Int. j. inf. tecnol.
https://doi.org/10.1007/s41870-019-00307-9

ORIGINAL RESEARCH

Pseudo-random number generators based on the Collatz conjecture

David Xu[1] · **Dan E. Tamir**[2]

Received: 22 November 2018 / Accepted: 8 April 2019

Abstract In this research, we have studied the applicability of the Collatz Conjecture to pseudo-random number generators (PRNG). The research was motivated by the simplicity of the Collatz function, which makes it attractive as a potential PRNG. We have experimented with several candidate PRNGs based on the trajectory property of the Collatz function and the Collatz graph. The NIST Test Suite (SP 800-22) was used to evaluate the statistical randomness of the output of our PRNGs. In addition, we utilized a method to rank each PRNG by quality of random output. The test results have demonstrated that two of our PRNGs pass the NIST Test Suite, and that there is no significant statistical difference between the outputs of our PRNGs to that of the Mersenne Twister, the built-in PRNG in Python 3.7. To the best of our knowledge, our algorithms are the first to successfully utilize properties of the Collatz function to generate random numbers. Additionally, we have proved that one of our PRNGs generates uniformly distributed output with a period of 2^{32}. Finally, we have found that two of our PRNGs perform on par with the Mersenne Twister algorithm. Because our algorithms pass the NIST Test Suite, they are suitable for usage in certain cryptographic applications as well as simulations.

Keywords Pseudo-random number generator · Collatz conjecture · Statistical testing

✉ Dan E. Tamir
dan.tamir@txstate.edu

David Xu
david.duanmu.xu@gmail.com

[1] Westwood High School, Austin, TX, USA

[2] Texas State University, San Marcos, TX, USA

1 Introduction

Randomness is a key component in areas such as statistical testing, simulations, and the generation of encryption keys for cryptographic purposes [4].

Random number generators can be divided into two categories: pseudo-random number generators (PRNGs) and true random number generators (TRNGs). TRNGs rely on physical processes, such as atmospheric noise or radioactive decay, to generate numbers. However, sources for TRNGs are not only hard to find, but also typically too inefficient to generate sufficient quantities of random numbers in a short time. This makes TRNGs less suitable for applications such as simulations, which require high throughput of random numbers. Additionally, TRNGs do not provide the ability to produce the same sequence of random numbers, which is often required in simulations and encryption applications.

PRNGs are algorithms that generate sequences of numbers with similar statistical properties to those from a TRNG. An example of a PRNG is the Mersenne Twister [10]. However, PRNGs are not truly random because they are deterministic and periodic.

PRNG algorithms may be evaluated through batteries of statistical tests to check for non-random behavior and suitability for cryptography. In this research, we utilized the National Institute of Science and Technology (NIST) Test Suite (SP 800-22), which was used in the selection of Advanced Encryption Standard (AES), to evaluate the effectiveness of our proposed PRNGs [5, 11]. Passing the NIST Test Suite indicates that the output from the PRNG is statistically similar to output from a true source of randomness. The NIST Test Suite contains 15 statistical tests, namely, the Frequency, Block Frequency, Runs, Longest Runs, Binary Matrix Rank, Spectral, Non-Overlapping/

Published online: 22 April 2019

Overlapping Template Matching, Universal Statistical, Linear Complexity, Serial, Approximate Entropy, Cumulative Sums, Random Excursions, and Random Excursions Variant tests. Each test focuses on detecting a specific type of non-random behavior, such as a non-uniform distribution of zeroes and ones in bit-strings. A detailed description of the test suite and each statistical test can be found in [11]. It should be noted, however, that the NIST tests constitute a pass-fail result and cannot be used to rank PRNGs by quality or suitability for specific applications.

In this paper, we experimented with different PRNGs in order to check for random behavior in the trajectory property of the Collatz function and the Collatz graph. We have proved that one of our algorithms based on the trajectory property produces random output with a period of 2^{32}. Using the results from the NIST Test Suite, we show that two PRNGs pass the NIST Test Suite and have statistically similar output to that of the Mersenne Twister algorithm. Additionally, we provide a method to rank PRNGs by quality of random output, and conclude that two of our PRNGs produce random output with quality that is on par with the quality of the Mersenne Twister. We have found that the trajectory property of the Collatz function displays random behavior, and that the Collatz graph and several other properties of the Collatz function are nonrandom.

