An Algorithmic Random Integer Generator based on the Distribution of Prime Numbers

Bertrand Teguia Tabuguia

May 1, 2019

Abstract

We talk about random when it is not possible to determine a pattern on the observed outcomes. A computer follows a sequence of fixed instructions to give any of its output, hence the difficulty of choosing numbers randomly from algorithmic approaches. However, some algorithms based on mathematical formulas like the Linear Congruential algorithm and the Lagged Fibonacci generator appear to produce "true" random sequences to anyone who does not know the secret initial input [1]. Up to now, we cannot rigorously answer the question on the randomness of prime numbers [2, page 1] and this highlights a connection between random number generator and the distribution of primes. From [3] and [4] one sees that it is quite naive to expect good random reproduction with prime numbers. We are, however, interested in the properties underlying the distribution of prime numbers, which emerge as sufficient or insufficient arguments to conclude a proof by contradiction which tends to show that prime numbers are not randomly distributed. To achieve this end, we use prime gap sequence variation. The algorithm that we produce makes possible to deduce, in the case of a binary choice, a uniform behavior in the individual consecutive occurrence of primes, and no uniformity trait when the occurrences are taken collectively.

1 Introduction

The use of randomness is needed in almost all areas, including cryptography [5] and bioinformatics [6]. The latter article gives an alert about the use of built in random implementations. In general, we do not produce true randomness, since the approach is algorithmic; hence we more commonly use pseudo random number generator (PRNG). PRNGs are periodic, and larger periods give better random imitation; that is why a major consideration in the choice of a pseudo random number generator is the size of its period, because this directly affects the frequency that a generator can be used. Another common issue with PRNG algorithms is the seeding or the initialization, since this is actually the construction of a sequence following states, two sequences having the same initial state must be identical. Nevertheless a good seeding may result from the aim of the random implementation; for example, in P&C Game [7], the author uses the first click of the player to seed the generator.

The main core of a PRNG is the black box to test for acknowledgment of pseudo-randomness. Among the most well-known pseudo-random number generator is the linear congruential generator. The formula for the algorithm it uses is $s_{n+1} \equiv (a \cdot s_n + c)[m]$, where $\equiv \cdot[m]$ denotes the arithmetic modulo m. There is a lot of investigations in the choice of the integers a and c to get a good quality of that PRNG. The most popular PRNG used is the Mersenne Twister which relies on Feedback Shift Registers by generating numbers mathematically in terms of hardware. The procedure can be seen as follows: an even number of positions are selected, the generator operates by performing XOR on the bits at these positions, taking the result as the new leftmost bit, and shifting the rest of the string right by one [1].

Positive integers having only two divisors, 1 or itself. These numbers called primes apart from belief up to now, are not well understood. And that is our concern, because randomness can be seen as a missing of information and so the question is why do not we use our ignorance to produce ignorance? This gives the importance of such a study because with enough improvement we may say that either we have produced random or got a better understanding of the distribution of primes. However, Lemke Oliver and Soundararajan saw that in the first billion primes, a 1 is followed by a 1 about 18% of the time, by a 3 or a 7 each 30% of the time, and by a 9 22% of the time. They found similar results when they started with primes that ended in 3, 7 or 9: variation, but with repeated last digits the least common. The bias persists but slowly decreases as numbers get larger [3] [8]. Therefore, there is not much randomness as one imagines, and the expected results are more related to the understanding of the distribution of prime numbers.

From Zhang's Theorem [9], we know that there exists a bound $B < 7\cdot 10^7$ such that there are infinitely many integers pairs of prime numbers $p < q < p+B$. Some other works that we will not mention have been done in that direction. Our algorithm for random integer generator uses the variation of the prime gap sequence. Given two different prime gaps $g_i, g_j \leqslant B$, Zhang's Theorem allows us to say that there are infinitely many pairs of primes $p, p+g_i$ and $p, p+g_j$. So we can say that the prime gap progress is not monotone, because g_i and g_j can certainly appear in decreasing and increasing order. Although Zhang result does not prove the twin prime conjecture [10], these works ensure altogether the consistency of an algorithm to produce "random" behavior relying on the variation of the prime gap sequence.

