Cellular Automaton–Based Emulation of the Mersenne Twister

Kamalika Bhattacharjee*
Nitin More
Shobhit Kumar Singh
Nikhil Verma

Department of Computer Science and Engineering
National Institute of Technology
Tiruchirappalli, Tamilnadu – 620015, India
**corresponding author, kamalika.it@gmail.com*
{Nitinmore6990, sks.shobhit12, nikhilverma793}@gmail.com

The Mersenne Twister (MT) (MT19937), developed 30 years ago, is the de facto pseudorandom number generator (PRNG) used in many computer programs. This paper proposes a candidate that offers a randomness quality that is better than MT19937 and its sisters SFMT19937 and TinyMT. A special three-neighborhood, two-state cellular automaton (CA), called CA (150′) is the underlying model of this PRNG. The same working style of MT19937 is used, while avoiding the problems of the MT, like a large state space and the zero-access initial state problem. Nonlinearity is added in the base simple linear CA such that the properties of the base CA are not violated. Finally, a PRNG is developed using this CA that beats MT19937 as well as its advanced versions over the standard empirical platforms Dieharder, TestU01 and NIST.

Keywords: Mersenne Twister (MT); Sophie Germain prime; pseudorandom number generator (PRNG); maximal-length cellular automata; CA (150′); nonlinearity

1. Introduction

Pseudorandom number generators (PRNGs) are inherently essential in modern computing paradigms including stochastic computing, Monte Carlo simulations, testing and randomized algorithms. Since its proposal by Makoto Matsumoto and Takuji Nishimura in 1997 [1], the Mersenne Twister (MT) has become the de facto PRNG in many software and language implementations, like Maple, Python, PHP, GLib, MATLAB and the GCC compiler. The reason is that it has a huge state space—the most popular version (MT19937) has 19 937 bits of state—due to which, the *period* is very large (the sequence will not

https://doi.org/10.25088/ComplexSystems.32.2.139

repeat before $2^{19937} - 1$ numbers for MT19937). Also, the performance of the MT and its sister PRNGs, like the SIMD-oriented fast Mersenne twister (SFMT) [2], is very good on empirical testbeds [3], which makes them attractive choices for ease of implementation in software.

However, the MT family has some serious drawbacks. First, it computes and stores a huge state even though only a few numbers are required; for example, in the case of MT19937, 623 32-bit numbers are generated and stored every time. Second, it suffers from the *zero-excess initial state* problem where, if a large number of bits in the initial state are 0, nonchaotic patterns may be generated, making the sequence nonrandom. Third and most important, due to its linearity, the MT as well as its sisters SFMT and tiny Mersenne twister (TinyMT) [4] fails several important statistical tests, such as Marsaglia's linear-complexity test, the binary-rank test [5] and BigCrush of the TestU01 library [6].

We propose a new PRNG that works in the same way as the MT, but uses a much smaller state space, while giving better performance than the MT family in all empirical tests. Also, it is fast enough and easy to implement for general usage. Our PRNG is based on cellular automata (CAs), which have long been established as an effective choice for a source of randomness [7]. More specifically, we exploit the properties of a special two-state three-neighborhood cellular automaton (CA), called CA (150′), introduced in [8]. This CA, defined over a null boundary, takes rule 150 for every cell except the first cell, which uses rule 90 and has the potential to generate a maximal cycle length of $2^n - 1$ if the size of the CA n is a Sophie Germain prime. To improve the inherent randomness quality of the CA, we also use the *Temper* and *Twist* functions, similar to MT19937. An initial version of this work is reported in [9]. This paper reports an extension of that work that further improves the proposed PRNG and establishes itself as the best random number generator, beating the MT family in all the standard empirical testbeds: Dieharder, BigCrush and NIST [10].

The paper is organized as follows: Section 2 briefly states the background required to understand the underlying concepts. In the next three sections, we gradually develop our PRNG and improve it based on its performance in testbeds. Section 3 recollects the basic prototype of the PRNG based on CA (150′). We choose 1409 as the CA size so that it generates 44 32-bit numbers at a time, just like the MTs, but takes only 1409 bits as its state, making it lightweight. A proper seed is selected to prevent the zero-excess initial state problem. Then we apply tempering and test our PRNG over the statistical testbeds. We can see that our PRNG fails the rank test of Dieharder, the Matrix Rank test of BigCrush and the LinearComp test of BigCrush. So to

further enhance its randomness, in Section 4 we add a nonlinear component to the PRNG. Two approaches are taken, Twist of MT19937 as well as XOR + multiplication. Now the PRNG passes all tests of Dieharder, but the linearity tests of BigCrush can still detect nonrandomness in the PRNG. Hence, in Section 5 we use a different technique. Instead of adding a nonlinear component outside the CA, which makes the PRNG slow, we switch to nonlinear CAs. This section identifies an appropriate nonlinear CA that satisfies our criteria and develops our final PRNG. We show that this PRNG performs better than all of its competitors, MT19937 and SFMT19937, passing almost all existing empirical tests.

2. Background

Here, a brief overview of the MT, different testbeds used in this paper and CAs, especially CA (150′), is given.

2.1 The Mersenne Twister

The MT, a well-known general-purpose PRNG, is derived from a twisted generalized feedback shift register (TGFSR) such that its period is a Mersenne prime $2^{19\,937} - 1$. This PRNG can generate a sequence of numbers very quickly by avoiding multiplication and division and efficiently utilizing memory and cache.

2.1.1 Algorithm

The MT works in the following way (see Figure 1): first, a 32-bit integer, considered as the seed, is employed to initiate the state of the TGFSR, which is 19 937 bits. This is done by consecutively filling the next 624 words of the initial state array using some XOR, shift, addition and multiplication with a constant over the previous word in the array (the first word being the seed). Observe that $32 \times 624 = 19\,936$. Then the Twist function is applied to produce the next state of the PRNG. However, instead of directly generating each state as output, the internal states are divided into 624 32-bit words (integers) and the Temper function is applied over each integer before producing it as output. The Twist function is used again to update the next state of the TGFSR but only when all 624 numbers are exhausted.

Tempering is defined by the following transformations applied successively:

$$\mathbf{y} = \mathbf{x} \oplus (\mathbf{x} \gg u) \tag{1}$$

$$\mathbf{y} = \mathbf{y} \oplus ((\mathbf{y} \ll s) \text{ AND } \mathbf{b}) \tag{2}$$

https://doi.org/10.25088/ComplexSystems.32.2.139

$$\mathbf{y} = \mathbf{y} \oplus ((\mathbf{y} \ll t) \text{ AND } \mathbf{c}) \tag{3}$$

$$\mathbf{z} = \mathbf{y} \oplus (\mathbf{y} \gg l) \tag{4}$$

where the integers t, u, l, s are tempering parameters, $\mathbf{c}$ and $\mathbf{b}$ are bitmasks of the word size of the computer, $\mathbf{x}$ is the next number generated in the series and $\mathbf{z}$ is the vector returned. Observe that all operations are bitwise XORs, ANDs, ORs and shifts. Table 1 depicts the