The rest of the paper is organized in the following way: Sect. 2 provides definitions and prior research concerning the Collatz conjecture. Section 3 presents descriptions of our algorithms. Section 4 details the experimental setup and presents the experiment's results. Section 5 analyzes the results. Section 6 presents the conclusion and proposed future work.

2 Collatz conjecture

In this section, we define the Collatz function $C(n)$ and its properties. In addition, we introduce an accelerated version of the Collatz function, $T(n)$, and summarize proven characteristics of the behavior of $T(n)$. In later sections, we present applications of these characteristics in generating random numbers.

2.1 The Collatz conjecture properties

Definition 1 The *Collatz Function* [6] can be defined as

$$C(n) = \begin{cases} 3n+1 & if\ n \equiv 1\ (mod 2) \\ \frac{n}{2} & if\ n \equiv 0\ (mod 2) \end{cases}.$$

Let $C^k(x)$ be the result after iteratively applying $C(n)$ k times to x. We can efficiently implement the $C(n)$ function by calculating $\frac{n}{2}$ using a right bit-shift operation $n >> 1$ and calculating $3n+1$ using $(n << 1) + n + 1$.

Definition 2 The *trajectory* of a number x is defined as the sequence $\{x, C(x), C^2(x), C^3(x), \ldots\}$.

Example 1 The trajectory of 13 is the sequence $\{13, 40, 20, 10, 5, 16, 8, 4, 2, 1\}$.

Definition 3 An alternate definition of the Collatz function is as follows [13]:

$$T(n) = \frac{n * 3^{X(n)} + X(n)}{2}$$

where $X(n) = n(mod 2)$.

Let $T^k(n)$ be the result of iteratively applying $T(n)$ k times to n. In addition, let $X_k(n) = X(T^k(n))$.

Definition 4 Let n be a positive integer and k be the smallest positive integer such that $n < 2^k$. The *encoding vector* of n is defined as the vector $E_k(n) = \{X_0(n), X_1(n), X_2(n), \ldots X_{k-1}(n)\}$ [13].

Example 2 The encoding vector of 26 is the vector $\{0, 1, 0, 0, 1\}$.

Definition 5 The *maximum trajectory point* of a positive integer n is the largest element of its trajectory.

Conjecture 1 *For all positive integers $n \neq 1$, there exists a positive integer k such that $C^k(n) = 1$.*

Conjecture 1, the Collatz conjecture first proposed by Lothar Collatz in 1937, remains an open question, but has been experimentally proved to be true for all positive integers up to 10^{13} [14]. The Collatz conjecture is also known as the $3x+1$ problem, the Ulam conjecture, and the Syracuse conjecture. Concerning the Collatz conjecture, Erdös stated that Mathematics is not yet ready for such problems", and offered $500 for a solution. Attempts to prove the Collatz conjecture have yielded interesting connections between the conjecture and the 2-adic integers, computability theory, and Diophantine approximation of $log_2 3$ [7].

Definition 6 Let the set V be the set of positive integers. For each positive integer n, consider a directed edge from $s(n)$ to n, where $s(n)$ is the following relation.

$$s(n) = \begin{cases} 2n\ if\ n \equiv 0, 1, 2, 3, 5 (mod 6) \\ \frac{n-1}{3}, 2n\ if\ n \equiv 4\ (mod 6) \end{cases}$$

$s(n)$ is defined to be the inverse Collatz relation.

Let the set E be the set of all such edges.

Then the *Collatz graph* is defined as the ordered pair (V, E) [8].

Conjecture 1 implies that such a graph is an acyclic connected graph, i.e., a tree, that contains all of the positive integers. The only exception to these properties is the cycle of 4–2–1.

Figure 1 shows a subgraph of the Collatz graph. It contains a cycle at 4. Nevertheless, based on the Collatz Conjecture, the subgraph that starts with 4 is a tree.

The above definitions are applied to random number generation in Algorithms 1, 3, 4, 5, 7, and 8, which are presented in Sect. 3.