In the sequel, after a brief presentation of a PRNG structure we will be giving details of our algorithm according to the given PRNG description. That is to define what we call first left minimum function (flm), and the update states. We end with some tests of pseudo-randomness.

2 Structure of Pseudo Random Integer Generator

Typically, there is a set S of states, and a function $f\colon S \longrightarrow S$ for state update (see [11, chapter 3]). There is an output space O and function $h\colon S \longrightarrow O$. Usually the output space is taken to be the interval $(0,1)$, but that correspond to a generator of real number. In our case however, we consider any finite set of labels or integers for simplicity. After choosing the seed, the sequence of random integers are generated as follows

$$\begin{aligned} S_n &= f(S_{n-1}), \quad n = 2,3,4,\ldots \\ O_n &= h(S_n) \end{aligned} \tag{1}$$

3 First Left Minimum (flm) Function

Let $G = (g_1, g_2, \ldots, g_n) \in \mathbb{E}^n$, where $\mathbb{E}$ is a non empty ordered set of objects. We define the first left minimum of G as

$$\mathrm{flm}_n(G) = \mathrm{flm}_n(g_1, g_2, \ldots, g_n) = \begin{cases} g_1 & \text{if } g_1 < g_2 \\ g_2 & \text{if } g_2 < g_3 \text{ and } g_2 \leqslant g_1 \\ g_3 & \text{if } g_3 < g_4 \text{ and } g_3 \leqslant g_2 \leqslant g_1 \\ \ldots \\ g_{n-1} & \text{if } g_{n-1} < g_n \text{ and } g_{n-1} \leqslant g_{n-2} \leqslant \ldots \leqslant g_1 \\ g_n & \text{if } g_n \leqslant g_{n-1} \leqslant g_{n-2} \leqslant \ldots \leqslant g_1 \end{cases} . \quad (2)$$

Indeed, it is the first left strict minimum value between two consecutive components of G starting on the left; with the particularity that in the worst case where there is no left strict minimum, the output is the last component.

Next, we show that, in any situation, flm gives an output.

Proof. Let $\mathbb{E}$ be a non empty ordered set of objects and $N \in \mathbb{N}$. We take $G = (g_1, g_2, \ldots, g_N) \in \mathbb{E}^N$. Let $g_j, 1 \leqslant j < N$ be the element at the j^{th} position in the tuple G. We assume that $g_1 \geqslant g_2 \geqslant \ldots \geqslant g_j$, and thus we have to consider two cases:

- $g_j < g_{j+1}$, then $\mathrm{flm}_N(G) = g_j$;
- $g_j \geqslant g_{j+1}$, then $\mathrm{flm}_N(G) \in \{g_{j+1}, g_{j+2}, \ldots, g_N\}$.

This leads us either to the output g_j or to the same situation with g_{j+1}. So it only remains to make sure that flm_N gives an output when $j = N$. This is satisfied, because, when $j = N$, the output is $\mathrm{flm}_N(G) = g_N$ which appears to be a situation when the components of G are in decreasing (not necessarily strict) order from the left. □

Example: Let $\mathbb{E} = \mathbb{N}$, and consider $G_1 = (5, 4, 3, 6, 2), G_2 = (1, 3, 5, 4, 2)$, $G_3 = (8, 7, 5, 4, 2)$, and $G_4 = (4, 4, 3, 2, 5)$ Then we have

$$\begin{aligned} \mathrm{flm}_5(G_1) &= 3, \\ \mathrm{flm}_5(G_2) &= 1, \\ \mathrm{flm}_5(G_3) &= 2, \\ \mathrm{flm}_5(G_4) &= 2. \end{aligned} \quad (3)$$

From the definition of an flm function, one can easily deduce the following properties.

