PSEUDO-RANDOM NUMBER GENERATORS IN THE GAME OF LIFE

Nicole Athanasiou

ABSTRACT

This paper presents a novel Python implementation of John Conway's *Game of Life* and evaluates the performance of the Mersenne Twister and Permuted Congruential generators. *Game of Life* takes place on a 2D matrix with cells that evolve based on their neighbors' states (Shiffman, 2024). Other Python implementations do not include randomized grid dimensions, randomized cell states, the ability to set specific pseudo random number (PRN) generators, and automatic program termination when a pattern starts repeating. Including these features allows users to tailor the simulation to their interests and needs. Additionally, the program can be used to test the difference in means between two PRN generators' simulation output. Specifically, testing the difference in the mean number of generations produced from the Mersenne Twister and PCG-64 shows there is no significant difference. Users of the program can experiment with other PRN generators to see which yields longer runs in *Game of Life*.

1. BACKGROUND & DESCRIPTION OF PROBLEM

The *Game of Life*, created by John Conway, is a cellular automation (Martin, 2024). A cellular automation is a model of a system where cells exist on a grid, are alive or dead, and have neighborhoods (Shiffman, 2024). The state of a cell changes depending on the state of its neighbors. Specifically in Conway's *Game of Life*, patterns should be unpredictable over time.

The grid for *Game of Life* is a 2D matrix with each entry representing a cell (Shiffman, 2024). Cell states are either 0 or 1 for dead or alive, respectively. The rules consider birth, death by overpopulation or loneliness, and stasis -- a dead cell will become alive if it has three living neighbors; an alive cell will die if it has at least four neighbors or if it has less than two living neighbors; otherwise, the cell stays in its current state. The simulation is considered complete when there are no alive cells left or the cells are alternating between the same two patterns.

One consideration in implementing *Game of Life* is the randomness of the starting states. If the goal is to create unpredictable patterns, then it is beneficial for the initial board and cell states to be randomly generated. PRN generators create a sequence of "random" numbers from a function (Kennedy, 2016). The sequence of numbers should be uniformly distributed, have a long period, be reproducible, and generate quickly (Leonelli, 2021). The period is the minimum number of iterations that the function completes before returning to a previous state. Specifying a seed guarantees reproducible results; therefore, the

numbers are *pseudo*-random. This implementation of *Game of Life* leverages PRNs to randomize grid dimensions and cell states, creating more unpredictable patterns than a predetermined board would.

2. MAIN FINDINGS

2.1 Previous Implementations

To reiterate, the problem is to build a grid in which each cell has eight neighbors and each cell's state of dead or alive is updated based on the neighbors' states (Conway's Game of Life, 2024). The cells must first be initialized in the grid. There is a cell at each row and column index (i, j). For each cell, its value will be updated according to its neighbors' states at each time step. Assuming a finite grid, boundaries must be accounted for. Various Python implementations are publicly available and provide a baseline for simulation *Game of Life*.

The implementation on GeeksforGeeks features an N-by-N matrix created using NumPy, "a package for creating and manipulating N-dimensional arrays, performing numerical computing" (Conway's Game of Life, 2024; NumPy, 2024). The board is initialized using numpy.random.choice(), where a higher weight is given to "off" values (0.8) than "on" values (0.2). There are no comments regarding this decision. The *RandomState* class uses the Mersenne Twister pseudo-random number generator (PRN), which boasts a period of $2^{19937} - 1$ (Intel, 2020). However, the *RandomState* class is considered frozen and use of the *Generator* class is recommended (Legacy random generation, 2024). The overall implementation appears to be focused on the animation aspects of *Game of Life*. Measures of performance are not reported and the output is not reproducible. The GeeksforGeeks approach implements the basic rules of *Game of Life* and can be used to inform the basis of any implementation.