Theorem 1 *Two positive integers n and m have the same encoding vector* $E_k(n) = E_k(m)$ *if and only if* $n \equiv m(\mathrm{mod} 2^k)$.

Theorem 2 *For a fixed k, the function that assigns the encoding vector* $E_k(n)$ *to a positive integer n, is periodic with period* 2^k *and assumes all values in* $\prod_0^{k-1}\{0,1\}$, *the set of all functions mapping* {0, 1,… k−1} to {0, 1}.

Theorem 3 *The sequence* $X_0, X_1, X_2, \ldots X_{k-1}$ *constitutes a family of independent random variables.*

The above three theorems are due to Terras [13]. These theorems are used later in Sect. 3 as part of Algorithm 1.

Conjecture 2 *Given a positive odd integer* $n \neq 1$, *let* n_1 *be the next odd integer in the trajectory of n. Suppose that each successive odd number is in a random residue class mod* 2^k *for all positive values of k. Then the probability that* $n_1 = \frac{3n+1}{2}$ *is* $\frac{1}{2}$, *the probability that* $n_1 = \frac{3n+1}{4}$ *is* $\frac{1}{4}$, *and so on. Thus, the expected geometric ratio between consecutive odd numbers is* $\frac{3}{4}$ [7].

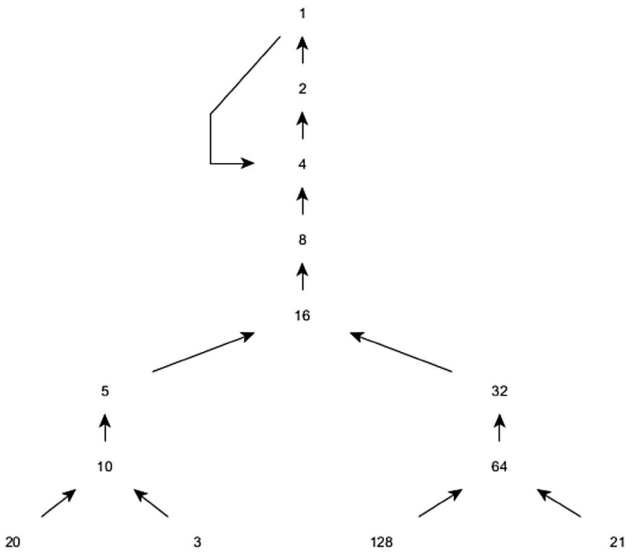

Fig. 1 A subgraph of the Collatz graph

If Conjecture 2 is true, then the following can be concluded: Suppose that n is a positive odd number. Then the expected number of even values between n and the next odd number k in the trajectory of n is 2. Thus, the average ratio between even and odd numbers in the trajectory of n is 2:1.

Proof As stated in Conjecture 2, the average geometric ratio between n and k is $\frac{3}{4}$. On average, then, $k = \frac{3n+1}{4}$. The two even numbers between n and k would be $3n + 1$ and $\frac{3n+1}{2}$.

Conjecture 2 is used as the basis of Algorithm 2, which is shown in Sect. 3.

2.2 Previous works

Previous work on the application of the Collatz function in cryptography have primarily focused on using variations of the Collatz function as a hash function or a method of encryption [1–3]. There are some informal discussions of a Collatz-based PRNG on Google Groups and Reddit, but as far as we know, there has been no attempt to research the feasibility of the inverse Collatz relation or the properties of the Collatz relation as a pseudo-random number generator.

3 Design of Collatz-based PRNGs

In this section, we describe the PRNGs we have developed. Each PRNG utilizes a property of the Collatz conjecture to generate random numbers.

3.1 Algorithm 1

Let $b(n)$ be the following function:

$$b(n) = n >> (\lceil log_2 n \rceil - 95)$$

Given a seed $n > 2^{95}$, let

$$a_0 = 0$$

$$a_i = (b(n + a_{i-1}) \oplus E_k(n + a_{i-1})[:95] + a_{i-1}) \left(\mathrm{mod} 2^{128}\right)$$

where $E_k(n)$ is the encoding vector of n, and $x[:k]$ represents the first k elements of vector x.

