

Application of Floyd's Cycle Finding Algorithm to Check the Quality of Random Number Generator

Adyaksa Wisanggeni 13517091
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
adyaksa.wisanggeni@gmail.com

Abstract— Graph is one of the most commonly used mathematical concept in real life. Many problems in our daily life can be modeled as graph. One of the problems that computer scientist encounter is checking the quality of Pseudorandom Number Generator. One of the easiest ways to check the quality of a Pseudorandom Number Generator is by finding its repeated period. In this paper, we will try to find the period by using Floyd's Cycle Finding Algorithm, or more commonly known as Hare and Tortoise Algorithm.

Keywords—Pseudorandom Number Generator, Graph, Floyd's Cycle Finding Algorithm, Cycle.

I. INTRODUCTION

Random Number Generator (RNG) have many uses in computer science. Some of its application is in computer simulation, statistical sampling, cryptography, and other areas where producing an unpredictable result is desirable. In reality, scientist have trouble to create a generator with a true randomness. As substitute, they create Pseudorandom Number Generator. Pseudorandom Number Generator (PRNG) is a generator that's looks random but actually deterministic given the same initial state, or more commonly called as *seed*.

One of the most basic ways to check the quality of a Pseudorandom Number Generator is to find its period, or how many iterations it takes to generate a same number more than once. The easiest way to check this for a general generator is to model that generator as a graph, where each node represents a state in its generator. To find the generator's period is same as finding the cycle length in that graph. In this paper, we will try to find the cycle length of a Pseudorandom Number Generator using Floyd's Cycle Finding Algorithm.

II. BASIC THEORY

A. Graph

Graph is a collection of vertices, edge, and a relation that associates each edge with 2 vertices called its endpoints [1]. Two different vertices are called neighbor if and only if there is an edge that have both vertices as its endpoint. Two different vertices A and B is connected if and only if B is neighbor of A or neighbor of B is connected to A. Degree of vertex A is defined as the number of edges that have A as its endpoint. A cycle is a graph with N vertices and N edges where each pair of vertices is

connected, and every vertex have degree 2.

B. Directed and Undirected Graph

A directed graph is a graph where each edge has direction associated with them, whereas an undirected graph is a graph where each edge is bidirectional.

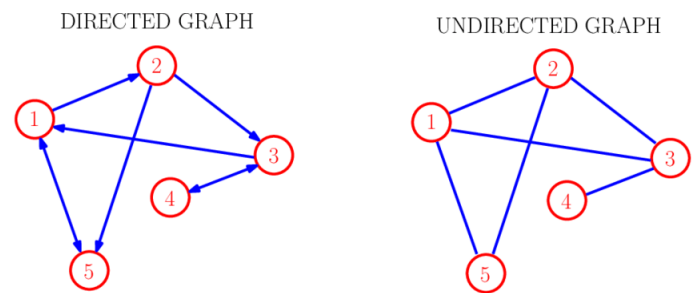


Figure 1. Directed and Undirected Graph [2]

C. Linear Congruential Generator

Linear Congruential Generator (LCG) is an algorithm that yields a sequence of pseudorandom number using a linear equation [3]. This generator is one of the oldest PRNG. Their advantage is very fast generation and its quality can be easily analyzed due to their simple structure. The following formulate is a formulate to generate a sequence of pseudorandom number using this generator

$$x_n = ax_{n-1} + b \pmod{m}$$

Linear Congruential Generator depends on 4 parameters:

- Initial state (x_0)
- Multiplier (a)
- Increment (b)
- Modulus (m)

For this paper, we will define function $LCG(x_0, a, b, m)$ as a Linear Congruential Generator with previously defined parameter.

D. Linear Congruential Generator as Graph

We can represent a Linear Congruential Generator as a directed graph with x_i as vertex and a directed edge connecting

x_i with x_{i+1} . The following is some graph model example for this generator.

- $LCG(0,3,2,5)$

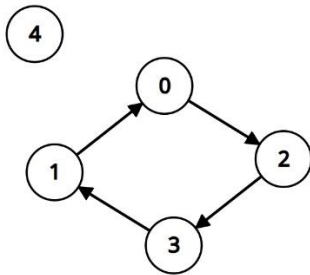


Figure 2. Graph representation of $LCG(0,3,2,5)$

(Generated from <https://nergi-r.github.io/GraphVisualizer/>)

- $LCG(0,1,1,5)$

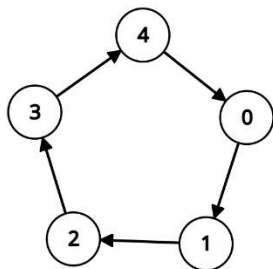


Figure 3. Graph representation of $LCG(0,1,1,5)$

(Generated from <https://nergi-r.github.io/GraphVisualizer/>)

