Evolutionary Algorithm on Random Number Generation

LU CHEN, Queen's University, Canada
BAORONG WEI, Queen's Univeristy, Canada
WANQING LI, Queen's University, Canada
YUZHE HE, Queen's Univeristy, Canada

In this work, we use an evolutionary algorithm to find the best rule set for a random number generator based on cellular automata. We evaluate the fitness of the generator's performance using the NIST statistical test suite, which includes a battery of tests for randomness. Our approach produces better generation results compared to prior works that use entropy as a fitness function. Our study demonstrates the effectiveness of using cellular automata and evolutionary algorithms in the design of high-quality random number generators. The code is available on https://github.com/eus-lwq/RandomNumberGeneration.

Additional Key Words and Phrases: Evolutionary algorithm, Genetic algorithm, Cellular Automata, Pseudo Random Generator

ACM Reference Format:
Lu Chen, Baorong Wei, Wanqing Li, and Yuzhe He. 2023. Evolutionary Algorithm on Random Number Generation. In . ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 PROBLEM DESCRIPTION

Pseudo-random number generators (PRNGs) have played a vital role in the development of modern computing, enabling a myriad of applications in various fields. The history of PRNGs traces its roots back to the early 20th century when simple mechanical devices were used to generate seemingly random numbers. Over time, the need for more sophisticated methods led to the development of computational algorithms that could produce sequences of numbers with high randomness.

PRNGs are crucial in numerous domains (i.e. cryptography, statistical simulations, computer graphics, and gaming), PRNG produces sequences of numbers that exhibit properties that mimic true randomness. This pseudo-randomness is essential for applications where unpredictable outcomes are desired, such as in secure communications, Monte Carlo simulations, or procedural content generation in video games.

One of the fundamental aspects of PRNGs is the cycle length, which refers to the number of iterations a generator can produce before repeating the sequence, as shown in Fig.1 (f). A longer cycle length is desirable, as it reduces the likelihood of patterns emerging in the generated numbers, thus enhancing the appearance of randomness. So far, modern generators have boasted periods long enough to render repetition practically undetectable, and the pursuit of improving cycle lengths will still remain at the forefront of computational disciplines.

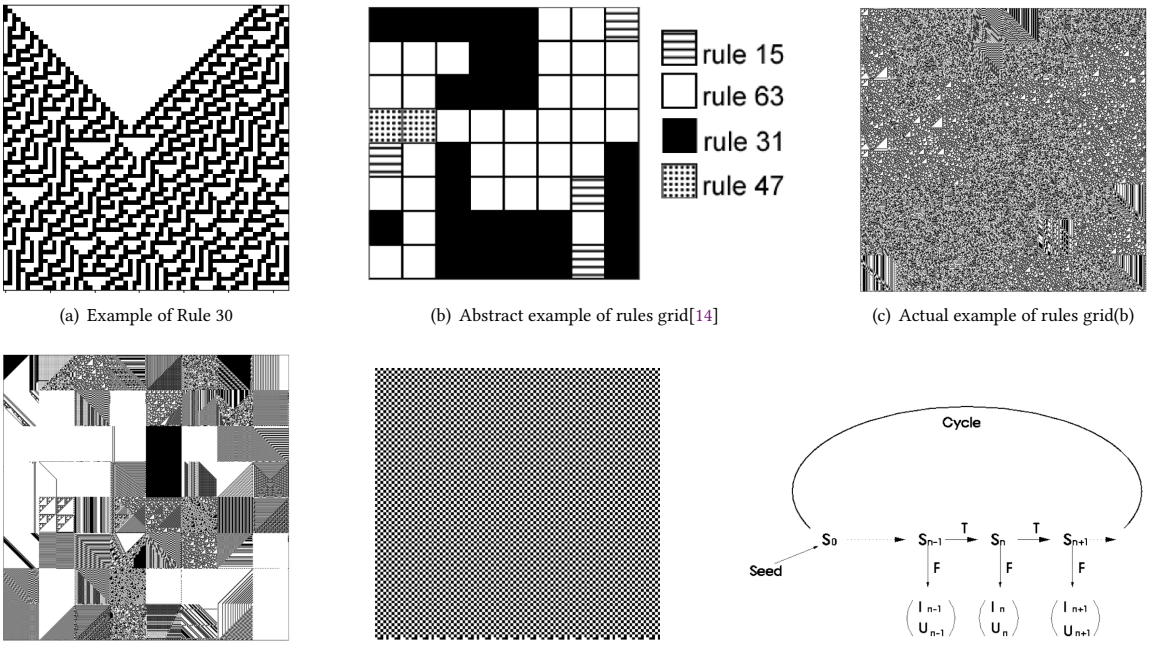
Cellular Automata (CA) are discrete mathematical models that have emerged as a powerful tool for understanding complex systems and generating pseudo-random numbers. CA consists of a grid of cells, each of which can be in one of

Manuscript submitted to ACM

a finite number of states. The cells evolve in discrete time steps according to a set of rules based on the states of their neighbouring cells. These simple, local rules can create intriguing and complex global behaviours.

The connection between CA and pseudo-random number generation lies in their ability to generate intricate patterns and sequences that appear random, even though they are deterministic. Some CA, such as Rule 30, are particularly adept at generating seemingly random sequences and have been employed as PRNGs in various applications.

Stephen Wolfram, a distinguished scientist and the inventor of Mathematica, has made substantial contributions to CA. In his groundbreaking book, "A New Kind of Science," Wolfram delves into the possibilities of CA as a basis for understanding nature and computational complexity. He points to Rule 30[15], shown in Fig.1 (a), as a cellular automaton capable of producing highly unpredictable patterns, emphasizing its potential as a PRNG.