We obtain a binary sequence by concatenating vectors $E_k(n + a_0), E_k(n + a_1), E_k(n + a_2) \ldots$ together.

This algorithm is guaranteed to generate pseudo-random output.

Proof By Theorem 3, the encoding vector (see Definition 4) of a positive integer n constitutes a family of independent random variables.

In addition, the output of Algorithm 1 has a minimum period of 2^{32}.

Proof Suppose that n_0 is the original seed supplied by the user.

Thus, $n + a_i \in \{n_0, n_0 + 2^{128} - 1\}$ for all $i \in \mathbb{N}$.

According to Theorems 1 and 2, each possible value of $n + a_i$ produces a different encoding vector because they are not in the same residue class $mod 2^{128}$. Thus, we only need to count the number of guaranteed unique values of $n + a_i$.

In order to do so, we establish an upper bound on the value of $a_i - a_{i-1}$, for all $i \in \mathbb{N}$. We make the following observations:

1. Because $b(n)$ and $E_k(n)[:95]$ are bit-strings with length 95, their base 10 representations are less than 2^{96}.
2. $a_i - a_{i-1} = b(n) \oplus E_k(n)[:95] < 2^{96}$

Because $n + a_{i-1}$ is incremented by $a_i - a_{i-1} < 2^{96}$ for $i \in \mathbb{N}$, and $n + a_i$ can take on 2^{128} values, the guaranteed number of unique $n + a_i$ is $\frac{2^{128}}{2^{96}} = 2^{32}$. This gives us the guaranteed period of the algorithm.

3.2 Algorithm 2

Algorithm 2 uses the trajectory property to generate random numbers, and relies on Conjecture 2 to ensure the uniform distribution of 0's and 1's.

Suppose we are given the seed n. Let the sequence S be the trajectory of n. By removing the first even number after each odd number in S, the frequency of odd and even numbers in S are expected to be the same according to Conjectures 2 and 3. We then replace each number in S with its residue mod 2 and return this sequence as the output.

Example 3 Given the seed 17, the algorithm generates the output $\{0, 1, 0, 0, 1\}$.

3.3 Algorithm 3

Given a seed n, we create a pseudo-random sequence of length i by returning the lengths of the trajectories (see Definition 2) with starting values beginning from n and going to $n + i - 1$ in increments of 1.

Example 4 Given a seed 3 and $i = 4$, the algorithm generates the output $\{8, 3, 6, 9\}$.

3.4 Algorithm 4

Given a seed n, we create a list containing n as its only element, and add integers to the list by recursively applying a function M3(x) to n until a cycle occurs. M3(x) was inspired by a function found in [2].

$$M3(x) = \begin{cases} \frac{x}{3} & if\ x \equiv 0\ (mod 3) \\ 2x + 1 & if\ x \equiv 1\ (mod 3) \\ 4x - 1 & if\ x \equiv 2\ (mod 3) \end{cases}$$

Each time that the function $M3(x)$ in Algorithm 4 is appended onto the end of the list, we set $x = M3(x)$. Since this is not the Collatz function, there is the possibility of having cycles within the i iterations. Thus, the algorithm is designed to terminate in the case that a cycle does occur

Example 5 Given a seed 7, the algorithm generates the output $\{7, 15, 5, 19, 39, 13, 27, 9, 3, 1\}$.

In the following 2 subsections, we utilize the Collatz graph (see Definition 6) to generate random numbers.

3.5 Algorithm 5

Algorithm 5 generates pseudo-random numbers by traversing the Collatz graph. It uses a priority queue to determine the order in which the numbers are visited, with smaller numbers being traversed first. When the traversal terminates after a predetermined number of steps, Algorithm 5 appends either a 1 or a 0 at the end of each element remaining in the priority queue, depending on parity of the number of ones in the original bit-string. The modified elements are then converted back to base ten, creating the pseudo-random sequence.

Example 6 Given the seed 25, the algorithm generates the output $\{18, 27, 15, 3, 34, 10, 6, 23\}$.

3.6 Algorithm 6

In this algorithm, we traverse the Collatz graph to generate random numbers, again using a priority queue. However, when determining the order of traversal, we take into account both the value of a node and its distance from 16 on the Collatz graph.