Proposition 1. *Let $\mathbb{E}$ be a non empty ordered set. We have*

a)

$$\mathrm{flm}_n : \mathbb{E}^n \longrightarrow \mathbb{E}$$
$$G \mapsto \mathrm{flm}_n(G).$$

b) $\mathrm{flm}_2 = \min$.

c) *Let* $G = (g_1, g_2, \dots, g_n) \in \mathbb{E}^n$,

$$\begin{aligned}\mathrm{flm}_n(G) &= \min(g_1,g_2)\delta_{g_1,\min(g_1,g_2)}\delta_{g_1,g_2} \\ &+ \delta_{g_2,\min(g_1,g_2)}\Bigg(\min(g_2,g_3)\delta_{g_2,\min(g_2,g_3)}\delta_{g_2,g_3} + \delta_{g_3,\min(g_2,g_3)}\bigg(\dots \\ &+ \delta_{g_{n-1},\min(g_{n-2},g_{n-1})}\left(\min(g_{n-1},g_n)\left(\delta_{g_{n-1},\min(g_{n-1},g_n)}\delta_{g_{n-1},g_n} + \delta_{g_n,\min(g_{n-1},g_n)}\right)\right)\dots\bigg)\Bigg)\end{aligned} \tag{4}$$

where min *return the minimum value of its arguments and* δ_{g_i,g_j} *denotes the Kronecker symbol defined as*

$$\delta_{g_i,g_j} = \begin{cases} 1 & \text{if } g_i = g_j \\ 0 & \text{otherwise} \end{cases}.$$

For c), if we take $G = (g_1, g_2, g_3)$ we have

$$flm_3(G) = \min(g_1,g_2)\delta_{g_1,\min(g_1,g_2)}\delta_{g_1,g_2} + \delta_{g_2,\min(g_1,g_2)}\min(g_2,g_3)\left(\delta_{g_2,\min(g_2,g_3)}\delta_{g_2,g_3} + \delta_{g_3,\min(g_2,g_3)}\right).$$

4 Definition of Our Pseudo Random Integer Generator

Instead of non-empty as before, here we need an infinite countable ordered set. And for simplicity we consider $\mathbb{E} = \mathbb{N}$ since in any case $\mathbb{E}$ can always be assimilated to a subset of $\mathbb{N}$ by bijection. Given $N \geqslant 2$ integers $k_1, k_2, \dots, k_N$ for a random choice among them, we consider the set of states as an N-tuple of N consecutive prime gaps. If we denote by σ a translation of the starting point in the primes set based on the seeding, then the set of states can be defined as

$$S = \left\{S_n = (g_{n+1}, g_{n+2}, \dots, g_{n+N}),\ g_j = p_{\sigma(j+1)} - p_{\sigma(j)},\ p_{\sigma(j)} \text{ primes},\ 1 \leqslant j - n \leqslant N, \right\}, \tag{5}$$

where n is a non negative integer.

The update state function is defined as the next ordered N-tuple of the prime gap sequence starting at $g_m = \mathrm{flm}_N(S_n),\ n+1 \leqslant m \leqslant n+N$; that is,

$$\begin{aligned} f\colon S &\longrightarrow S \\ S_n = (g_{n+1}, g_{n+2}, \dots, g_{n+N}) &\longmapsto f(S_n) = S_{n+1} = (g_m, g_{m+1}, \dots, g_{m+N-1}), \\ & \quad\ g_m = \mathrm{flm}_N(S_n),\ g_j = p_{\sigma(j+1)} - p_{\sigma(j)},\ 0 \leqslant j - m \leqslant N-1. \end{aligned} \tag{6}$$

Finally the output function is given by

$$\begin{aligned} h\colon S &\longrightarrow \{k_1, k_2, \dots, k_N\} \\ S_n &\longmapsto h(S_n) = k_{\mathrm{flmpos}(S_n)}, \end{aligned} \tag{7}$$

where flmpos(S_n), returns the index position, counted from the left, of the flm_N output applied to the tuple S_n. In fact, flmpos can be defined as in (2), with the difference that the output is the index of the output given by flm.