(a) Example of Rule 30 (b) Abstract example of rules grid[14] (c) Actual example of rules grid(b)

(d) Bad example of rules grid (e) noise graph of Bad example(d) (f) Cycle length[4]

Fig. 1. Caption for the entire figure

Using this concept, later research [3] suggested an improved and more secure method for generating random numbers using cellular automata. This method employs a grid, with each cell containing a rule; see Fig.1 (b). Each rule is processed individually and then reassembled into the larger grid, shown in Fig.1 (c).

However, some rule sets have a limited cycle length, see Fig.1 (f), causing the cycle length to regress earlier and making the generator output recognizable by patterns which produce very little randomness, as shown in Fig.1 (d, e). While complex methods have been proposed to increase the cycle length, an alternative approach involves searching for rule sets that can extend the cycle length. The search range, however, is vast: with 64 cells and 255 possible rules per cell, the search space becomes $64^{255} = 2^{1530}$. Exhaustive searching is inefficient because the total number of atoms in

the universe is around 2^{80}[12]. Consequently, a more effective search technique, such as an evolutionary algorithm, is required.

In this case, an Evolutionary Algorithm (EA) is better than exhaustive searching for various reasons. The search space in this problem is immense (2^{1530}), making exhaustive searching computationally infeasible. An EA, on the other hand, is more scalable and can handle large search spaces more efficiently. Additionally, EAs are adaptive and can navigate complex landscapes to find optimal or near-optimal solutions without thoroughly exploring the entire search space. Furthermore, EAs are less likely to get trapped in local optima as they maintain a diverse population of potential solutions, enabling them to explore multiple areas of the search space simultaneously. Overall, using an Evolutionary Algorithm, in this case, is a more effective and efficient method for finding rule sets that can extend the cycle length.

2 LITERATURE REVIEW

PRNGs are widely used in cryptographic applications because the security of these systems depends on the assumption that future values in the random sequence are unpredictable.

Linear Feedback Shift Registers (LFSRs) are widely used binary polynomial generators in PRNGs. LFSRs employ a shift register with binary bits shifted right each clock cycle and a feedback function combining bits using XOR logic. The output bit is often taken from the rightmost bit. The seed value dictates the generated sequence, which appears random and unpredictable but is deterministic and repeatable. The LFSR length and feedback function complexity impact the generated pseudo-random bit stream's quality. Longer registers and complex functions yield longer, harder-to-predict sequences. However, LFSRs may not be secure due to their linearity, making cryptanalysis easier. Researchers utilized GA to improve LFSR, generating an initial population of 16 random LFSRs with eight levels each. Through tournament selection, crossover, and mutation, they developed a novel PRNG with a complex architecture, longer period, and enhanced unpredictability in the generated sequence[9].

CA is another approach that involves cellular automata, initially proposed by Stephen Wolfram to employ rule 30[15] for random number generation. Chowhunrry[3] later suggested representing distinct cells with a grid and combining all created cellular automata into a larger grid of 0s and 1s. Numbers can then be extracted from rows and columns in 64-bit (int size), 128-bit (long size), or larger increments. The state of each cell in the combined grid is updated using the following formula; see Fig. 2's step 3:

$$s_{i,j}(t+1) = X \oplus (C \cdot s_{i,j}(t)) \oplus (N \cdot s_{i-1,j}(t)) \oplus (S \cdot s_{i+1,j}(t) \oplus (E \cdot s_{i,j+1}(t)) \oplus (W \cdot s_{i,j-1}(t)) \tag{1}$$

Here, N (north), S (south), E (east), and W (west) represent the existence of a cell's four neighbours, while C (center) denotes the cell itself. These letters determine whether to include a part of the equation: 0 for exclusion and 1 for inclusion. X can be a binary number, either 1 or 0. For example, rule 14 has a binary representation of 001110, resulting in:

$$s_{i,j}(t+1) = 0 \oplus (0 \cdot s_{i,j}(t)) \oplus (1 \cdot s_{i-1,j}(t)) \oplus (1 \cdot s_{i+1,j}(t) \oplus (1 \cdot s_{i,j+1}(t)) \oplus (0 \cdot s_{i,j-1}(t)) \tag{2}$$

$$= 0 \oplus s_{i-1,j}(t) \oplus s_{i+1,j}(t) \oplus s_{i,j+1}(t) \tag{3}$$

Once the formula is applied to all entries in the large combined grid, the grid's appearance will depend on the quality of the CA rules set. A good set will resemble Fig. 5 (a), while a bad set will look like Fig. 1 (c). Since each entry in the combined CA grid is 0 or 1, rows and columns can be treated as binary streams and converted to integers.

Nonetheless, this approach faces the challenge of dealing with a bad rule set, which can result in a short cycle length and negatively impact the quality of generated random numbers. To address this issue, Tomassini[14] suggested that

the Evolutionary Algorithm (EA) could be employed as an effective search method to identify rule sets with sufficient randomness for random number generation.

3 EA DESIGN

Because of the high computing demands and memory usage, we chose a shorter cycle length of 10 and performed five tests for the various methods discussed below. Running these tests was challenging, as the extremely high memory consumption frequently led to program shutdowns or took an exceptionally long time to complete (over 30 hours, exceeding the limited 24-hour session of Colab). The most suitable machine we discovered was the Colab with TPU runtime, offering 40 CPU cores. For more information, please refer to Appendix C.