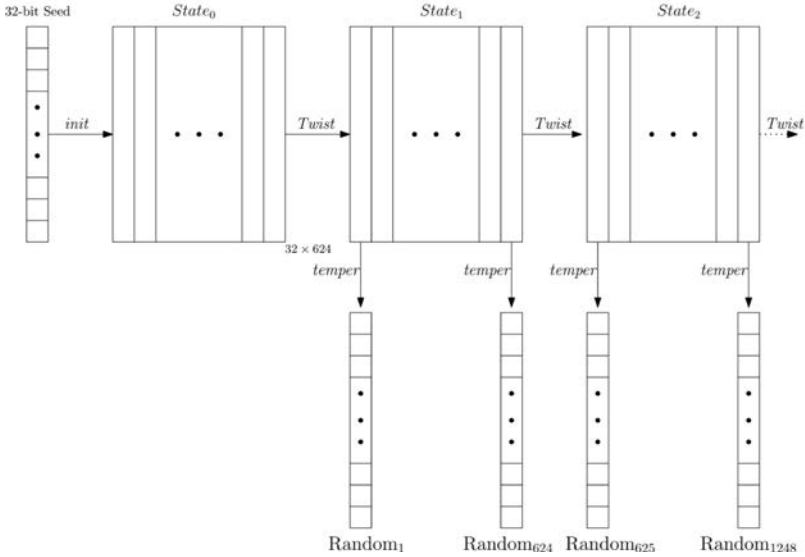


Figure 1. Working principle of the MT.

	Value	
Parameter	32-bit	64-bit
n	624	312
w	32	64
r	31	31
m	397	156
a	0x9908B0DF	0xB5026F5AA96619E916
u	11	29
s	7	17
t	15	37
l	18	43
b	0x9D2C5680	0x71D67FFFEDA6000016
c	0xEFC60000	0xFFF7EEE00000000016

Table 1. MT19937 32-bit and 64-bit parameters.

32-bit and 64-bit parameters used for MT19937. These parameters are selected in such a way that the period is the Mersenne prime $2^{19\,937} - 1$, longest among all its predecessors, which is one of the reasons for the popularity of the MT. Further, for every $1 \le k \le 623$, the MT is k-distributed to 32-bit accuracy. However, the MT is not cryptographically secure because the Twist function is not a one-way function.

2.1.2 Limitation and Improvement

The MT has a serious limitation in initialization, known as the zero-access initial state problem. If the number of zeros in the initial state is sufficiently high, then, for more than 10 000 generations, the generated sequence may continue to have many zeros, resulting in correlated output sequences. Furthermore, due to its large state space of 19 937 bits and linearity, the next state of the MT is easy to predict from the previous states, which makes it unsuitable for cryptographic purposes. New variants are proposed to avoid these limitations:

SFMT. Single instruction, multiple data (SIMD)-oriented fast MT (SFMT) uses SIMD (like 128-bit integer) operations and multistage pipelines along with all properties of the MT. It generates both 32-bit and 64-bit unsigned integer numbers, plus double-precision floating-point numbers. It is almost two times faster than the MT and has an improved equidistribution property and a quicker recovery from the zero-excess initial state problem than the MT. Here we use SFMT19937, which has the same period length as MT19937.

TinyMT. TinyMT is introduced to operate over a small state space of 127 bits, resulting in a shorter period of $2^{127} - 1$. Still, it passes many noncryptographic statistical tests. Here we use TinyMT32, which has to be initialized with the following set of parameters:

$$\begin{aligned} \text{mat1} &= \text{0x8f7011ee} = 2\,406\,486\,510 \\ \text{mat2} &= \text{0xfc78ff1f} = 4\,235\,788\,063 \\ \text{tmat} &= \text{0x3793fdff} = 932\,445\,695. \end{aligned}$$

2.2 Testbed

This paper uses three standard batteries of empirical tests: Dieharder, NIST statistical test suite and BigCrush of the TestU01 library. Next we briefly discuss them.

2.2.1 Dieharder

The Diehard tests [11] are a battery of statistical tests developed by George Marsaglia in 1995 for measuring the quality of a PRNG. Later, an extended version called Dieharder that contains 114 tests

https://doi.org/10.25088/ComplexSystems.32.2.139

was developed [5]. Similar to the Diehard tests, the Dieharder tests also return p-values that indicate the performance of a PRNG in the test.

2.2.2 TestU01

TestU01 is an ANSI C language software library that provides a collection of utilities for statistically testing the randomness of PRNGs [6]. Both classical statistical tests for PRNGs as well as many original tests are implemented as part of the library. These tests can be used over streams of random numbers stored in files as well as any user-defined generators and the predefined generators in the library. If the corresponding p values are within 0.001 to 0.999, the PRNG passes the test. We have considered the most stringent battery of tests—the "Big Crush," containing 160 tests.

2.2.3 NIST

The NIST statistical test suite [10] is comprised of 15 tests generated to test cryptographic properties of PRNGs. The reference distribution is considered as the standard normal and the chi-square (χ^2) for many of these tests. For a sample size α, the range of acceptable proportions for x is calculated as

$$(1-\alpha) \pm 3\sqrt{\frac{\alpha(1-\alpha)}{m}}$$

where $\alpha = 0.01$ and x is the minimum pass rate.

2.3 Cellular Automata

We consider three-neighborhood, two-state n-cell CAs under the null boundary condition where each cell of the CA can take a different rule, that is, the CA is nonuniform. Such a CA can be represented by a *rule vector* $\mathcal{R} = \langle R_0, R_1, \ldots, R_{n-1}\rangle$, which indicates, for each i, $0 \le i \le n-1$, the i^{th} cell takes rule R_i to update its next state. For example, consider two rules:

$$\begin{aligned} \text{rule } 90 : R_{90}(x, y, z) &= x \oplus z \\ \text{rule } 150 : R_{150}(x, y, z) &= x \oplus y \oplus z \end{aligned}$$

where $x, y, z \in \{0, 1\}$ represents the states of the left neighbor, the cell under consideration and the right neighbor, respectively. These are *linear* rules.

Definition 1. A rule $R_i : \{0, 1\}^3 \to \{0, 1\}$ is called linear if $R_i(x, y, z) = c_0 x \oplus c_1 y \oplus c_2 z$, where $c_i \in \{0, 1\}$ is a constant; otherwise it is a nonlinear rule.

Definition 2. A rule $R_i : \{0, 1\}^3 \to \{0, 1\}$ is complemented if $(x, y, z) = 1 - R_i(x, y, z)$, where rule R_i is a linear rule.

There are eight linear rules—0, 60, 90, 102, 150, 170, 204 and 240, and eight complemented rules—15, 51, 85, 105, 153, 165, 195 and 255. A CA that uses these 16 rules can be efficiently characterized by algebraic tools. We call a rule *nonlinear* if it is neither linear nor complemented.

Definition 3. If a rule vector $\mathcal{R}$ contains only linear rules, then the CA is called linear. If at least one rule is complemented and the rest are linear, then the CA is a complemented CA. Otherwise, (i.e., at least one rule of $\mathcal{R}$ is nonlinear) the CA is a nonlinear CA.

The snapshot of all cells at any time instant is called the *configuration* of the CA. A CA evolves through its configuration space $C_n = \{0, 1\}^n$ by its *global transition function* G_n, where $G_n : C_n \to C_n$.

Definition 4. A CA is reversible if for each of its configurations $x \in C_n$, $x = G_n^t(x)$ for some $t \in \mathbb{N}$.

Definition 5. A rule is balanced if its eight-bit binary representation contains an equal number of zeros and ones.