- $LCG(1,2,0,8)$

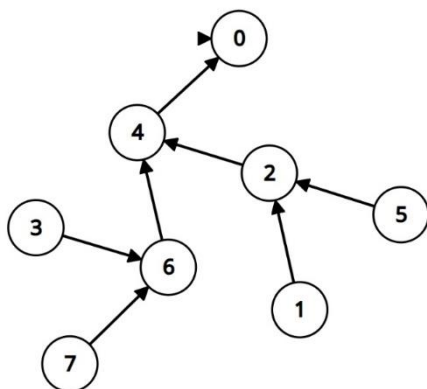


Figure 4. Graph representation of $LCG(1,2,0,8)$

(Generated from <https://nergi-r.github.io/GraphVisualizer/>)

From this example, we can see that function $LCG(x_0, a, b, m)$ have at most m states, and not necessarily have m period. Furthermore, an LCG with x_0 as its initial state doesn't

guarantee that they will have x_0 as its period base. What LCG guarantee is that its graph representation will always have a cycle, given finite modulus.

E. Depth First Search

Depth First Search (DFS) is one of graph traversal algorithm. In this algorithm, we find the "deepest" vertex connected to current vertex that not already visited, mark it visited, and search another one. This algorithm, as opposed to Breadth First Search, uses stack principle as its routine, where earliest visited vertex will finish last. Because we visit each vertex at most twice, and we use each edge at most twice too, the time complexity of this algorithm is $O(n + m)$ where n is the number of vertices, and m is the number of edges. Because we store whether a vertex is already visited or not, the space complexity of this algorithm is $O(n)$ where n is the number of vertices.

The following is pseudocode example for Depth First Search.

```

DFS(G)
1  for each vertex  $u \in G.V$ 
2     $u.color = WHITE$ 
3     $u.\pi = NIL$ 
4   $time = 0$ 
5  for each vertex  $u \in G.V$ 
6    if  $u.color == WHITE$ 
7      DFS-VISIT( $G, u$ )

DFS-VISIT( $G, u$ )
1   $time = time + 1$ 
2   $u.d = time$ 
3   $u.color = GRAY$ 
4  for each  $v \in G.Adj[u]$ 
5    if  $v.color == WHITE$ 
6       $v.\pi = u$ 
7      DFS-VISIT( $G, v$ )
8   $u.color = BLACK$ 
9   $time = time + 1$ 
10  $u.f = time$ 

```

Figure 5. Pseudocode of DFS [4]

III. CYCLE-FINDING ALGORITHM

A. Depth First Search

We can find cycle length in a generator using DFS. Let G be graph that represent generator, $time$ as how many vertices is already visited, $entry(u)$ as the time needed to start processing node u . Let v be the vertices that we visited twice. The period of a cycle with v as its base is given as $time - entry(v)$.

The following pseudocode is implementation of DFS to find generator's period

```

function cycleLength(Graph G) -> integer

time <- 0
G.period <- INF
for each u in G.node
    if (!G.visited[u])
        dfs(G,u)

-> G.period

procedure dfs(Graph G, node u)

if (G.visited[u])
    isCycle <- True
    G.period <- min(time-G.entry[u], G.period)
else
    G.entry[u] <- time
    time <- time + 1
    G.visited[u] <- True
    for each v in G.adj[u]
        dfs(G,v)

```

Figure 6. Pseudocode of Cycle-finding Algorithm using DFS

Because this cycle-finding algorithm uses DFS as its base, this algorithm has the same complexity as DFS, with time complexity of $O(n + m)$ and space complexity of $O(n)$. Because in generator m is always 1, the time complexity become $O(n)$

B. Floyd's Cycle-finding Algorithm

For $n \geq 1e9$, we need at least additional $1e9$ memory to find the generator's period. This is very costly. Because of that, Robert W Floyd created *Floyd's Cycle Finding Algorithm* or more commonly known as *Hare and Tortoise* algorithm with the time complexity $O(n)$ and space complexity $O(1)$. This algorithm uses double pointer as its routine, diminishing the space that DFS needed. Although DFS and this algorithm both have $O(n)$ complexity, this algorithm has a bigger constant, resulting a bit slower calculation in this algorithm.

The algorithm is as described below:

1. Let $f(x)$ be the next edge after x , $tortoise$ with initial value x_0 , $hare$ with initial value $f(x_0)$, $step_{hare}$ as the step that hare take, and $step_{tortoise}$ as the step that tortoise take.
2. While $hare$ is not equal to $tortoise$, $tortoise = f(tortoise)$, and $hare = f(f(hare))$.
3. If $hare = tortoise$, then we find a period with cycle at most $step_{hare} - step_{tortoise}$. Because $step_{hare} = 2 * step_{tortoise}$, we can simplify this as $step_{tortoise}$
4. Because step 1-3 doesn't guarantee that the period is the minimum one, we need to advance tortoise little by little while hare stay still until tortoise have the same step as hare. Or in other word, $step_{tortoise} = step_{tortoise} + 1$, $step_{hare} = step_{hare}$, $tortoise = f(tortoise)$ for every step until $step_{tortoise} = step_{hare}$. If while advancing the tortoise remain in the same state as hare,

we store the value of $step_{hare} - step_{tortoise}$.