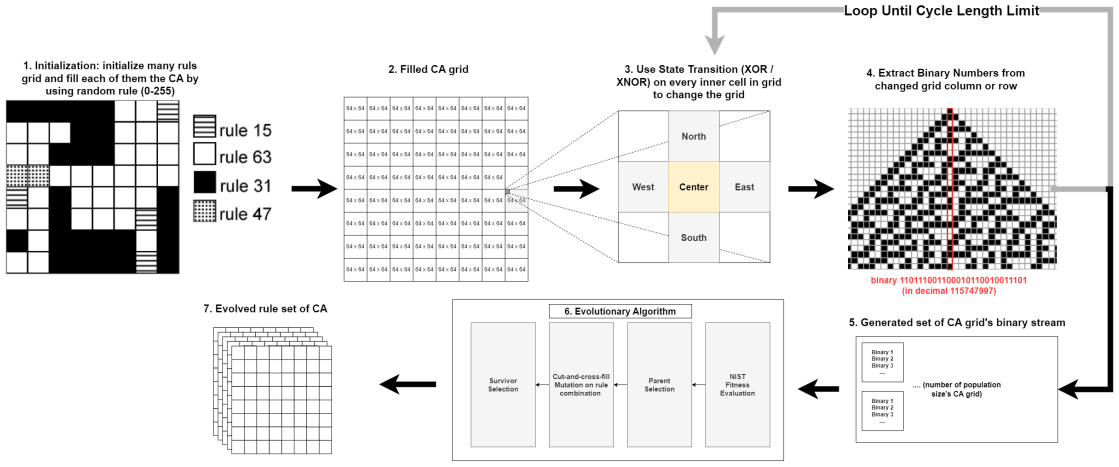


Fig. 2. Flow chart of evolution process, step 1 figure from [13]

Representation: We utilize an 8x8 CA matrix as depicted in Figure 2 step 1 and Figure 1 (b), where each matrix cell corresponds to a rule. These rules are later expanded to a 64x64 CA grid in their respective positions, as shown in Figure 2 step 2. Initially, the filled CA grid is in state 0, displaying a distinct pattern. To proceed, we perform state transitions using the formula (1) from the previous section, applying it to every cell in the filled CA grid, as demonstrated in Figure 2 steps 2 and 3.

Initialization: During initialization, we randomly fill an 8x8 matrix with numbers ranging from 0 to 255 for each cell. At this stage, the rule set grid is not expanded to the CA grid due to excessive memory requirements. The expansion occurs only during fitness evaluation, which improves the program's efficiency.

Fitness: Instead of employing traditional Diehard [7] tests from earlier implementations, we used the 2010 NIST [11] statistical tests to measure fitness. The NIST tests are superior and more advanced than the 1995 Diehard tests, now considered outdated and simplistic. The NIST (National Institute of Standards and Technology) statistical tests are a collection of tests designed to evaluate the randomness of a bit sequence, such as those generated by a cryptographic algorithm, making them well-suited for assessing our generator's fitness. NIST tests comprise 15 distinct tests (calculation methods in appendix). Each NIST test produces a p-value; if a test returns a p-value > 0.01, it indicates the generator has passed that test. We calculate the fitness using the formula (number of tests passed / total number of tests), with a maximum fitness score of 1.

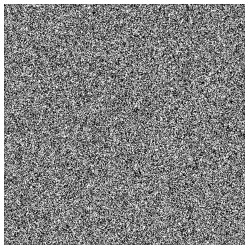
Variation Operator: The variation operator includes both parent selection and survival selection. For parent selection, we used *MPS*, *tournament*, and *random uniform*; for survivor selection, we employed *mu plus lambda*, *replacement*, and *random uniform*. Given the vast search space and the significant computational demands of the fitness component, our goal is to achieve faster convergence while obtaining high-quality results. We provide a more detailed discussion in the Comparison section.

4 EA RESULT

In Figure 3 (a) and (c), we present two exemplary cases of the best evolved cellular automata (CA) grids obtained using the aforementioned method. Both cases have successfully passed all the NIST tests with a cycle length of 10, indicating a fitness score of 1. Concurrently, we employed a noise generator to produce the CA randomness depicted in Figure 3 (b) and (d). It is important to note that the more discernible the pattern, the lower the quality of randomness (e.g., Figure 1 (c) and (d) exhibit recognizable patterns in the noise graph, thus implying poor randomness for these two CAs). Conversely, the absence of any discernible patterns in the noise graph signifies superior randomness. As a result, the high degree of randomness in Figure 3 (b) and (d) is evident, given that no recognizable patterns are discernible.

31	156	157	7	76	168	91	124
217	91	144	115	251	217	7	56
38	180	163	167	163	180	159	149
113	77	141	91	141	118	77	143
159	71	74	74	194	194	211	197
163	235	107	152	163	27	206	191
44	15	78	10	78	10	225	218
165	181	238	102	137	181	74	201

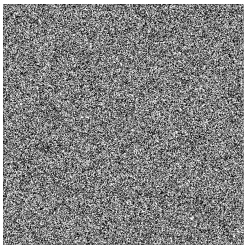
(a) rules of gird1



(b) noise graph of grid1

28	232	201	48	41	170	253	173
211	245	76	102	38	204	181	123
114	170	132	182	178	107	25	49
225	236	195	120	215	117	9	22
136	165	24	228	167	90	3	60
125	5	199	49	156	2	87	20
212	252	27	226	74	198	71	73
224	75	133	134	180	95	15	108

(c) rules of grid2



(d) noise graph of grid2

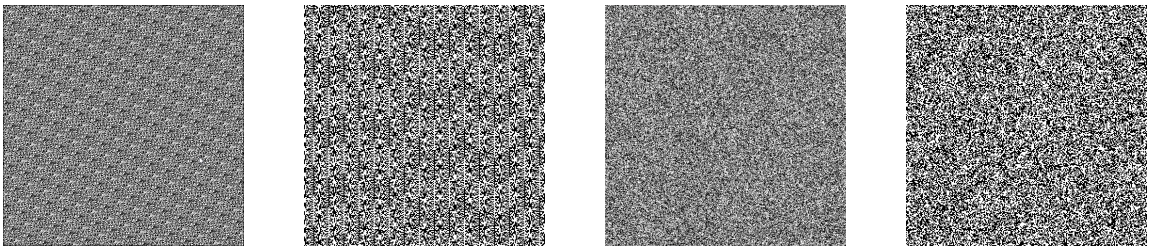
Fig. 3. Best evolved CAs grid example

5 COMPARISON

Compared to previous work [13], which achieved a fitness score of 0.93 for their 8x8 CA in NIST tests using our measurement method, our study yielded better results by consistently testing various combinations of evolutionary algorithm (EA) parameters. Many CAs identified through this approach achieved a fitness score of 1 (10 cycles). This improvement can be attributed to using the latest NIST testing method to assess randomness rather than relying on entropy measurements.