Out of 256 rules, only 70 rules are balanced. Under the null boundary condition, however, only 62 balanced rules can participate in forming a reversible CA. Some reversible CAs can be of maximal length. An n-cell CA is called a *maximal-length* CA if all except one of the configurations form a single cycle of length $2^n - 1$. Only rules 90 and 150 can form a nonuniform linear maximal-length CA.

Theorem 1. A linear CA is of maximal length if and only if the rule vector of the CA consists of rules 90 and 150, and its characteristic polynomial is primitive over GF(2) [8].

However, any arbitrary sequence of rules 90 and 150 does not form a maximal-length CA. In this paper, we are interested in one special CA formed with these two rules that has the potential to become a maximal-length CA.

Definition 6. An n-cell rule vector $\mathcal{R} = \langle R_0, R_1, \ldots, R_{n-1} \rangle$ is called a CA (150′) if $R_0 = 90$ and $R_i = 150$, where $1 \le i \le n - 1$ [8].

This CA was introduced by Adak and Das in [8], who describe three greedy strategies for finding primitive polynomials of a large degree over GF(2). Some greedy strategies are reported that synthesize (linear) CAs, which are almost always maximal-length CAs, that is, with a primitive characteristic polynomial. The paper reports a list of CA sizes for which CA (150′) is a maximal-length CA (reproduced as Table 2). Here, * marks false positives.

https://doi.org/10.25088/ComplexSystems.32.2.139

2	3	5	9	11	14	23	26	29	35	39	41	50*	53	65				
69	74	81	83	86	89	95	98*	99*	105	113	119	131	134*	146				
155	158	173	179	189	191	194*	209	221	230	231	233	239	243	251				
254	278*	281	293	299	303	323	326	329	338*	350*	359	371	375	386				
398	410*	411	413	419	429	431	443	453	470	473	491	509	515	519				
530	531	543	554	561	575	593	614*	615	629	638	639	641	645*	650*				
653	659	683	686*	713	719	723	725	741*	743	746	749	761	779	783*				
785	803*	809	818	831	833	866	873	893	911	923	950*	953	965	974*				
975	986*	989	993	998	1013	1019	1031	1034*	1041	1043	1049	1070	1103	1106*				
1118	1119	1121	1133	1154	1155	1166	1169	1178*	1185									

Table 2. Value of n such that the n-cell CA (150′) is of maximal length [8]. Here, * marks false positives.

Note that the presence of rule 90 in this CA in its first cell with rule 150 over all other cells helps it to achieve maximality for some n. The following conjecture gives an idea of such n [8]:

Conjecture 1. Characteristic polynomials of CA (150′) of size n are primitive over GF(2) if n and $2n+1$ are both primes (i.e., n is a Sophie Germain prime) [8].

Table 2 validates this conjecture. Here, we can see that, if n is a Sophie Germain prime, that is, both n and $2n+1$ are primes, then CA (150′) of size n has the maximal-length property. There are 42 Sophie Germain primes up to 1200 and for all of them, CA (150′) shows maximality. Hence, for our work, we take an n-cell CA (150′) where n is a Sophie Germain prime.

3. Phase I: Making Our Own Pseudorandom Number Generator

As discussed, the number of cells n of our CA for developing PRNG is chosen to be a Sophie Germain prime. Hereafter, we consider only those n where n is a Sophie Germain prime. So our task is to finalize the specific Sophie Germain prime suitable for our purpose.

3.1 Selection of Proper *n*

Before selecting n, the maximality claim needs to be verified because there are some false positives in Table 2. We can evolve the CA and check if there is actually a cycle of length $2^n - 1$. However, if n is large, it is practically infeasible to run the CA up to 2^n configurations. In this situation, the following property of CA (150′) can reduce the search space:

Property 1. The CA (150′) with size n is reversible for any n except $n \in 3\mathbb{N} + 1$ [8].

Therefore, if n is selected as a Sophie Germain prime such that $n \notin 3\mathbb{N} + 1$, then the initial configuration will eventually repeat, and all configurations are part of some cycles. Using this information, we develop the following strategy to test maximality:

> CA (150′) with size n where n is a Sophie Germain prime and $n \notin 3\mathbb{N} + 1$ is declared as a maximal-length CA if either the complete cycle length test was possible or the initial configuration has not returned even after 10^9 steps, that is, the cycle length is larger than 10^9.

Therefore, our maximality testing method is:

1. Take an n where n is a Sophie Germain prime and $n \notin 3\mathbb{N} + 1$.
2. For that n, check if CA (150′) has configuration $000\cdots00$ as a single-length cycle (this configuration repeats itself). If the CA is a maximal-length CA, all other configurations must be in the same cycle.
3. Take any other configuration as the seed and run CA (150′) on it until the seed is obtained as the next configuration. If the seed is obtained again after exactly $2^n - 1$ steps, the CA is a maximal length for size n.

https://doi.org/10.25088/ComplexSystems.32.2.139

4. For large n, check if the seed is repeated within 10^9 steps. If not, declare the CA as a (possible) maximal-length CA.

Table 3 shows the results for $3 \le n \le 36$ where the seed is $00\cdots001$. The bold rows indicate the sizes n where CA (150′) is a maximal-length CA. In the case $n \in 3\mathbb{N}+1$, the CA is irreversible and not tested. We can observe that for all Sophie Germain primes up to 36, CA (150′) is a maximal-length CA. We also test this scheme on arbitrary large values of Sophie Germain primes (up to 2000) and see that for all those n, the period length of the CA is greater than 10^9.

n	Number of Configurations	Observed Cycle	n	Number of Configurations	Observed Cycle
3	**8**	**7**	20	1048576	1023
4	16		21	2097152	127
5	**32**	**31**	22	4194304	
6	64	21	**23**	**8388608**	**8388607**
7	128		24	16777216	2097151
8	256	15	25	33554432	
9	**512**	**511**	**26**	**67108864**	**67108863**
10	1024		27	134217728	1048575
11	**2048**	**2047**	28	268435456	
12	4096	1023	**29**	**536870912**	**536870911**
13	8192		30	1073741824	17043521
14	**16384**	**16383**	31	2147483648	
15	32768	31	32	4294967296	63
16	65536		33	8589934592	1227133513
17	131072	4095	34	17179869184	
18	262144	29127	**35**	**34359738368**	**34359738367**
19	524288		36	68719476736	511

Table 3. Observed cycle lengths of CA (150′).

Now, since the size of the state in MT19937 takes the form of $32\times x+1$, we finalize our CA size with a similar n. The benefit is, at any given time, x number of 32-bit outputs will be ready from a single configuration, with only one unused bit. However, to make the size of each configuration not too large, while keeping the cycle length not too small, we select $n = 1409$ for the PRNG, a Sophie Germain prime of the form $32\times x+1$. Here, every single configuration produces 44 32-bit random numbers.