Remark: The seed impact is an important aspect to highlight, because it tells us how σ is chosen. Indeed, if for example, we have $\sigma(j) = j + 4$, then $g_1 = p_{\sigma(2)} - p_{\sigma(1)} = p_6 - p_5 = 13 - 11 = 2$ and $g_2 = 17 - 13 = 4$. In such a case, for $N = 2$, we obtain k_1 as the first output. Thus depending on the situation, one has to define a proper σ from the seed.

However, as the goodness of the algorithm also depends on the period, a question to answer is the one related to an estimation of the period. This of course relies on the prime number theorem which states the following

Theorem 4.1 (Prime Number Theorem [2]). *The number of primes less than a given integer n is*

$$(1 + \varepsilon_n)\frac{n}{\ln n}, \quad \lim_{n\to\infty} \varepsilon_n = 0. \tag{8}$$

Where ln *denotes the natural logarithm.*

Therefore, given the maximal integer reachable by the working programming language or software, one can estimate the period of our PRNG algorithm using the prime number theorem. Having such an estimation, extremity (max and min) behavior has to be defined to make sure the algorithm continues; the need is to define the gap between the maximum prime and 2.

Nevertheless, there are infinitely many primes, so as far as the system can go in the calculation of large prime numbers, as large as the period of the algorithm will be. But on the other hand, the apparent concern will be the speed that could reduce the largest possible period to the largest accessible in a short time.

Remark:

- If the twin primes conjecture is verified, then our algorithm will always be able to change the component choice in a state S_n, since twin primes give the smallest prime gap for high integers.
- More generally, any unstable behavior of primes can be used in this way to try randomness imitation. The gap variation is just an example, since we cannot be sure of its increasing, decreasing or constant behavior.

Next, we evaluate our algorithm. This may give us a probabilistic argument about the distribution of primes.

5 Application and Tests

For tests, we consider the two-labels case. This corresponds to Carole's behavior in the P&C Game for the Prime level [7]. We are going to generate a sequence of sequence of $\{0,1\}'s$ following our pseudo random integer generator algorithm and make a test on independence and uniformity as explained in [11, chapter 3]. Let n, d be two large enough integers. We generate n d-tuples of $\{0,1\}$ by our algorithm and check uniformly distribution with the χ^2-test.

For application, we seed the generator with the s^{th} prime number as the minimal prime of the initial gap, where s is taken randomly on a certain interval in the system used. For this purpose, simple codes to generate a csv test file, can be written in python 2.7 [12] as follows

- random code:

```
#BTrandom2.py file
#By Bertrand Teguia
#Random code for two labels (0,1) based on prime numbers distribution

from math import *
import random


def nextprime(p):
    if p > 2:
        value = p
        while True:
            i = 3
            value += 2
            q = int(floor(sqrt(value)))
            while i <= q and value % i:
                i += 2
            if i > q:
                break
        return value
    value = 3 if p==2 else 2
    return value



def nthprime(n):
    cpt=1
    p=2
    while cpt<n:
        p=nextprime(p)
        cpt+=1
    return p



def BTrandom2():
    global prime
    global gap
    prevprime = prime
    prime = nextprime(prime)
    prevgap = gap
    gap = prime - prevprime;
    if prevgap < gap:
        return 0
    else:
        return 1

# seeding
s = nthprime(random.randint(10000, 20000))
prime = nextprime(s)
gap = prime - s
```

- csv file creator code:

```
#BTrandom2_test.py file
#By Bertrand Teguia
#Random code csv generator of n d-tuple for BTrandom2


from BTrandom2 import *

import csv

d = 100
n = 100

lines = []
for i in range(n):
    row = []
    for j in range(d):
        row.append(BTrandom2())
    lines.append(row)

with open(’BTrandom2_test.csv’, ’w’) as writeFile:
    writer = csv.writer(writeFile)
```

```
        writer.writerows(lines)

    writeFile.close()
```

Running the file *BTrandom2_test.py* in the same folder with *BTrandom2.py* produces a csv file named *BTrandom2_test.csv*. We can thus generate as many files as we want.