Additionally, we evaluated our proposed algorithm against two widely-used random number generation methods: Linear Congruential Generator (LCG) [6] and PHP rand() [8]. Examining the noise graphs produced by these algorithms revealed discernible patterns in both Figure 5 (a) and (b), implying their predictability and unreliability. Using such algorithms in safety or cryptography applications could introduce significant security risks. In contrast, our proposed algorithm displayed a highly unpredictable pattern in Figure 5 (c), comparable to true randomness seen in Figure 5 (d), where both lack recognizable patterns. This suggests our algorithm surpasses LCG and PHP rand() in terms of randomness, providing enhanced security and unpredictability.

We compared the performance from three different metrics of 9 combinations of variation operator, as shown in Fig 5. The method success rate is calculated by (number of EA highest fitness > 0.99) / (number of unique CA). And the



(a) LCG Randomness Noise [1] (b) PHP Random Noise [10] (c) Our Best CA Noise (d) True randomness from atmospheric noise [10]

Fig. 4. Random Performance Comparison

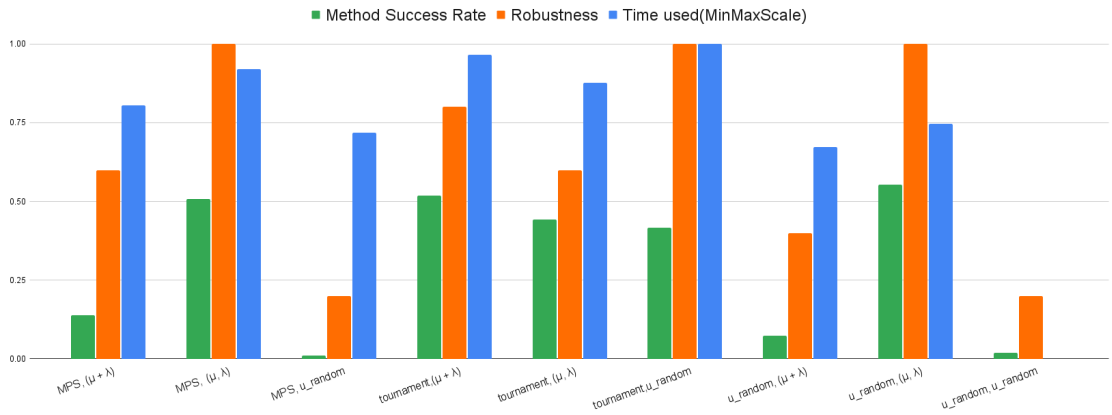
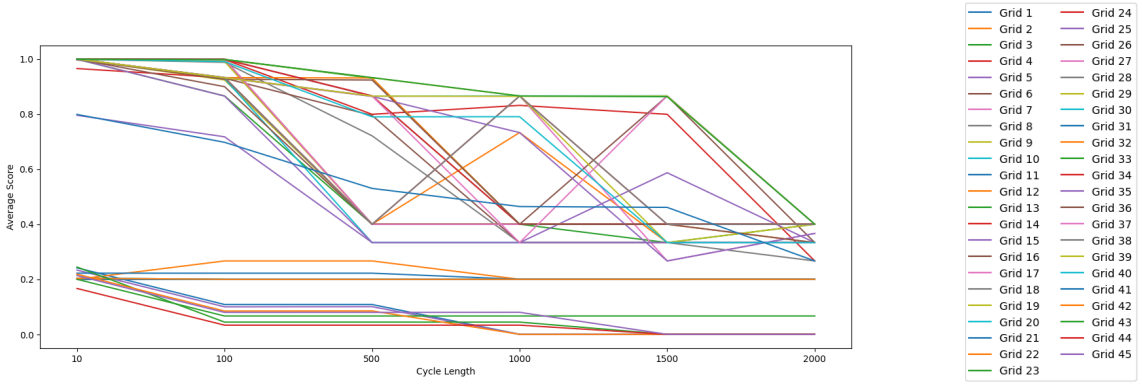


Fig. 5. Different EA variation operator performance comparison

Robustness is the average score to run five times, the Timeused is the speed to run the EA in minutes. These three metrics was normalized between 0 to 1 for better visualization. We aim to rapidly find high-quality results in the vast 2^{1530} search space. Combining all measures, using random uniform parent selection and replacement selection for survivor selection yielded the highest method success rate (0.55), capable of producing diverse solutions. This combination also demonstrated 100% robustness, indicating the results were not obtained by chance. Among methods with similar success rates and robustness, this combination took less time (30.29 minutes).

We conducted cycle length tests on all rule grids obtained from the EA with a cycle length of 10. These grids were tested at cycle lengths of 10 (original), 100, 500, 1000, 1500, and 2000. This step was highly computationally intensive and often exhausted RAM, causing program termination. We used techniques to parallelize the program and limit the number of open processes, eventually collecting the required data. As depicted in Fig 6, the grids were not highly tenacious against increasing cycle lengths, with overall scores decreasing. However, some grids like grid-30, grid-39, and grid-43 maintained fitness scores above 0.8 at 1000 cycle length. Notably, these grids, found by EA at a cycle length of 10, remained robust with 100 times larger cycle lengths.