3.2 Need of Temper Function

However, the presence of rules 90 and 150 in the CA always creates equilateral triangles in its spacetime diagram. See Figure 2 for an example with $n = 353$. If we extract numbers containing these patterns, they will be correlated and not random. To remove these patterns from our PRNG, we use the *tempering* technique that has been utilized by the MT family. In fact, the same tempering functions used by MT19937 are used in this paper to make the comparison between their performance on a similar platform (see equations (1) to (4)). Here x is the 32-bit number generated by the consecutive 32 bits from the CA configuration with the remaining coefficients:

$$\begin{aligned}(u, d) &= (11, 0x\text{FFFFFFFF}16)\\ (s, b) &= (7, 0x\text{9D2C568016})\\ (t, c) &= (15, 0x\text{EFC6000016})\\ l &= 18.\end{aligned}$$

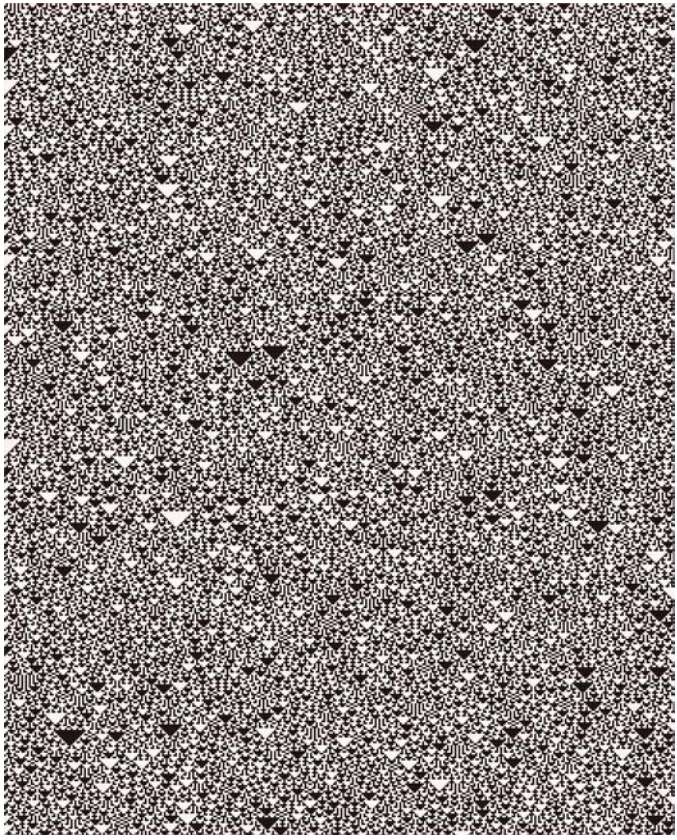


Figure 2. Spacetime diagram for CA (150′) with $n = 353$.

https://doi.org/10.25088/ComplexSystems.32.2.139

Figure 3 shows a sample spacetime diagram for CA (150′) with $n = 353$ that can generate 11 32-bit numbers from one configuration (the binary representation of these numbers is concatenated in Figure 3 to compare with Figure 2). It can clearly be seen that tempering eliminates the equilateral triangles and patterns, significantly improving the randomness quality of the produced numbers.

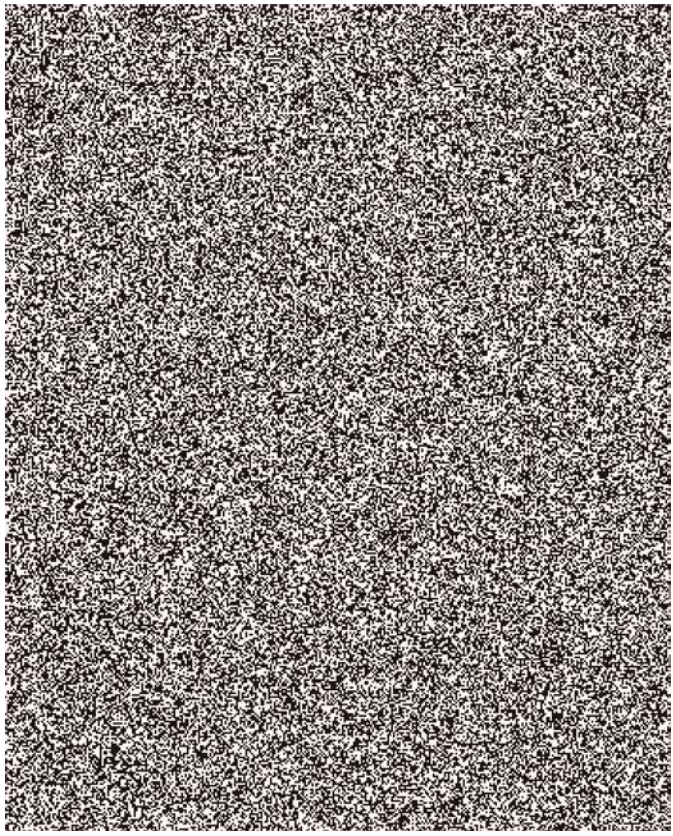


Figure 3. Spacetime diagram for tempered numbers generated using CA (150′) with $n = 353$.

3.3 Finding the Best Seed

The first characteristic of a PRNG is that its sequence can be reproduced by using the same seed. Hence, the choice of a good random seed is crucial. We take the following steps to obtain the best-performing seed for our PRNG:

1. Generate 1000 random seeds using rand, where each seed has an equal distribution of zeros and ones for every 32 bits. This is to remove any bias of zeros and ones. A seed with an equal distribution of zeros and ones increases the probability of an equal distribution throughout the cycle.
2. For each seed, generate a 12MB binary file using the 32-bit outputs of the PRNG.
3. Test the binary file on the Dieharder [5] test suite.
4. Since Dieharder works on large file sizes, a small file size of 12MB will obviously result in many failed tests. However, if the seed fails more than six tests, testing stops.
5. Out of all the seeds, those passing more than 40 tests before stopping are considered.

After complete analysis, seed numbers 187 and 367 are found to be the best-performing seeds. These seeds are shown in Table 4. In the following section, we consider results of only these two seeds.

Seed Number 187
10100111111000110100100110100110001010111111100110001000011
10100000101101100010010100111010001111100010001100011110101
11111000001100111010011100010011000011110011000000011110101
10010011100111010001101110101111010100010101010011110010011
01010110001110010100000100100111101000010110110101101010101
10001100111010011000110101101010000101111111100100101001001
11101001101000000011011001110101100111110110000001001110010
00110011001111010101101000010011011001001110110000111010110
00010110110001101001001111000101010110101010011101100001110
10110000101101111100101100111101010000001110001111001100001
01001000100100111110011110010101011010100111001100000011110
01010011101101110100100001010011001010100100111111100100010
10011101111110001000010010101001101000010010101101111010101
10010111000110100100011011010100110100001010011101001111010
01111000011011110010100000101010101011111011100100110111011
10000100001001010101011010001010101011010011100010001100011
11010111111000001101011011100110011010100101000101000100000
11010111110110100101010111011011011100101000110000101000010
01101011010001101011101110011110010101000111110010010001001
10110011110000011101010001110010110000110101000100101101111
01100101111010110100101000100101000110011001110010111000110
01101000111011111001010011101010001000100100010001111111010
00100101000111110111010101110011000010101100010001001111110
10010101100111010000101011010010011010100011001011

Table 4. (*continues*)

Seed Number 367
10001110001111011001000111010101000001001010011110011001101
00111110111001100110110000100011110001101011100111101001000
01010101010011011011101001111000010000110100101110100101100
10010001101010111110110001101111110001000010001000000100110
01110111101110011000111010011001011010000100010110111101001
00101010110101101011001010000011001110100101111000101101100
10011100000010101010011111001111100000001011101001010010111
10111001111110100111001101010000000011110100010001001011111
00010111010100010010011011000011011110011001111001101010110
00001100110110100101011100011011010100101011000101111010100
00010010001111101011010000011110001111101010000011100010110
11000110010110100111001111010100000111010111001000110110100
10110001110000111001111000101101110011010001100110100010101
10110010011011000010111001011001010111011010100010011111000
00101010001100100011111001010010111011001011100011100101010
01001100011100001101011000110000011111010111010011010000100
00011011111110101101001100010100010111011100100110110100011
00100011010001001111110001011100101110010100100101011110100
00100111101101011100010001100101000110011111101010111000000
00011011100001010111111100100100100001010111011010111110100
10000101010001100011001111011001011000110000101111011110000
11011000010011111001001000011110100101100100011000011010111
11100011000111001011100011010010001101101010011010111011001
11110101010000011000110001000000010001111011111110011

Table 4. Seed numbers 187 and 367 selected for our PRNG.

