2001 and Beyond*, 2001, pp. 1139–1155.

[2] A. M. Turing, “On computable numbers, with an application to the Entscheidungsproblem,” in *Journal of Math*, 1936, pp. 1–36.

[3] C. Bays, “Introduction to cellular automata and Conway’s game of life,” in *Game of Life Cellular Automata*, 2010, pp. 1–7.

[4] M. Gardner, “The fantastic combinations of John Conway’s new solitaire game ‘life’, in *Scientific American*, volume 223, 1970, pp. 120–123.

[5] P. Rendell, “A universal Turing machine in Conway’s game of life,” in *International Conference on High Performance Computing and Simulation*, 2011, pp. 764–772.

[6] J. von Neumann, “Theory of self-reproducing automata,” in *University of Illinois Press*, 1966, pp. 64–73.

[7] , S. Brenner, “Life’s code script,” in *Nature*, 2012, volume 482, pp. 461.

[8] P. P. Choudhury, A. R. Khan, K. Dihidar, and R. Verma, “Text compression using two-dimensional cellular automata,” in *Computers and Mathematics with Applications*, 1999, volume 37, pp. 115-127.

[9] P. L. Rosin, “Image processing using 3-state cellular automata,” in *Computer Vision and Image Understanding*, 2010, volume 114, issue 7, pp. 790-802.

[10] A. Schadschneider and M. Schreckenberg, “Cellular automaton models and traffic flow,” in *Journal of Physics A: Mathematical and General*, 1993, volume 26, pp. 679–683.

[11] S. Wolfram, “A new kind of science,” *Wolfram Media*, 2002.