An implementation by a data engineer similarly begins by initializing a 2D array, implementing the rules, and iterating through each generation (Kovalchuk, 2024). Rather than randomly initializing the cell states and board, the approach requires a defined 2D array as input. The number of generations to iterate through must be specified. The approach attempts to account for an expanding grid by padding the border with zeros at the initial state. In the final state, the board is cropped around the remaining live cells; no comments are made regarding this decision. Plots and animations are not aspects of this approach. The mean ± standard deviation of 7 runs, 10 loops each run is 44.2 ms ± 1.13 ms per loop. Like the GeeksforGeeks implementation, the Kovalchuk program's unique considerations can be used to inform a new implementation.

2.2 Approach

Similar to previous approaches, NumPy is leveraged in this project to initialize a 2D board of zeros. The grid can be customized by specifying the number of rows and columns, denoted by *m* and *n*, when calling the *game_of_life* function. If the dimensions are not specified, a 10 by 10 board is created. As well, plots can be displayed for each generation if *show_plot* is set to true. The PRN generator can be specified; the NumPy default generator uses O'Neill's permutation congruential generator, PCG-64, with a period of 2^{128} (Permuted congruential generator, 2024). The generator type is instead specified to default to Mersenne Twister due to the higher period, assuming a longer period will yield a greater number of distinct initial states. Next, a PRN is generated for the number of cells to be initialized to the alive state using numpy.random(). Using the number of cells, PRNs are generated for the row and column indices and stored in respective arrays. Each pair of coordinates is set to alive with "1" on the 2D grid. The initial 2D board, created using PRN generators, represents the first generation.

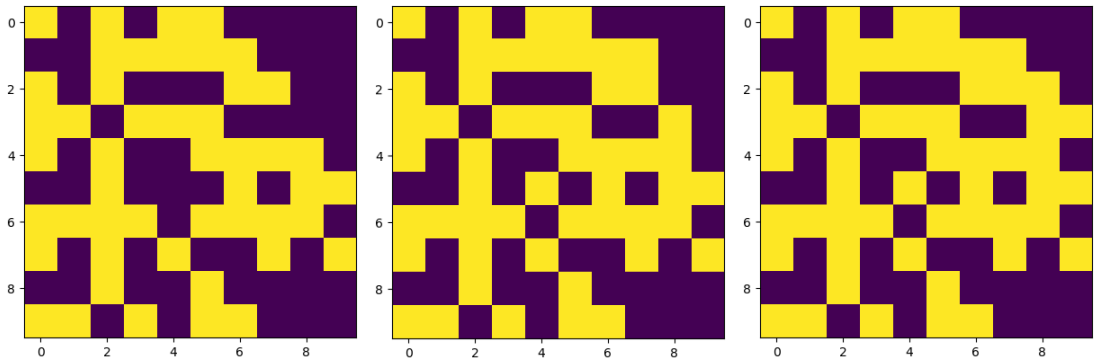

Figure 1: Three iterations/generations from a run of Game of Life. Plots generated from attached code.

After the first generation is defined, the rules of *Game of Life* are applied to update cell states in each generation. The next portion of code loops through the coordinates, counts the number of alive neighbors, and updates the cell based on the rules, accounting for the grid boundaries. A similar method is applied to the dead cells. The last two generations are kept and the former is compared to the current generation. If there are no cells left or the current generation is identical to the generation from two iterations ago, the simulation ends. The number of generations completed and the initial generation grid are returned; these are used to evaluate the performance of the simulation.

The function *eval_game* runs the simulation for a given number of runs and PRN generator and calculates several quantities of interest. The default number of runs is 10,000 and default PRN generator is Mersenne Twister. The grids for each first generation are compared and identical grids are counted. The maximum number of identical starting grids is displayed. Moreover, the average, minimum, maximum,

and median number of generations completed are displayed. The mean ± standard deviation of 7 runs, 10 loops each run is 3.45 ms ± 1.33 ms per loop and 2.21 ms ± 38.7 ms per loop for the Mersenne Twister and PCG-64 generators respectively, which are both faster than the Kovalchuk program. The number of generations the simulation completes is the metric for the PRN generator performance. The intent of the function is to evaluate whether the generators produce different numbers of iterations or runs for *Game of Life*.