3.4 Results and Comparison with Mersenne Twister

We test our generator using Dieharder and BigCrush of the TestU01 test suites. As already mentioned, the Dieharder test suite contains 114 tests, whereas BigCrush of TestU01 contains the 106 most stringent tests. For comparison, we consider MT19937, SFMT19937 and TinyMT for 32 bits with their default seeds. The results are discussed in the following.

3.4.1 Dieharder

The desirable file size for a Dieharder test is usually more than 10GB. However, for testing our PRNG (on seed numbers 187 and 367) and to get a meaningful comparison, we use two file sizes of 1.2GB and 12GB. We can observe in Table 5 that, for all the generators, not even a single test fails on Dieharder over 12GB file size; hence, we use a smaller file size to compare the performance. It can be noticed that for this smaller file size of 1.2GB, our PRNG outperforms all its competitors for both the seeds. Since seed number 367 performs better between these two seeds, we choose it as our default seed to perform the remaining tests.

	CA (150′)+tempering				Competitors					
	Seed 187		Seed 367		MT19937		TinyMT		SFMT19937	
File Size	Weak	Failed	Weak	Failed	Weak	Failed	Weak	Failed	Weak	Failed
1.2GB	7	2	6	1	5	3	5	2	7	5
12GB	4	0	2	0	2	0	3	0	2	0

Table 5. Comparison using Dieharder.

3.4.2 BigCrush of the TestU01 Library

Table 6 reports on the tests that failed on the BigCrush of TestU01 library for our PRNG (with default seed 367) and its competitors. Here, for each of the failed tests, the p-value is either eps, meaning a value $< e^{-300}$, or 1-eps1, which means a value $< e^{-15}$. Over BigCrush, our PRNG performs more poorly than MT19937, SFMT19937 failing four out of 106 tests, whereas these variants of the MT fail on only two tests. These four tests are basically the Matrix Rank and LinearComp tests [6], which detect elements of linearity in the generated sequence. Therefore, the failure of our PRNG on these two tests may be due to its inherent linearity. To address this, we add nonlinearity in our generator to further improve its randomness quality.

https://doi.org/10.25088/ComplexSystems.32.2.139

Our PRNG		MT19937 and SFMT19937		TinyMT			
Test Number	Test Name	Test Number	Test Name	Test Number	Test Name	Test Number	Test Name
70	Matrix Rank, $r = 15$, $s = 15$	80	LinearComp, $r = 0$	8	CollisionOver, $t = 7$	69	Matrix Rank, $L = 1000$, $r = 26$
71	Matrix Rank, $r = 0$, $s = 30$	81	LinearComp, $r = 29$	10	CollisionOver, $t = 14$	70	Matrix Rank, $r = 15$, $s = 15$
80	LinearComp, $r = 0$			12	CollisionOver, $t = 21$	71	Matrix Rank, $r = 0$, $s = 30$
81	LinearComp, $r = 29$			19	BirthdaySpacings, $t = 8$	81	LinearComp, $r = 29$
				21	BirthdaySpacings, $t = 8$	87	LongestHeadRun, $r = 27$
				27	SimpPoker, $r = 27$	102	Run of bits, $r = 27$
				58	AppearanceSpacings, $r = 27$		

Table 6. Comparison using BigCrush of TestU01. Here, only failed tests are shown.

4. Phase II: Improving Our Pseudorandom Number Generator by Nonlinearity

Empirical data from the previous section suggests that our PRNG has an element of linearity, which makes the sequence less random and more susceptible to attack. This section explores some techniques to add a nonlinear module to our generator and compare the performances. We take two main approaches:

1. Using procedure Twist of MT19937. There are two ways to incorporate this into our generator:
   (a) Twist on output.
   (b) Twist on CA configuration.
2. Using a basic nonlinear component, such as XOR followed by Multiply, performed after Twist on the output.

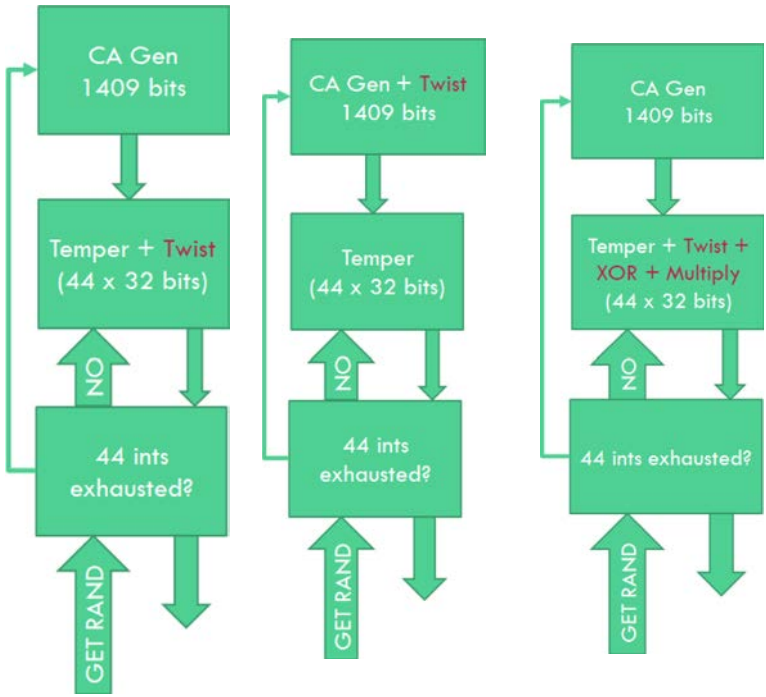
4.1 Using Twist

MT19937 uses Twist as the function to generate the next state of its 623-word state. Even though Twist produces the 623-word state in a single iteration, it does so by manipulating the individual words one at a time. We exploit this fact to design our two approaches to incorporate Twist in our generator:

- *On output*. Instead of directly generating output after applying Temper, we apply Temper followed by Twist to generate the output (see Figure 4(a)).
- *On CA configuration*. This is similar in principle to what MT19937 does. We first generate the next configuration of the CA and then apply Twist to generate the final form of the PRNG state. Then we apply Temper to extract individual numbers (see Figure 4(b)).

The generated PRNGs are once again tested using the default seed 367. Table 7 depicts the results, showing that our generator with Temper+Twist on Output passes all Dieharder tests, beating the others at both input sizes 1.2GB and 12GB. However, there is no improvement for these generators in the BigCrush test results. Therefore, the Twist function is not sufficient to remove the element of linearity from our PRNG. Next, we explore another method to add nonlinearity.

https://doi.org/10.25088/ComplexSystems.32.2.139



(a) Twist on output (b) Twist on configuration (c) Using XOR + Multiply

Figure 4. Applying nonlinearity to our PRNG.