Assuming that $l(n)$ represents the length of the trajectory of the positive integer n and that S is the seed, nodes with smaller values of $c(n$ are traversed first, where $c(n)$ is the following function:

$$c(n) = 3 * (l(n) - 4) + (S - n).$$

Because $l(n) - 4$ is much smaller than $S - n$, we multiply $l(n) - 4$ by 3 to give it more weight.

In addition, because the Collatz graph could potentially be an infinite graph, the user must provide another parameter c, which specifies the total number of nodes that are explored. Upon exploring c nodes, the algorithm terminates.

Example 7 Given the seed 67 and $c = 50$, the algorithm generates the output $\{4, 12, 6, 9, 3, 8\}$.

3.7 Algorithm 7

Algorithm 7 uses the maximum trajectory point (see Definition 5) to a sequence of random numbers with length i. Given a seed value n and a value i, we create a pseudo-random sequence of length i by returning the maximum trajectory points for the numbers from n to $n + i - 1$.

Example 8 Given a seed value of 15 and $i = 8$, the algorithm generates the output $\{65, 237, 272, 273, 377, 432\}$.

4 The NIST statistical tests

In this section, we present our experimental setup and the results from evaluating our algorithms using the NIST Test Suite. In addition, we explain the significance of the results.

4.1 Explanation of statistical testing

The NIST Test Suite evaluates the null hypothesis H_0, which states that a given candidate sequence is random. If H_0 is false, then it is rejected in favor of the alternate hypothesis H_a, which states that the sequence is non-random.

Each statistical test in the NIST Test Suite generates a p value, which determines whether we should reject H_0. The p-value from a statistical test is the probability that a perfect RNG would produce a sequence less random than the one that was tested. Thus, a large p-value would indicate that the tested sequence is random, and a small p-value would indicate that the tested sequence is nonrandom [11].

The level of significance α within a test is the point at which we reject H_0, and is usually fixed beforehand to be in the range $[0.001, 0.05]$. For our tests, we have selected $\alpha = 0.05$. Thus, for p-values $< \alpha$, we reject H_0; otherwise, we accept H_0 [5].

In addition, the NIST Test Suite provides 2 further analysis methods that minimize the occurrences of Type I and II errors. Type I error occurs when we reject H_0 and accept H_a when the sequence is actually random. Type II error occurs when we fail to reject H_0 when the sequence is non-random [5]. First, the NIST Test Suite examines whether the p-values for a statistical test are uniformly distributed using a Chi squared test, resulting in another p-value called $P\text{-}value_T$. Second, the NIST Test Suite calculates the proportion of sequences with p-value $\geq \alpha$. If this proportion is within the interval $p \pm 3\sqrt{\frac{p(1-p)}{n}}$ where $p = 1 - \alpha$ and n is the sample size, and $P\text{-}value_T \geq 0.0001$, then H_0 is accepted [11, 12].

4.2 Experimental setup

For the reported PRNGs, we have generated 200 sequences of 10^6 bits each, and have tested these sequences for randomness using the NIST Test suite. Additionally, we have evaluated the results for the Mersenne Twister algorithm.

The NIST Test Suite does not provide a relative ranking of each PRNG. Since relative ranking might be important for certain applications, we define a simple method for comparing the performance of PRNGs (see Sect. 6). This method assigns points to each PRNG based on the p-values from the NIST Test Suite, with more points given for lower p-values. A PRNG with a high number of points is considered poor, and a PRNG with a low number of points is considered better.

Because the NIST Test Suite only tests bit-strings, we convert the output of Algorithms 3, 4, 5, 7, and 8 from base 10 into base 2 and remove the most significant bit of each resulting bit-string since it is guaranteed to be 1. Next, we concatenate the produced bit-strings to form a single bit-string for that PRNG, Finally, this bit-string is tested to obtain data for that PRNG. This process is illustrated in Example 10.

Example 9 If a PRNG produced the sequence {5, 7, 3} then the tested bit-string for this PRNG would be 01111.