2.3 Requirements

The code is run with an IDE, such as Jupyter Notebook, by running all cells. The first cell block imports the necessary packages – packages will need to be installed in order for the code to run successfully. For more information on installing packages, visit the Python packaging guide (Gedam, 2024). The next cell runs the *game_of_life* function, displaying a tuple with a note stating the simulation ended, the number of iterations of the game, and the starting population. The output is convenient for the next cell block, which runs 10,000 games and saves respective number of iterations to a dictionary, then displays summary statistics. As long as the packages are installed, adjusting the code is not necessary for the program to run.

3. CONCLUSIONS

3.1 Results

The *eval_game* function returns summary statistics and can be used to run a t-test. The Mersenne Twister generator has a mean of 10.64, median of 7, and max of 203 iterations, while the PCG-64 generator has a mean of 10.38, median of 7, and max of 219 iterations. Using the output of the *eval_game* function, a t-test for the difference in means between the number of generations for each PRN generator is conducted. The t-statistic is 1.53 and the p-value is 0.12; there is not a significant difference between the mean number of generations of the Mersenne Twister and PCG-64 for their applications to *Game of Life*. Additionally, out of 10,000 runs, neither generator produces a duplicate starting state. Both Mersenne Twister and PCG-64 are sufficient for running *Game of Life* to create unique patterns.

3.2 Future Work

Future work should incorporate dynamic plots and optimize efficiency of the program. One aspect of *Game of Life* that is of interest to many is the patterns that are formed by the simulation (The Game of Life, 2011). While this implementation of *Game of Life* provides an option for displaying static plots for each generation, a dynamic visualization will make observing patterns less tedious and more engaging. Another improvement that could be made is the efficiency of the program in terms of run time.

A single run of the program is not time-consuming; however, running the *eval_game* function with 10,000 runs takes over an hour. One suggestion for reducing run time is to implement a way to compare initial states that does not save each state to a dictionary. Updating the program to include an animation and faster run time will allow for more patterns to be discovered in less time.

REFERENCES

Conway's Game of Life. (2024, 03 08). Retrieved from GeeksForGeeks: https://www.geeksforgeeks.org/conways-game-life-python-implementation/Geeks

Gedam, P. (2024, 11 13). *Installing Packages*. Retrieved from Python Packaging Guide: https://packaging.python.org/en/latest/tutorials/installing-packages/

Intel. (2020, 12 04). *MT19937*. Retrieved from Intel: https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-vector-statistics-notes/2021-1/mt19937.html

Kennedy, T. (2016). *Chapter 3 Pseudo-random numbers generators.* Retrieved from University of Arizona Math Department: https://math.arizona.edu/~tgk/mc/book_chap3.pdf

Kovalchuk, G. (2024, 08 06). *Simulating Conway's Game of Life in Pure Python: A Step-by-Step Guide*. Retrieved from Medium: https://medium.com/@goldengrisha/simulating-conways-game-of-life-in-pure-python-a-step-by-step-guide-859129901348

Legacy random generation. (2024). Retrieved from NumPy: https://numpy.org/doc/stable/reference/random/legacy.html

Leonelli, M. (2021, 04). *Pseudo Random Numbers.* Retrieved from Simulation and Modelling to Understand Change: https://bookdown.org/manuele_leonelli/SimBook/pseudo-random-numbers.html

Martin, E. (2024). *Conway's Game of Life*. Retrieved from playgameoflife.com: https://playgameoflife.com/info

NumPy. (2024). Retrieved from NumPy: https://numpy.org/

Permuted congruential generator. (2024). Retrieved from NumPy: https://numpy.org/doc/stable/reference/random/bit_generators/pcg64.html#numpy.random.PCG64

Shiffman, D. (2024, 09 03). *Chapter 7: Cellular Automata*. Retrieved from The Nature of Code: https://natureofcode.com/cellular-automata/

The Game of Life. (2011). Retrieved from Conway's Life: https://conwaylife.appspot.com/library/