Diehard								BigCrush on Both	
Twist on Output				Twist on Configuration				#	Test Name
1.2 GB		12 GB		1.2 GB		12 GB			
Weak	Failed	Weak	Failed	Weak	Failed	Weak	Failed	70	Matrix Rank, $r = 15, s = 15$
4	1	1	0	9	3	2	0	71	Matrix Rank, $r = 0, s = 30$
								80	LinearComp, $r = 0$
								80	LinearComp, $r = 29$

Table 7. Results after applying Twist on Diehard and BigCrush of the TestU01 library.

4.2 XOR + Multiply

To add nonlinearity in this approach, XOR followed by the multiplication operation on the output with numbers close to Sophie Germain primes is used over numbers generated via the Temper+Twist on Output method. Figure 4(c) shows the flowchart.

However, unfortunately, there is also no improvement in performance in BigCrush. The same tests fail again (likewise Table 7). At this point, our PRNG has become much slower by getting overloaded

with costly operations like multiplications along with Twist and Temper. Hence, we choose not to use this approach. In fact, we propose to select Temper+Twist on Output on CA (150′) (Figure 4(a)) with seed 367 as the default seed as our PRNG for applications where nonrandomness due to linearity can be ignored. As per Conjecture 2 of [8], this PRNG has a period length of $2^{1409} - 1$ and it passes all Dieharder tests.

Nevertheless, our generator still cannot pass the Matrix Rank and LinearComp tests of BigCrush. Further addition of any nonlinear module to the generator is impractical as it hampers the efficiency and speed. Therefore, we employ a totally different approach. As the root cause of the problem may be the linearity of the CA rules, in the next section we proceed to use a nonlinear CA.

5. Phase III: Search for a Nonlinear Maximal-Length Cellular Automaton

Until now, we have used a linear CA (recall that CA (150′) contains only rules 90 and 150, both of which are linear rules) and externally added nonlinearity. But this has not produced the desired result. So in this section, instead of using a linear CA and adding nonlinear components after it, we integrate nonlinearity inside the CA. However, the challenge is finding an appropriate nonlinear CA that satisfies the maximal-length property. For this, we use some existing theories from [12, 13].

5.1 Greedy Strategy for Constructing a Potential Nonlinear Maximal-Length Cellular Automaton

In [13], a technique is given to formulate a nonlinear CA from a known linear maximal-length CA that has the potential to be a maximal-length CA. In this section, we use those strategies to create such a potential nonlinear CA based on our selected CA (150′) by changing only some of its rules. We know that maximal-length CAs are always reversible. Hence, our first target is to ensure that this condition is satisfied by the synthesized CA.

5.1.1 Synthesis of a Reversible Cellular Automaton

To synthesize an n-cell reversible CA based on CA (150′), the methodology of [12] is used. Here, all the balanced rules are divided into six classes. Then, a rule-class relationship table is given that guides which rules can be chosen as the rule for cell $i + 1$ from the class information of the rule of cell i. For the sake of completeness, we reproduce

Table 8, which depicts the class information of the participating rule $\mathcal{R}_i$, $0 \le i \le n-1$. Using this table, we update some of the cells' rules (having rule 150 by default) of our CA to nonlinear rules such that the obtained CA is always reversible. Table 9 shows such a CA for $n = 32$.

(a) Class relationship of $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$.

Class of $\mathcal{R}_i$	$\mathcal{R}_i$	Class of $\mathcal{R}_{i+1}$
I	51, 204, 60, 195	I
	85, 90, 165, 170	II
	102, 105, 150, 153	III
	53, 58, 83, 92, 163, 172, 197, 202	IV
	54, 57, 99, 108, 147, 156, 198, 201	V
	86, 89, 101, 106, 149, 154, 166, 169	VI
II	15, 30, 45, 60, 75, 90, 105, 120, 135, 150, 165, 180, 195, 210, 225, 240	I
III	51, 204, 15, 240	I
	85, 105, 150, 170	II
	90, 102, 153, 165	III
	23, 43, 77, 113, 142, 178, 212, 232	IV
	27, 39, 78, 114, 141, 177, 216, 228	V
	86, 89, 101, 106, 149, 154, 166, 169	VI
IV	60, 195	I
	90, 165	IV
	105, 150	V
V	51, 204	I
	85, 170	II
	102, 153	III
	86, 89, 90, 101, 105, 106, 149, 150, 154, 165, 166, 169	VI
VI	15, 240	I
	105, 150	IV
	90, 165	V

(b) First rule table.

Rules for $\mathcal{R}_0$	Class of $\mathcal{R}_1$
3, 12	I
5, 10	II
6, 9	III

https://doi.org/10.25088/ComplexSystems.32.2.139

(c) Last rule table.

Rule Class for $\mathcal{R}_{n-1}$	Rule Set for $\mathcal{R}_{n-1}$
I	17, 20, 65, 68
II	5, 20, 65, 80
III	5,17, 68, 80
IV	20, 65
V	17, 68
VI	5, 80

Table 8. Rules to generate a reversible CA.

5.1.2 No Blocking Word

In a maximal-length CA, all configurations except one are reachable from one another. This indicates that the CA does not have any *blocking word* in it.

Definition 7. A subconfiguration $s = (s_i)_{i\in[j,j+k-1]}$ of a configuration $c = (s_i)_{0\le i<n}$ is a blocking word if for any $t \ge 1$, $G_n^t(c)\,|_i = s_i$ for each $i \in [j, j+k-1]$, but $G_n^t(c)\,|_l \ne s_l$ for some $l \notin [j, j+k-1]$. Here G_n is the global transition function of the n-cell CA and $0 \le l < n$ [13].

To illustrate a blocking word, let us consider a CA (10, 90, 172, 150, 204, 20). For this CA, the subconfiguration 111 is a blocking word when it is applied to the second through fourth cells. The chain of configurations

$$001\,110 \to 011\,111 \to 111\,110 \to 101\,111 \to 001\,110$$

forms a cycle of length four. Existence of a blocking word indicates that the flow of information in the CA is blocked. Obviously, there is no information flow from the first two cells (and from the last cell) to other cells in this particular CA. The maximum possible cycle length that can be formed by these configurations is bounded by 2^{n-k}, where k is the length of the blocking word (here $n = 6$ and $k = 3$). Further, a configuration having a blocking word is always unreachable from any configuration that does not have that particular word. For example, the configuration 001010 cannot be reached from 001110 of the given CA. Therefore, existence of such a blocking word ensures that the CA is not a maximal-length CA.

5.1.3 Unique Single-Length Cycle

Once we ensure that our CA has no blocking word, we need to check for the existence of a unique single-length cycle.

Cell number (i)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R_i	10	150	150	150	150	**86**	150	150	150	150	150	**149**	150	150	150	150
Class	2	1	3	2	1	6	4	5	6	4	5	6	4	5	6	4
Cell number (i)	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R_i	150	**101**	150	150	150	150	150	150	150	150	150	150	150	**154**	150	20
Class	5	6	4	5	6	4	5	6	4	5	6	4	5	6	4	

Table 9. A sample nonlinear (potential) maximal-length CA following Table 8 based on CA (150′). Nonlinear rules appear in bold.

