The Mersenne Twister

B10607011, Zhi-Yu Huang

July 1, 2023

1 Introduction of the Mersenne Twister

The Mersenne Twister is a **pseudorandom number generator (PRNG) algorithm**, meaning that though it seems like it is generating random numbers, it is actually using **a deterministic way to generate the sequence of numbers**. Different from older random number generators, such as linear congruential generators, the Mersenne Twister is known for its **high efficiency, long period sequence**, and most importantly, **high-quality random number generation**. It is efficient for computer computations since it is based on several **linear recurrence equations**, and instead of using simple arithmetic such as addition and multiplication, it uses bitwise operations such as **bit-shifts, and's, or's** to generate pseudorandom numbers.

To explain what Mersenne Twister is, we first want to understand the concept of Mersenne primes. Mersenne primes are prime numbers in the form of $2^p - 1$, where p is a prime number (note that not all prime numbers p could produce a Mersenne prime). In fact, the name "Mersenne" refers to the fact that **the period of the algorithm is based on Mersenne primes**; It has a period of $2^{19937} - 1$, meaning that **it generates a sequence of** $2^{19937} - 1$ **random numbers before repeating itself**. This makes the Mersenne Twister an excellent algorithm for **applications that may require a large number of random values**.

2 How it Works

2.1 Initialization

The algorithm starts with initialization. The algorithm takes a **seed value**, which is typically an integer, and initializes its internal state. The internal state is represented by an array containing **624 integers with 32-bit each**, which is the **current state** of the algorithm. The internal state is for **maintaining the sequence** of random numbers. The algorithm will perform operations on this state to produce the next random number in the sequence.

2.2 State Update

A state update comes into play. Once all 624 integers in the internal state array have been utilized, the Mersenne Twister algorithm performs a special operation known as **the "twist" operation**. Each time a random number is generated, the internal state is updated, and the subsequent random number will depend on the new state. The algorithm iteratively updates its internal state to generate random numbers using a **recurrence relation** that involves a series of **bitwise operations**. The produced sequence of random numbers follows a specific **statistical distribution** to make the pseudorandomness even more random. It is worth mentioning that the Mersenne Twister passes almost all statistical randomness tests.

2.3 Number Extraction and Iteration

The algorithm performs number extraction. The algorithm takes the current state and applies **additional mathematical operations** to generate a random number to **extract a 32-bit random value**. This extracted value is **the output of the random number generation process**. Last but not least, we **iterate the process**. As we mentioned before, the Mersenne Twister has a very **long period of** $2^{19937} - 1$.

3 Reflections on the Videos

3.1 NMCS4ALL: Random Number Generators [link]

The author, Dave Ackley, first introduced a PRNG algorithm using a **linear congruential generator (LCG)**. The LCG uses a **recurrence relation** expressed by the formula $x_{i+1} = (ax_i + b) \bmod c$, where x_i represents the current state and a, b, and c are constants.

However, in certain applications, such as encryption, it is crucial to have **a random sequence that is completely unpredictable**. This is where the Mersenne Twister comes into play. The algorithm continues to generate a long sequence of random numbers with desirable statistical properties.

I learned from the video that randomness can be evaluated based on two factors: **predictability and correlation**. Predictability refers to the degree to which **future numbers can be determined from previous ones**, while correlation measures the **relationship between consecutive numbers**. Depending on **the level of randomness needed**, one must determine whether an unpredictable or uncorrelated algorithm is more suitable.

3.2 How To Predict Random Numbers Generated By A Computer [link]

The author, PwnFunction, taught us how, according to the rules provided by documents and research on generating functions (namely the resources from the V8 team's blog on Math.random and the research paper on xorshiftplus.), we could use **the number array generated previously** to **predict the number coming up next** by implementing the operation rules. **I personally conducted a demonstration shown below** following the instructions in the video. The goal is that after **finding the state values**, we use them to **perform operations and predict future numbers**. From the implementation, I got **a more concrete understanding of the algorithm** behind the pseudorandom number generators, including **bit-wise operation, linear recurrence, and more**. I felt like I really got the chance to have a better understanding by implementing all the steps.

3.3 Implementation of Random Numbers Predictor

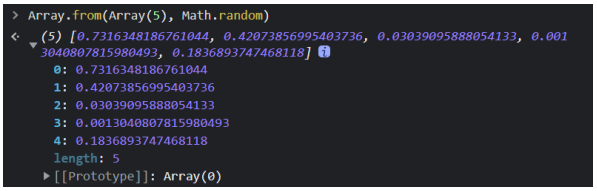

Figure 1: Generate five random numbers using Math.random

```
se_state0, se_state1 = z3.BitVecs("se_state0 se_state1", 64)

sequence = [
    0.7316348186761044,
    0.42073856995403736,
    0.03039095888054133,
    0.0013040807815980493,
    0.1836893747468118
  # 0.23137147109312428
][::-1]
```

Figure 2: Put this sequence of numbers into our code

```
{'se_state1': 3388470885014166747, 'se_state0': 11970025830218494813}
0.6488964004915247
```

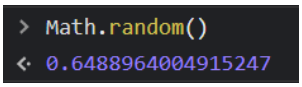

Figure 3: The generated result is 0.6488964, which is the same as when we run Math.random again!