4.3 Results

In this section, the results from the NIST battery of statistical tests are displayed for Algorithm 1, Algorithm 2, and the Mersenne Twister, in Tables 1. For brevity, we only provide the results for the algorithms that passed the NIST tests.

For each algorithm, the column 1, titled P-values, shows the $P\text{-}value_T$ and the column 2, titled proportion shows the proportions of p-values $\geq \alpha$.

For the selected value of α and sample size of 200, the proportion of p-values $\geq \alpha$ must be in the interval $[\frac{181}{200}, \frac{199}{200}]$ with the exception of the Random Excursions and Random Excursions Variant tests, which have a passing interval of $[\frac{110}{123}, \frac{123}{123}]$.

Table 1 shows the P-values and Proportions obtained for Algorithm 1, Algorithm 2, and the Mersenne Twister and demonstrates that both Algorithm 1 and Algorithm 2 pass the NIST Test Suite. Comparing the algorithms results with the Mersenne Twister results, we can see that the results for each test differ by no more than 6. Thus, there is no significant statistical difference between the outputs of Algorithm 1, 2 and the Mersenne Twister.

Table 1 Results for Algorithm 1, Algorithm 2, and the Mersenne Twister

Statistical test	Algorithm 1		Algorithm 2		Mersenne Twister	
	P-value	Proportion	P-value	Proportion	P-value	Proportion
Frequency	0.605916	188/200	0.334538	191/200	0.504219	191/200
Block frequency	0.255705	187/200	0.514124	192/200	0.162606	191/200
Cumulative sums 1	0.213309	189/200	0.657933	191/200	0.017912	191/200
Cumulative sums 2	0.410055	191/200	0.699313	189/200	0.668321	192/200
Runs	0.699313	194/200	0.807412	191/200	0.375313	189/200
Longest run	0.176657	195/200	0.514124	191/200	0.825505	189/200
Rank	0.798139	187/200	0.997823	191/200	0.689019	193/200
FFT	0.474986	187/200	0.249284	184/200	0.825505	190/200
Non overlapping template	0.620053	190/200	0.739918	194/200	0.574903	191/200
Overlapping template	0.230755	189/200	0.719747	190/200	0.514124	186/200
Universal	0.544254	191/200	0.504219	195/200	0.383827	191/200
Approximate entropy	0.029796	188/200	0.073417	188/200	0.955835	187/200
Serial 1	0.068999	190/200	0.474986	187/200	0.883171	190/200
Serial 2	0.564639	191/200	0.484646	195/200	0.474986	193/200
Linear complexity	0.014051	190/200	0.626709	189/200	0.689019	193/200
Random excursions	0.578756	117/123	0.467909	116/124	0.424377	124/131
Random excursions variant	0.459566	117/123	0.500652	117/124	0.467753	122/131

Table 2 PRNG quality scores

Algorithm	Points
Mersenne Twister	752
Algorithm 1	752
Algorithm 2	694
Algorithm 3	29,252
Algorithm 4	33,758
Algorithm 5	35,226
Algorithm 6	40,150
Algorithm 7	30,024

5 Analysis

In this section, we analyze the results of the evaluation and present a method for comparing the performances of PRNGs.

Algorithms 1 and 2 pass all the NIST tests. Hence, these PRNGs are acceptable from a statistical point of view, and are suited for non-cryptographic applications such as simulations or games. Their applicability to cryptographic applications must be evaluated using cryptanalysis methods.

Because the NIST Test Suite does not provide a method for comparing the quality of PRNGs, we have utilized a method which scores the PRNGs using the following function [9]:

$$G(p) = \begin{cases} 0 & \text{if } p > .05 \\ 2 & \text{if } .002 < p \leq .05 \\ 4 & \text{if } p \leq .002 \end{cases}$$

where p is a p-value from a statistical test. The total score for a PRNG is the sum of $G(p)$ for all p-values across all tests for that PRNG. A higher score indicates a poor PRNG, while a lower score indicates a better PRNG. We can then rank the PRNGs based on their total scores.