Definition 8. A CA G_n forms a single-length cycle for a configuration x if $G_n(x) = x$.

We can imagine a single-length cycle as a full-length blocking word. In the case of a maximal-length CA, there is only one single-length cycle. Therefore, if a CA has no or more than one single-length cycle, then the CA is not a maximal-length CA [13].

https://doi.org/10.25088/ComplexSystems.32.2.139

Now we move on to develop our nonlinear CA that satisfies these conditions. The following can be observed from Table 8:

1. The rule vector can have at most three consecutive nonlinear rules.

2. The rule vector can have at most $(\lfloor (n-5)/2 \rfloor) + 3$ nonlinear rules, and the rest of the rules are chosen from the set of four rules: 90, 150, 105 and 165, where $n \geq 5$. For our case, the first and last rules are 10 and 20, respectively (which are equivalent to applying rule 90 at the first cell and rule 150 at the last cell for the null boundary condition), and the rest of the rules are rule 150.

Hence, while synthesizing, we consider these two criteria. Note that it is highly likely that the CA developed in this way is of maximal length, although there is no guarantee. An example of such a nonlinear CA based upon CA (150′) is given in Table 9. Here, only four nonlinear rules are *injected* into the base CA to make it nonlinear.

5.2 Constructing Our Nonlinear Cellular Automaton

Our target is to find a nonlinear maximal-length CA of size 1409 that has a good randomness quality. As there is no known foolproof method to find such a CA, we take the greedy strategy mentioned in the previous section. That means we take our CA (150′) as the base CA and following Conditions 1 and 2, form a nonlinear CA using Table 8 such that the CA is reversible, has no blocking word and has a unique single-length cycle. However, there is no theoretical measure known to check whether the CA synthesized in this way is of maximal length. The only way is to run the n-cell CA and see whether it really has a cycle of length 2^{n-1}.

Now, with current computational resources, checking the result on a length of 1409 is an impossible task. So, we take another greedy approach. Our strategy is to synthesize 32-bit CAs following the aforementioned conditions and test whether these CAs have a good randomness quality (in terms of performance in the Dieharder testbed). Then we choose the ones that give the best result to be repeated until length 1409. However, this cannot be a blind repetition of 32 bits, but rather we have to be careful while applying the nonlinear rules such that the conditions of reversibility, no blocking word and unique single-length cycle are not violated. To maintain the simplicity of code and computation, we take the occurrence of such nonlinear rules only at specific intervals that are multiples of six, though the starting point of this pattern may vary.

While selecting a particular pattern and inserting a set of combinations of nonlinear rules, each rule needs to be checked for whether it is producing a blocking word in the CA. If the pattern does not give a blocking word, it is then tested for the presence of a unique single-

length cycle. Different combinations of nonlinear rules of length 32 that are both nonblocking and give one single-length cycle are put to the test on Dieharder. Out of all the CAs that are tested, a set of six CAs that performed the best are shown in Table 10. We use these patterns and extend them to a length of 1409. For example, in the fourth row of Table 10), the CA has $R_0 = 10$, $R_5 = R_{35} = R_{65} = \cdots = R_{1385} = 86$, $R_{11} = R_{41} = \cdots = R_{1391} = 149$, $R_{17} = R_{47} = \cdots = R_{1397} = 101$, $R_{23} = R_{53} = \cdots = R_{1403} = 169$, $R_{29} = R_{59} = \cdots = R_{1379} = 154$, $R_{1408} = 20$ and $R_i = 150$ for all other i. These CAs have a good chance of being a nonlinear maximal-length CA, though there is no guarantee.

Nonlinear Rules	Positions of Insertion in Sequence	Failed Test Results
86, 149, 101, 154, 169	regular interval of six cells except the fourth and fifth, which are applied in an interval of 12	70 and 71
86, 149, 101, 154, 169	regular interval of six cells	70 and 71
86, 149, 101,169, 154	regular interval of six cells	71
86, 149, 101, 154	regular interval of six cells	71
86, 149, 101, 154	regular interval of six cells except the last rule, which is applied after 12 cells	71
154, 86, 86, 86, 86	regular interval of six cells	71

Table 10. BigCrush test results for different nonlinear CAs synthesized based on CA (150′).

To choose a seed for this new generator, we once again follow the same procedure of Section 4. Table 11 shows the best-performing seed for these nonlinear CA-based generators. The new generator works the same way as the linear CA-based PRNG developed in Section 3. That is, each configuration produces 44 32-bit numbers that are tempered using the same tempering function. We then put them to rigorous testing on Dieharder, NIST and BigCrush of the TestU01 library using this seed. If not otherwise mentioned, this seed is set as the default for the nonlinear PRNG and the seed number 367 (Table 4) as the default for the PRNGs of Sections 3 and 4.

https://doi.org/10.25088/ComplexSystems.32.2.139

11100111100110000110101010001100110111000101001110101001100
01101011100010011000111011000000111110011010001100100011010
00100111111010100000111000111001011101111000111000100110100
01101010011110011011101010111001100001010110001011110011001
00110111011000000010101001111101001001010010111110000011100
11111000010000110011011001001011110001011100111001010011000
00011010010111100000101001111111000111010111101101001000110
00011010000000111100010011100111110100111100010101001111010
00110001111011010001010101101010001110010010110000010101011
01011010010111101110110100011111000001000110010110010011101
11000100111001100010001100001101011111000110001110111101000
01101101100000110100101101101001111011010101000000110101001
01101000001011000110010111111011010011011000101101010100101
01110111000100010000011111010101011111111110001000010000101
01011000100011000101101001011100101101110011000000110100010
01111100111110001101101111110000101011100000000010111110011
10100100101110001100111011011000100000110001110111011101001
01000101001101001011101111101010001011100001000110010111101
00001110101111000010011001011011111101011100000001010101000
11100001011001100110011011000101101111010000101110010100011
01001100101000110011111101010111000000100010100000111100101
11000111110111000001101000111111100100110011010101100101110
01100100011000101110011111100110000000110011100111000001101
00011110100111001100100100111111000000101010111111000

Table 11. Selected seed for nonlinear PRNGs of Table 10.

5.3 Results and Comparison

We first test our PRNG with the Dieharder testbed and see that it also passes all tests for 1.2GB; we do not include this result as a table. Next, the NIST test is applied to all versions of our PRNGs developed so far and the MT family. Table 12 reports these results.

PRNG	Failed Test
MT19937	NIL
SFMT19937	NIL
TinyMT	rank test
CA (150′) + Temper	rank test
CA (150′) + Temper + Twist (both approaches)	rank test
CA (150′) + Temper + Twist + XOR + Multiply 1 (Rank Test)	rank test
Nonlinear PRNGs	NIL

Table 12. Consolidated results on NIST for all PRNGs.

In Table 12, we can observe that TinyMT and the previous versions of PRNG fail the rank test of the NIST statistical test suite. However, our final nonlinear PRNG as well as MT19937 and SFMT19937 pass all tests of NIST. Therefore, this new PRNG is on par with the MT family with respect to NIST.