(a) increasing Cycle length test

(b) legend of each grid

Fig. 6. Performance under different Cycle length, A better graph is in Appendix B

To discover more tenacious grids, the only option is to set a higher cycle length, such as 5000 or more, during the EA stage. In the paper [14], the author evaluated each CA grid's fitness using a 4096 cycle length. However, this was difficult for us to achieve due to limited computational resources and the inability to use GPU(CUDA programming). For more information, refer to Appendix C, and also you can visit our GitHub page to find out what rules each grid uses.

6 DISCUSSION

Throughout the development process, computational bottlenecks were a persistent challenge. We made numerous behind-the-scenes efforts, such as finding suitable cloud services with more CPU cores (Kaggle, Colab), and shifting computationally intensive tasks, like filling the actual CA grids and fitness evaluation, from Python to C/C++. This successfully reduced the completion time for an EA process from 200 hours to 30 minutes. However, these efforts did not completely solve the problem, as increasing the cycle length continued to pose challenges to RAM and parallel design.

6.1 Main Finding

The main goal in designing an EA for a problem like this is to achieve faster convergence. We tested nine combinations of classical selection methods and concluded that using at least one random selection, either in parent or survivor selection, can speed up the EA process while maintaining good quality outcomes. In the vast CA search space (2^{1530}), CA rule sets can be classified based on their ability to withstand increased cycle lengths while maintaining a high fitness score (at least greater than 0.9). The higher this ability, the more computational resources are required for EA to find such a rule set. Therefore, when designing a CAPRNG based on this method, it is essential to identify all rule sets that possess two important merits: producing sufficient randomness and resisting cycle length growth. However, it is important to note that the EA can provide a solution that resists the cycle length set during the EA process, but it doesn't guarantee that these solutions will perform well in cycle lengths higher than the one used during the EA process.

6.2 Future Work

Performance is a top priority when working on a task like this. Due to our inability to use CUDA for GPU utilization, we place all the computational pressure on the CPU, while GPU usage remains at 0. However, the CPU is not optimal for handling such tasks. We discovered that some tests [5] can actually be parallelized, as well as generating the CA [2]. Focusing on developing CUDA versions of these algorithms in the future can significantly improve overall efficiency, speeding up the EA process, allowing for more accurate and precise testing, and providing a more comprehensive conclusion on the performance of CAPRNG. By leveraging parallelization, we can better explore the potential of CAPRNG and optimize its performance.

REFERENCES

[1] AmiditeX. 2023. RandomBitmapGenerator. https://github.com/AmiditeX/RandomBitmapGenerator.

[2] Daniel Cagigas-Muñiz, Fernando Diaz-del Rio, Jose Luis Sevillano-Ramos, and Jose-Luis Guisado-Lizar. 2022. Efficient simulation execution of cellular automata on GPU. *Simulation Modelling Practice and Theory* 118 (July 2022), 102519.

[3] Sanjoy Chowdhury and Bidyut B Chaudhuri. 1990. A class of two-dimensional cellular automata and their applications in random pattern testing. *Journal of Electronic Testing* 1, 1 (1990), 8–11. https://doi.org/10.1007/BF00971964

[4] Illinoi. 2016. Notes on Pseudo-Random Number Generation. https://courses.physics.illinois.edu/phys466/fa2016/lnotes/PRNG.pdf. Accessed: April 11, 2023.

[5] HyungGyoon Kim, Hyungmin Cho, and Changwoo Pyo. 2019. GPU-based acceleration of the Linear Complexity Test for random number generator testing. *J. Parallel and Distrib. Comput.* 128 (June 2019), 115–125.

[6] D. H. Lehmer. 1949. Mathematical methods in large-scale computing units. In *Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery*. Harvard University Press, Cambridge, MA, 141–146.

[7] George Marsaglia. 1996. DIEHARD: a battery of tests of randomness. http://stat.fsu.edu/ geo/.

[8] PHP. 2023. PHP: rand - Manual. https://www.php.net/manual/en/function.rand.php.

[9] Alireza Poorghanad, Ali Sadr, and Alireza Kashanipour. 2008. Generating High Quality Pseudo Random Number Using Evolutionary methods. In *2008 International Conference on Computational Intelligence and Security*, Vol. 1. IEEE, 331–335. https://doi.org/10.1109/CIS.2008.220

[10] RANDOM.ORG. 2023. Analysis of RANDOM.ORG. https://www.random.org/analysis/.

[11] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. 2010. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. *NIST Special Publication* 800-22 (2010).

[12] Vishal Thaku. 2022. How Many Atoms Are There In The Universe? https://www.scienceabc.com/nature/universe/how-many-atoms-are-there-in-the-universe.html.

[13] Marco Tomassini, Moshe Sipper, and Mathieu Perrenoud. 1996. On the generation of high-quality random numbers by two-dimensional cellular automata. *IEEE Trans. Comput.* 45, 10 (1996), 1176–1185.