The NIST Test Suite uses the complementary error function when computing p-values, so p-values are considered good if they are closer to 1 [11]. We replace $G(p)$ with a new function $H(p)$ to reflect this difference.

$$H(p) = \begin{cases} 0 & \text{if } p > .05 \\ 2 & \text{if } .002 < p \leq .05 \\ 4 & \text{if } p \leq .002 \end{cases}$$

The total scores for all PRNGs are displayed in Table 2, titled PRNG Quality Scores.

The score for Algorithm 2 is lower than the score for the Mersenne Twister, and Algorithm 1 and the Mersenne Twister have the same score. Thus, we can conclude that the quality of random output from Algorithms 1 and 2 is on par, or slightly better, than that of the Mersenne Twister.

6 Conclusion and future work

We have experimented with several pseudo-random number generators based on the properties of the Collatz conjecture, and tested these PRNGs using the NIST Test Suite. We have proved that Algorithm 1, which is based off the encoding vector property, generates uniformly distributed bit-strings with a period of 2^{32}. In addition, the PRNGs based on the trajectory and encoding vector properties (Algorithms 1 and 2) successfully pass the NIST Test Suite. We have ranked the PRNGs and the Mersenne Twister by the quality of random output and concluded that Algorithms 1 and 2 perform on par with the Mersenne Twister. Thus, Algorithms 1 and 2 can be used in certain cryptographic and non-cryptographic applications. On the other hand, we have found that the length of the trajectory, the Collatz graph, the maximum trajectory point, and the M3(x) function do not display statistically random behavior.

Future work will include examining the suitability of our PRNGs for stringent cryptography systems through the application of rigorous testing suites and security analysis, as well as increasing the speed of and evaluating the computational complexity for each PRNG. In addition, we will further research the Collatz graph as a source of randomness.

Finally, we plan to explore additional methods for evaluating and ranking PRNGs based on the quality of random output.

Acknowledgements We thank Ashley Chen and Michael Liu for contributing to drafts of this paper, as well as developing Algorithms 3, 4, 5, 6, and 7.

References

1. Bauchot F (2012) Efficient and low power encrypting and decrypting of data. US Patent 8,130,956 B2
2. Ciet M, Farrugia A, Icart T (2013) System and method for a Collatz based hash function. US Patent Application Publication 2013/0108038 A1
3. Givens R (2006) On Conway's generalization of the 3x + 1 problem. Honors Theses. Paper 484. Retrieved from https://scholarship.richmond.edu/cgi/viewcontent.cgi?article=1496&context=honors-theses. Accessed 22 Apr 2019
4. Haahr M (2017) Introduction to randomness and random numbers. https://www.random.org/randomness/. Accessed 22 Apr 2019
5. Kim S, Umeno K, Hasegawa A (2004) Corrections of the NIST statistical test suite for randomness, chaos-based cipher chip project. https://eprint.iacr.org/2004/018.pdf. Accessed 22 Apr 2019
6. Kontorovich A, Lagarias J (2010) Stochastic models for the 3x + 1 and 5x + 1 problems and related problems. The ultimate challenge: the 3x + 1 problem, Amer Math Soc pp 131–188
7. Lagarias J (1985) The 3x + 1 Problem and its Generalizations. Am Math Monthly 92(1):3–23
8. Lang W (2014) On Collatz words, sequences, and trees. J Int Sequences 17:1–14
9. Martin P (2002) An analysis of random number generators for a hardware implementation of genetic programming using FPGAs and handel-C. In: GECCO proceedings of the 4th annual conference on genetic and evolutionary computation, Morgan Kaufmann Publishers, San Francisco, pp 837–844
10. Matsumoto M, Nishimura T (1998) Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans Model Comput Simul 8(1):3–30
11. Rukhin A et al (2018) A statistical test suite for random and pseudo-random number generators for cryptographic applications. Special Publication 800-22, Revision 1a. Retrieved from https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf. Accessed 22 Apr 2019
12. Soto J (1999) Statistical testing of random numbers. National Institute of Standards and Technology, Gaithersburg
13. Terras R (1976) A stopping time problem on the positive integers. Acta Arithm 30(3):241–252
14. Tzanis E (2003) Collatz conjecture: properties and algorithms. Retrieved from https://www.delab.csd.auth.gr/bci1/Panhellenic/700tzanis.pdf. Accessed 22 Apr 2019