5.1 Probability variation of the prime gap sequence

Before going through the χ^2-test, let us first give an estimation of the probability p that 1 appears. Thus we use the law of large numbers [13], so we consider the outcomes from our generator to be independent. As seen in our code, we consider $n = d = 100$. The law of large number tells us that an estimation of the expectation of the distribution followed by our generator, which is the expectation of distribution followed by the prime gap increasing behavior at any index, is the limit

$$E = \lim_{d\longrightarrow\infty} \frac{\sum_{j=1}^{d} x_j}{d}, \tag{9}$$

where x_j, $1 \leqslant j \leqslant d$ denote the observations of the trial processes of a row in our generated csv files. After computations from csv files we realized that the expectation oscillates between 0.48 and 0.57; and moreover, using again the law for the estimated expectations on each row gives us amazingly in all the cases the value 0.52.

Note that the second use of the law of large numbers adds the hypothesis that each row is taken independently, which is natural from the first independence hypothesis. Furthermore, due to the experience characteristics, one trivially sees that we are in the case of a Bernoulli scheme. Therefore, since the expectation of a Bernoulli experience is the winning probability we have our estimation; that is the probability that the prime gap sequence increases at any index is given by

$$p = E \approx 0.52, \tag{10}$$

or, rigorously, in the convenient probability space with the probability $\mathcal{P}$, given any three consecutive prime numbers

$$p_n < p_{n+1} < p_{n+2},\ n \in \mathbb{N},$$

we have

$$\mathcal{P}\left\{p_{n+2} - p_{n+1} \geqslant p_{n+1} - p_n\right\} \approx 0.52. \tag{11}$$

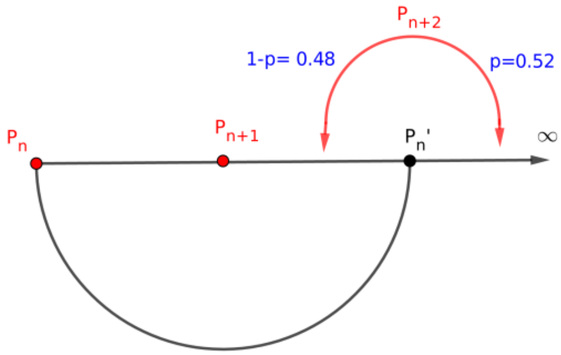

Figure 1: Occurrence of prime numbers

By the closure of the value obtained with 0.5 (uniformity), from this result one might say that the variation of the prime gap sequence is "random". So we may think at this stage that our random integer generator produces a good imitation of randomness. Note, however, that talking about randomness for a sequence of numbers is more for the unknowns or not-computed values of that sequence when going to infinity. Indeed, there is no randomness to expect from value that we already have. A picture of this result is given in Figure 1.

Notice that, using the χ^2-test for individual variation as we have just done will lead to the same conclusion. In the next paragraph, we give the result from the χ^2- test on a different view of the prime gap sequence.

5.2 χ^2-test of independence

To check if the rows in our generated csv files are uniformly distributed over $\{0, 1\}^d$, we choose $d = 4$ and $n = 10000$. This tests the independence to some extent, but it only tests if d consecutive calls of our generator are independent. Note, however, that the code producing the csv file has to change, because we prefer to count occurrences directly in python. The new code looks as follows.

```
#BTrandom2_testX2.py file
#By Bertrand Teguia
#Random code csv generator for a X^2 test of BTrandom2


from BTrandom2 import *

import csv

d = 2
n = 20000

lines = []
```

```
for i in range(n):
    row = []
    for j in range(d):
        row.append(BTrandom2())
    lines.append(’’.join(map(str, row)))

lines = map(lambda y: y.split(),list(set(map(lambda x:\
    x+" "+str(lines.count(x)), lines))))

with open(’BTrandom2_testX2.csv’, ’w’) as writeFile:
    writer = csv.writer(writeFile)
    writer.writerows(lines)

writeFile.close()
```

Running this update python code lead us to the following table 1, where $E_i = np_i$, with $p_i = \frac{1}{16}$ denote the expectation of O_i : occurrence number of the label i. o_i is the observed value. Remember that our null hypothesis is to have uniform distribution ($p_i = \frac{1}{16}$, $\forall i$).