[14] M. Tomassini, M. Sipper, and M. Perrenoud. 2000. On the generation of high-quality random numbers by two-dimensional cellular automata. https://ieeexplore.ieee.org/document/888056. *IEEE Trans. Comput.* 49, 10 (October 2000), 1146–1151. https://doi.org/10.1109/12.888056

[15] Stephen Wolfram. 1986. Random sequence generation by cellular automata. *Journal of Mathematical Psychology* 30, 2 (1986), 141–168. https://doi.org/10.1016/0022-2496(86)90028-X

A APPENDIX: NIST TEST SUITES

(1) **Frequency Test**: This test checks whether the proportion of zeros and ones in the sequence is approximately equal. The formula for the Frequency Test, also known as the Chi-Squared Test, is given by:

$$\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$$

where:

- χ^2 is the test statistic
- O_i is the observed frequency in the i-th category
- E_i is the expected frequency in the i-th category
- k is the number of categories

(2) **Block Frequency Test**: This test divides the sequence into non-overlapping blocks and checks whether the proportion of zeros and ones in each block is approximately equal.

$$\chi^2 = 4N \sum_{i=1}^{M} \frac{(f_i - \frac{N}{2})^2}{N}$$

where:

- χ^2 is the test statistic
- N is the length of the bit sequence
- M is the number of blocks
- f_i is the number of ones in the i-th block

(3) **Cumulative Sums Test**: This test checks whether the cumulative sum of the deviations of the sequence from its mean is within certain bounds. The Cumulative Sums Test is used to test if a sequence of numbers is random or not. The test involves calculating the cumulative sum of the deviations of the numbers from their mean, and then checking if the sum exceeds a certain threshold. The formula for the test statistic is:

$$C_n = \max_{1 \le i \le n} \left| \sum_{j=1}^{i} (X_j - \bar{X}) \right|$$

where:

- C_n is the test statistic
- n is the length of the sequence
- X_j is the j-th number in the sequence
- $\bar{X}$ is the mean of the sequence

(4) **Runs Test**: This test checks whether the oscillation of consecutive zeros and ones. The test statistic R is defined as:

$$R = \sum_{i=1}^{n-1} I(x_i < x_{i+1})$$

where x_i is the ith observation in the sequence, n is the sample size, and $I(\cdot)$ is the indicator function that takes the value 1 if the argument is true, and 0 otherwise. The expected value of R is given by:

$$E(R) = \frac{(n-1)}{2}$$

and its variance is given by:

$$\mathrm{Var}(R) = \frac{(n-1)(n-2)}{12}$$

(5) **Longest Run of Ones Test**: This test checks whether the length of the longest run of consecutive ones in the sequence is within certain bounds.

$$V = \max_{i=1,2,\ldots,n} N_i$$

Let n be the sample size, and let N_i be the number of runs of i consecutive ones in the sequence, for $i = 1, 2, \ldots, n$. The test statistic V is defined as the above formula. Under the null hypothesis, the expected value of V is given by:

$$E(V) = \frac{2n}{3} + \frac{1}{2}$$

and its variance is given by:

$$\mathrm{Var}(V) = \frac{16n - 29}{90}$$

(6) **Rank Test**: This test checks whether the rank of each submatrix of a matrix derived from the sequence is within certain bounds.

$$T = \mathrm{sgn}\left(\sum_{i=1}^{n} r_i - \frac{n(n+1)}{2}\right) \cdot \min\left(\sum_{i=1}^{n} |r_i - i + 1|, \sum_{i=1}^{n} |r_i - i|\right)$$

where $\mathrm{sgn}(x)$ is the sign function, defined as $\mathrm{sgn}(x) = 1$ if $x > 0$, $\mathrm{sgn}(x) = -1$ if $x < 0$, and $\mathrm{sgn}(x) = 0$ if $x = 0$. Under the null hypothesis of randomness, the expected value of T is given by:

$$E(T) = \frac{n(n+1)}{4}$$

and its variance is given by:

$$\mathrm{Var}(T) = \frac{n(n-1)(2n+1)}{24}$$

(7) **Discrete Fourier Transform Test**: This test checks whether the power spectrum of the sequence follows a certain distribution.

$$S = \frac{2}{n}\sum_{k=0}^{n-1} |\hat{x}_k|^2$$

Under the null hypothesis of randomness, S follows a chi-squared distribution with 2 degrees of freedom. The test is conducted by comparing S with the critical value of the chi-squared distribution at a specified significance level.

The discrete Fourier transform of the data can be calculated using the fast Fourier transform (FFT) algorithm. Let $\omega_n = \exp(-2\pi i/n)$ be the nth root of unity, and let X_k be the discrete Fourier transform of the data. Then:

$$X_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk}$$

for $k = 0, 1, \ldots, n-1$.

(8) **Nonperiodic Template Matching Test**: This test checks whether the sequence contains any non-overlapping patterns that match a given template.

$$C = \max_{1 \le i \le n-m+1} \sum_{j=1}^{m} I(x_{i+j-1} = t_j)$$

where $I(\cdot)$ is the indicator function that takes the value 1 if the argument is true, and 0 otherwise.

Under the null hypothesis of randomness, the expected value and variance of C are given by:

$$E(C) = \frac{m(n-m+1)}{2^n}$$
$$\mathrm{Var}(C) = \frac{m(n-m+1)(2m-2n+3)}{2^{2n}}$$

The template pattern T can be chosen to test for the presence of a specific subsequence or motif in the data. The test can be repeated for multiple template patterns to detect multiple motifs.

(9) **Overlapping Template Matching Test**: This test checks whether the sequence contains any overlapping patterns that match a given template. Let $x_1, x_2, \ldots, x_n$ be the observed data, and let $T = t_1, t_2, \ldots, t_m$ be the template pattern of length m. The test statistic C is defined as:

$$C = \sum_{i=1}^{n-m+1} W(x_i, x_{i+1}, \ldots, x_{i+m-1})$$

where $W(\cdot)$ is a weighting function that assigns a weight of 1 to the occurrences of the template pattern, and a weight of 0 otherwise.