Finally, we apply the BigCrush test on these new generators. This result is depicted in Table 10. Here, test numbers 70 and 71 are Matrix Rank, with $r = 15$, $s = 15$ and $r = 0$, $s = 30$, respectively. We can observe that except for the first two CAs, the next four CAs pass all BigCrush tests except test number 71. This is even better than the performance of all versions of the MT family (see Table 13 for the detailed comparison of results). Among these four CAs, we propose to choose the CA with nonlinear rules 86, 149, 101, 154 applied at a regular interval of six cells as our default PRNG. An implementation of this PRNG along with the linear versions is publicly available in GitHub [14].

https://doi.org/10.25088/ComplexSystems.32.2.139

Our Final Nonlinear PRNG		MT19937 and SFMT19937		TinyMT			
Test Number	Test Name	Test Number	Test Name	Test Number	Test Name	Test Number	Test Name
71	Matrix Rank, $r = 0, s = 30$	80	LinearComp, $r = 0$	8	CollisionOver, $t = 7$	69	Matrix Rank, $L = 1000, r = 26$
		81	LinearComp, r=29	10	CollisionOver, $t = 14$	70	Matrix Rank, $r = 15, s = 15$
				12	CollisionOver, $t = 21$	71	Matrix Rank, $r = 0, s = 30$
				19	BirthdaySpacings, $t = 8$	81	LinearComp, $r = 29$
				21	BirthdaySpacings, $t = 8$	87	LongestHeadRun, $r = 27$
				27	SimpPoker, $r = 27$	102	Run of bits, $r = 27$
				58	AppearanceSpacings, $r = 27$		

Table 13. Updated comparison table using BigCrush of TestU01. Only failed tests are shown.

It can be easily concluded from these tables that our nonlinear CA-based PRNG outperforms all its competitors: MT19937, TinyMT and SFMT19937 on TestU01. It also performs equal to or better than the MT family on Dieharder and NIST, in accordance with which, we can say that our PRNG is the best PRNG among all.

6. Conclusions and Next Steps

Our target in this paper is to see if simple two-state three-neighborhood cellular automata (CAs) that have only local interaction can outperform the strongest pseudorandom number generator (PRNG) like the Mersenne Twister (MT) family. The approach we have taken is to work in the same way as the MT and see how far we can go. We started with a very simple, almost uniform cellular automaton (CA), known as CA (150′) and developed PRNGs using it over cell length $n = 1409$ (i.e., $32 \times 44 + 1$). That means, similar to the MT, our generators also produce a group of numbers (here 44 32-bit numbers) together with only one bit of wastage per configuration. We have applied the same tempering function of MT19937 to scatter the numbers generated as part of a CA configuration. We have observed that if we apply a Twist function like MT19937, our CA (150′)-based PRNG can pass all Dieharder tests, which is even better than its peers MT19937 and SFMT19937.

However, this version of the PRNG fails some BigCrush tests that detect nonlinearity in the system. To address this, we move on to inject nonlinear rules into our base CA (150′), making it nonlinear while preserving some conditions such that it has the potential to be a nonlinear maximal-length CA. On this newly synthesized nonlinear CA, we apply the tempering function of MT19937 to develop our final PRNG. This PRNG also produces 44 32-bit numbers from a single configuration. This PRNG was tested with the default seed for the PRNG on Dieharder, NIST and TestU01 of the BigCrush library. It can be seen that our PRNG beats the MT and all its peers in terms of randomness quality and simplicity, passing every test of Dieharder and NIST with only one failed test in BigCrush.

This shows that such a simple CA can also beat the strongest PRNG developed so far. Also, it shows the power of nonlinear maximal-length CAs even if only a tiny amount of nonlinearity is added to the base linear maximal-length CA. Since we have not been able to completely explore the vast world of possible nonlinear CAs, we believe some nonlinear maximal-length CAs may be identified that can pass all empirical testbeds even without the need of tempering.

https://doi.org/10.25088/ComplexSystems.32.2.139

Furthermore, many other CAs similar to CA (150′) can be investigated (e.g., CA (90′) of [8]), which may have a similar quality. As bitwise operations are possible for these simple CAs, work may be directed to use them as parallel random number generators, which the current parallel computing environment is in dire need of.

Acknowledgments

This work is partially supported by Start-up Research Grant (File number: SRG/2022/002098), SERB, Department of Science & Technology, Government of India and NIT, Tiruchirappalli SEED Grant. We are grateful to Prof. Sukanta Das for his continuous encouragement, valuable comments, guidance and support for completing this work.

References

[1] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator," *ACM Transactions on Modeling and Computer Simulation*, **8**(1), 1998 pp. 3–30. doi:10.1145/272991.272995.

[2] M. Saito and M. Matsumoto, "SIMD-Oriented Fast Mersenne Twister: A 128-bit Pseudorandom Number Generator," *Monte Carlo and Quasi-Monte Carlo Methods 2006* (A. Keller, S. Heinrich and H. Niederreiter, eds.), Berlin, Heidelberg: Springer-Verlag, 2008 pp. 607–622. doi:10.1007/978-3-540-74496-2_36.

[3] K. Bhattacharjee and S. Das, "A Search for Good Pseudo-random Number Generators: Survey and Empirical Studies," *Computer Science Review*, **45**, 2022 100471. doi:10.1016/j.cosrev.2022.100471.

[4] M. Saito and M Matsumoto, "A High Quality Pseudorandom Number Generator with Small Internal State," *IPSJ SIG Notes*, **3**, 2011 pp. 1–6.

[5] R. G. Brown, D. Eddelbuettel and D. Bauer. "Dieharder: A Random Number Test Suite." (Jul 11, 2023) webhome.phy.duke.edu/~rgb/General/dieharder.php.

[6] P. L'Ecuyer and R. Simard, "TestU01: A C Library for Empirical Testing of Random Number Generators," *ACM Transactions on Mathematical Software*, **33**(4), 2007 pp. 1–40. doi:10.1145/1268776.1268777.

[7] S. Wolfram, "Origins of Randomness in Physical Systems," *Physical Review Letters*, **55**(5), 1985 pp. 449–452. doi:10.1103/PhysRevLett.55.449.

[8] S. Adak and S. Das, "(Imperfect) Strategies to Generate Primitive Polynomials over GF(2)," *Theoretical Computer Science*, **872**, 2021 pp. 79–96. doi:10.1016/j.tcs.2021.03.007.

[9] K. Bhattacharjee, N. More, S. K. Singh and N. Verma, "Emulating Mersenne Twister with Cellular Automata," in *Proceedings of First Asian Symposium on Cellular Automata Technology*, Kolkata, India (S. Das and G. J. Martinez, eds.), Singapore: Springer Nature, 2022 pp. 95–108. doi:10.1007/978-981-19-0542-1_8.

[10] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, et al., *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, Gaithersburg, MD: National Institute of Standards and Technology, U.S. Department of Commerce, 2010. csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final.

[11] G. Marsaglia. "DIEHARD: A Battery of Tests of Randomness." (Jul 11, 2023) stat.fsu.edu.

[12] S. Das, "Theory and Application of Nonlinear Cellular Automata in VLSI Design," Ph.D. thesis, Bengal Engineering and Science University, Shibpur, India, 2007.

[13] S. Adak, "Maximal Length Cellular Automata," Ph.D. thesis, Indian Institute of Engineering Science and Technology, Shibpur, India, 2021.

[14] K. Bhattacharjee and S. Das. "PRNG Library." (Jul 11, 2023) github.com/kamalikaB/PRNG-library/tree/main/CA/MTbyCA150'.