Table 1: χ^2-test of independence for four consecutive variations of the prime gap sequence

i	O_i	E_i	$\frac{(O_i-E_i)^2}{E_i}$	
1111	104	625	434,3056	
0000	49	625	530,8416	
0011	501	625	24,6016	
0111	399	625	81,7216	
1010	1326	625	786,2416	
1101	818	625	59,5984	
1100	490	625	29,16	
1000	275	625	196	
0010	676	625	4,1616	
1001	859	625	87,6096	
1011	835	625	70,56	
1110	422	625	65,9344	
0110	944	625	162,8176	
0001	236	625	242,1136	
0101	1414	625	996,0336	
0100	652	625	1,1664	
	10000	10000	3772.8672	Total

As we have 15 degrees of freedom, from the χ^2-distribution table, one sees that we are far from uniformity, so we reject our null hypothesis. Thus despite the random behavior observed previously, the prime gap sequence appears to not behave randomly when collection of its consecutive variations is considered.

Conclusion

Terence Tao concluded "*Individual primes are believed to behave randomly, but the collective behavior of the primes is believed to be quite predictable*" [2]. Here we have find out an argument for sufficient progress towards improving that statement; and rather say, we can behave that individual consecutive primes appear randomly, but taken as small groups, the argument of randomness for primes is rejected. Thus our random imitation from prime numbers does not satisfy all the criteria used for acknowledgment of pseudo-randomness reproduction. This give us an understanding of the P&C Game [7], because as seen in the previous section, a consecutive constant behavior of the prime gap sequence variation does not happen often. That means the average displacement of a player during a game play is less in general.

For the study of the occurrence of primes, we think that a good recommendation for further understanding of their behavior is to consider them in small groups or better small groups of consecutive primes.

In a further study, since the hypothesis of uniformity is not accepted for consecutive groups of the prime gap sequence, interesting results for classification of prime numbers can be obtained considering the irregularity of these groups. Thus, we may probably deduce mathematical formulas hidden behind certain probabilistic arguments on primes.

Acknowledgments: I would like to thank AIMS-Cameroon for the facility that they give to do research.

References

[1] David F DiCarlo. Random number generation: Types and techniques. *Liberty University*, 2012.

[2] Terence Tao. Structure and randomness in the prime numbers. In *An Invitation to Mathematics*, pages 1–7. Springer, 2011.

[3] Robert J Lemke Oliver and Kannan Soundararajan. Unexpected biases in the distribution of consecutive primes. *Proceedings of the National Academy of Sciences*, 113(31):E4446–E4454, 2016.

[4] S Torquato, G Zhang, and M de Courcy-Ireland. Uncovering multiscale order in the prime numbers via scattering. *Journal of Statistical Mechanics: Theory and Experiment*, 2018(9):093401, sep 2018.

[5] OF TRUE RANDOMNESS. The importance of true randomness in cryptography. *Citeseer*.

[6] David Jones. Good practice in (pseudo) random number generation for bioinformatics applications. *URL http://www.cs.ucl.ac.uk/staff/d.jones/GoodPracticeRNG.pdf*, 2010.

[7] Bertrand Teguia. P&C Game. ResearchGate, 2019.

[8] Lamb Evelyn. Peculiar pattern found in 'random' prime numbers. `https://www.nature.com/news/peculiar-pattern-found-in-random-prime-numbers-1.19550`.

[9] Yitang Zhang. Bounded gaps between primes. *Annals of Mathematics*, pages 1121–1174, 2014.

[10] Eric W Weisstein. Twin primes. Wolfram Research, Inc., 2003.

[11] Tom Kennedy. *Monte Carlo Methods - a special topics course*. Spring, 2016.

[12] Python Software Foundation. Python 2.7.0 release. https://www.python.org/, 2019.

[13] John Renze and Eric W. Weisstein. Law of large numbers. *Mathworld*.