Under the null hypothesis of randomness, the expected value and variance of C are given by:

$$E(C) = \frac{n-m+1}{2^m}$$
$$\mathrm{Var}(C) = \frac{n-m+1}{2^{2m}}\left(\frac{4^m - 2^{2m}}{3}\right)$$

(10) **Universal Statistical Test**:

Require: A binary sequence $x_1, x_2, \ldots, x_n$, and the length of the blocks m.
Ensure: A p-value that measures the randomness of the sequence.
function MaurerTest($x_1, x_2, \ldots, x_n, m$)
Initialize b and j to 0.
Divide the sequence into blocks of length m, and compute their frequencies.
for $i = 1$ to $\min(100, n/m)$ **do**
Choose a random subset of the blocks and compute their frequencies.
Compute the test statistic T_i as the difference between the two sets of frequencies.
if $T_i > b$ **then**
Set $b \leftarrow T_i$, and $j \leftarrow i$.
end if
end for
if $b \ge 0$ **then**
Compute the p-value p as the probability that a random sequence would have a larger value of T_j.
else
Compute the p-value p as the probability that a random sequence would have a smaller value of T_j.
end if
return p.
end function

(11) **Approximate Entropy Test**: This test checks whether the sequence is significantly less predictable than a random sequence of the same length.

$$C_m = \frac{1}{n-m+1}\sum_{i=1}^{n-m+1} \ln\left(\frac{\hat{C}_m(i)}{\hat{C}_{m+1}(i)}\right)$$

where $\hat{C}_m(i)$ is the conditional probability that a sequence of length m matches a given pattern in the data, and $\hat{C}_{m+1}(i)$ is the conditional probability that a sequence of length $m+1$ matches the same pattern. The logarithmic term in the formula measures the logarithmic difference in probabilities between the two cases.

Under the null hypothesis of randomness, the expected value of C_m is approximate:

$$E(C_m) \approx -\ln(2)$$

and its variance is approximate:

$$\mathrm{Var}(C_m) \approx \frac{2}{n}$$

The template pattern length m can be chosen to test for the complexity of different aspects of the data. The test can be repeated for multiple values of m to detect multiple levels of complexity.

(12) **Random Excursions Test**: This test checks whether the sequence exhibits certain patterns of excursions from the mean.
Under the null hypothesis of randomness, the expected value of X_k is given by:

$$E(X_k) = \begin{cases} \frac{n-k+3}{2^{k+2}}, & k < n \\ 1 - \frac{3}{2^{n+1}}, & k = n \end{cases}$$

and its variance is given by:

$$\mathrm{Var}(X_k) = \begin{cases} \frac{n-k+3}{2^{k+2}}\left(1 - \frac{n-k+3}{2^{k+2}}\right), & k < n \\ \frac{3}{2^{n+1}}\left(1 - \frac{3}{2^{n+1}}\right), & k = n \end{cases}$$

The test is repeated for values of k between 1 and n, where n is the length of the sequence. The overall result of the test is determined by examining the p-values of the individual tests.

(13) **Random Excursions Variant Test**: This test is a modified version of the Random Excursions Test.
Under the null hypothesis of randomness, the expected value of X_k is given by:

$$E(X_k) = \begin{cases} \frac{n-k+3}{2^{k+2}}, & k < n \\ 1 - \frac{3}{2^{n+1}}, & k = n \end{cases}$$

and its variance is given by:

$$\mathrm{Var}(X_k) = \begin{cases} \frac{n-k+3}{2^{k+2}}\left(1 - \frac{n-k+3}{2^{k+2}}\right), & k < n \\ \frac{3}{2^{n+1}}\left(1 - \frac{3}{2^{n+1}}\right), & k = n \end{cases}$$

The test is repeated for values of k between 1 and n, where n is the length of the sequence. The overall result of the test is determined by examining the p-values of the individual tests.

(14) **Serial Test**: This test checks whether the sequence contains any overlapping patterns of a given length.
Let $x_1, x_2, \ldots, x_n$ be the observed data, and let m be the length of the serial patterns to be tested. The test statistic S_m is defined as:

$$S_m = \frac{(n - m + 1)(\hat{c}_1 - \hat{c}_2)}{\hat{s}}$$

where $\hat{c}_1$ is the observed frequency of occurrence of the serial pattern, $\hat{c}_2$ is the expected frequency of occurrence of the serial pattern under the assumption of independence, and $\hat{s}$ is the estimated standard deviation of the number of occurrences of the pattern.
Under the null hypothesis of independence, the expected value of S_m is zero, and its variance is approximate:

$$\mathrm{Var}(S_m) \approx 2(n - m + 1)\left(\frac{1}{2^m} - \frac{1}{2^{2m}}\right)$$

The test is repeated for different values of m, and the overall result of the test is determined by examining the p-values of the individual tests.

(15) **Linear Complexity Test**: This test checks whether the linear complexity of the sequence is within certain bounds.
Under the null hypothesis of randomness, the expected value of T_n is approximate:

$$E(T_n) \approx \frac{n}{2}\left(1 - \frac{1}{2p}\right)$$

and its variance is approximate:

$$\mathrm{Var}(T_n) \approx \frac{n}{4p}$$

B APPENDIX: LARGER GRAPH

Following is the heat Map of the degradation of each rule set while increasing of the cycle length, the lighter the colour, the higher the score will be.

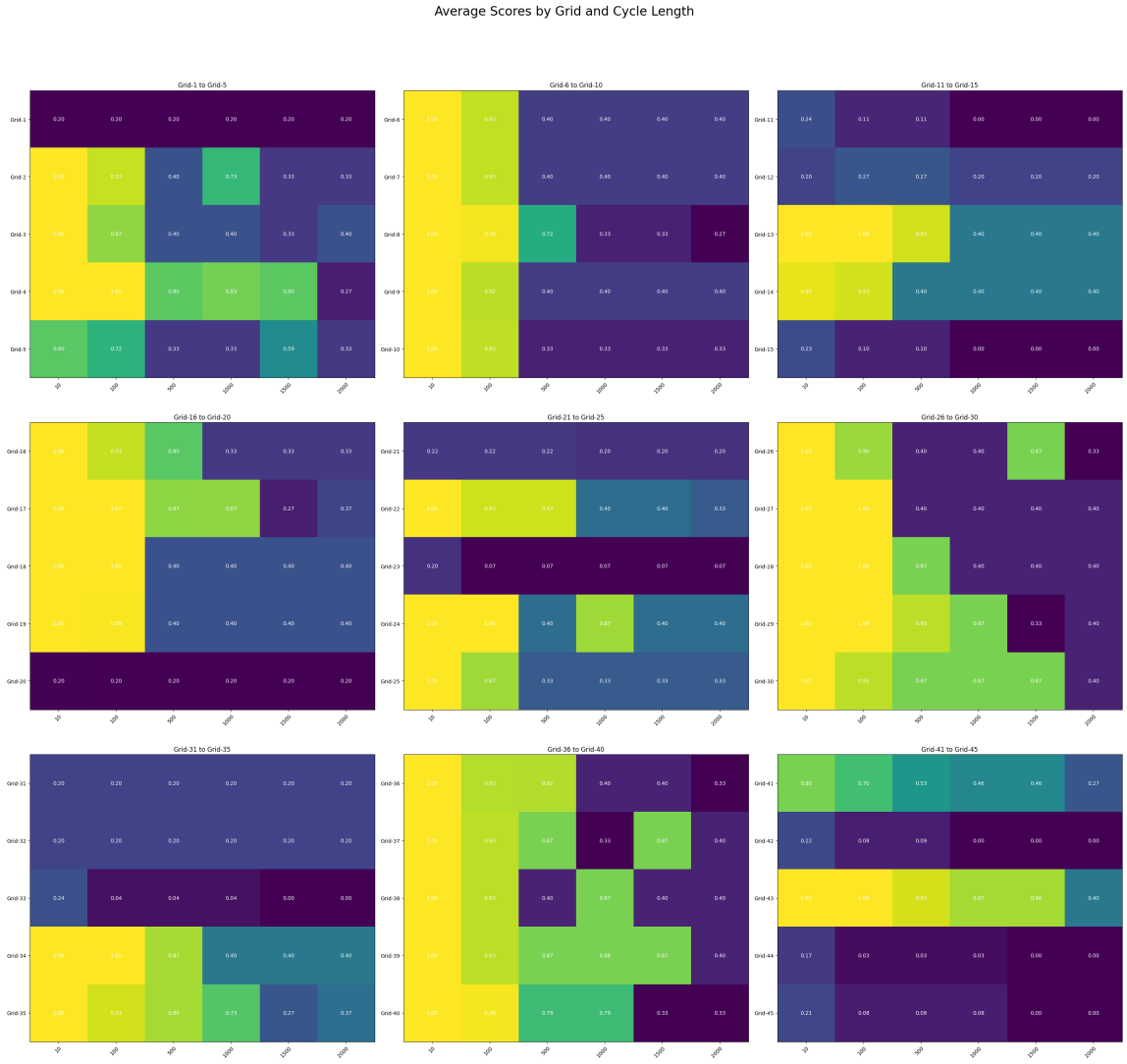


Fig. 7. Heat Map of the cycle length testing

C APPENDIX: MORE INFORMATION ON MACHINE EFFICIENCY

The status of using CPU and the status of using RAM The status of using RAM, a program takes 21g in the ram

Received 11 April 2023; revised 22 April 2023; accepted 23 April 2023

```
Terminal ×
top - 13:18:40 up 40 min,  1 user,  load average: 134.15, 128.89, 112.10
Tasks: 127 total,  41 running,  83 sleeping,   0 stopped,   3 zombie
%Cpu0  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  : 99.3 us,  0.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  : 99.3 us,  0.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu8  : 99.3 us,  0.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu9  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu10 : 99.3 us,  0.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu11 : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu12 : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu13 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu14 : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu15 : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu16 : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu17 : 99.0 us,  0.7 sy,  0.0 ni,  0.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu18 : 99.3 us,  0.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu19 : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu20 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu21 : 97.7 us,  2.0 sy,  0.0 ni,  0.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu22 : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu23 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu24 : 99.3 us,  0.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu25 : 99.3 us,  0.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu26 : 99.3 us,  0.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu27 : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu28 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu29 : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu30 : 98.3 us,  1.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu31 : 99.0 us,  1.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu32 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu33 : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu34 : 99.7 us,  0.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu35 : 99.0 us,  1.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu36 : 98.7 us,  1.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu37 : 98.7 us,  1.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu38 : 99.3 us,  0.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu39 :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  36085.9 total,  27212.3 free,   3478.9 used,   5394.7 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  32125.6 avail Mem
```

Fig. 8. CPU usage status

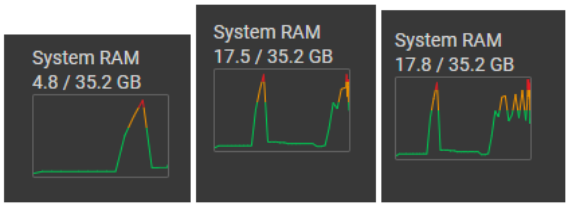


Fig. 9. RAM usage

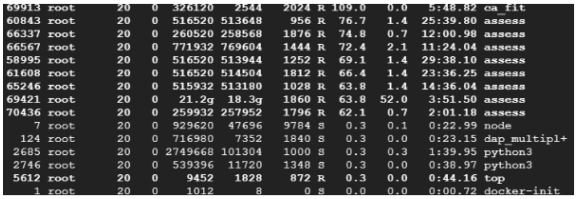


Fig. 10. Memory usage status