5. The period is the minimum of $step_{hare} - step_{tortoise}$ that satisfy $hare = tortoise$ where $step_{hare}$ is not equal to $step_{tortoise}$

To explain further, the following is C implementation of *Hare and Tortoise* algorithm.

```

1 void hareAndTortoise(int seed){
2     unsigned long int tortoise = 0;
3     unsigned long int hare = findNext(tortoise);
4     Long Long diff = 1;
5     while(tortoise != hare){
6         tortoise = findNext(tortoise);
7         hare = findNext(findNext(hare));
8         diff++;
9     }
10    Long Long length = diff;
11    while(diff){
12        if(tortoise == hare){
13            length = diff;
14        }
15        diff--;
16        tortoise = findNext(tortoise);
17    }
18    printf("Cycle length is %lld\n", length);
19 }
20

```

Figure 7. C Implementation of Hare and Tortoise Algorithm

IV. PSEUDORANDOM NUMBER GENERATOR TESTING

A. glibc

The following is glibc implementation of `rand()` function that gcc use.

```

352 int
353 __random_r (struct random_data *buf, int32_t *result)
354 {
355     int32_t *state;
356
357     if (buf == NULL || result == NULL)
358         goto fail;
359
360     state = buf->state;
361
362     if (buf->rand_type == TYPE_0)
363     {
364         int32_t val = ((state[0] * 1103515245U) + 12345U) & 0x7fffffff;
365         state[0] = val;
366         *result = val;
367     }
368     else

```

Figure 8. glibc `rand()` implementation

(Taken from https://github.com/lattera/glibc/blob/master/stdlib/random_r.c, December 9, 2018 at 19.26 UTC+7)

From that code, we can see that `rand()` function uses $LCG(seed, 1103515245, 12345, 2^{31})$ as its routine. We will use our *Hare and Tortoise Algorithm* to find its period.

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  static unsigned long int next = 1;
5
6  unsigned long int findNext(unsigned long int cur){
7      return ((cur * 1103515245U) + 12345U)&0xffffffff;
8  }
9
10 void hareAndTortoise(int seed){ ...
29 }
30
31 void srand(unsigned int seed){
32     next = seed;
33 }
34
35 int main(){
36     hareAndTortoise(0);
37 }
38

```

Figure 9. Hare and Tortoise Algorithm to Find glibc rand() period

From that algorithm, our program finds 2147483648 as the period of glibc rand() function. Notice that this is 2^{31} , or full cycle of its modulus.

B. Delphi Pascal

The following is Delphi Pascal implementation of Random().

```

unit lcg_random;
// Delphi compatible LCG random number generator routines for FreePascal.
// (c)2017, Thaddy de Koning. Use as you Like
// Algorithm, Delphi multiplier and increment taken from:
// https://en.wikipedia.org/wiki/Linear_congruential_generator
// The default Delphi RandomSeed is determined as zero.
{$ifdef fpc}{$mode objfpc}{$endif}

interface

function LCGRandom: extended; overload; inline;
function LCGRandom(const range: longint): longint; overload; inline;

implementation

function IM: cardinal; inline;
begin
    RandSeed := RandSeed * 134775813 + 1;
    Result := RandSeed;
end;

function LCGRandom: extended; overload; inline;
begin
    Result := IM * 2.32830643653870e-10;
end;

function LCGRandom(const range: longint): longint; overload; inline;
begin
    Result := IM * range shr 32;
end;

end.

```

Figure 10. Delphi Pascal Random() Implementation

(Taken from

http://wiki.freepascal.org/Delphi_compatible_LCG_Random,
December 9, 2018 at 19.50 UTC+7)

From that code, we can see that Random() function uses LCG($seed, 134775813, 1, 2^{32}$). We will use our Hare and Tortoise Algorithm to find its period.

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  static unsigned long int next = 1;
5
6  unsigned long int findNext(unsigned long int cur){
7      return ((cur * 134775813U) + 1U)&0xffffffff;
8  }
9
10 void hareAndTortoise(int seed){ ...
29 }
30
31 void srand(unsigned int seed){
32     next = seed;
33 }
34
35 int main(){
36     hareAndTortoise(0);
37 }
38

```

Figure 11. Hare and Tortoise Algorithm to Find Delphi Pascal Random() period

From that algorithm, our program finds 4294967296 as the period of Delphi Pascal Random() function. Notice that this is 2^{32} , or full cycle of its modulus.

C. Xorshift

Marsaglia, in his paper, proposed a pseudorandom number generator that uses xor as its routine. Because of its simple operation by only using xor, this PRNG is faster than LCG that uses multiplication as its routine. Example implementation of xorshift for 32-bit number that he explained in his paper is as follows.

```

unsigned long xor() {
static unsigned long y=2463534242;
y^=(y<<13); y^=(y>>17); return (y^=(y<<5)); }

```

Figure 12. C implementation of Xorshift [5]

We can use our Algorithm to check its period.

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  unsigned long findNext(unsigned long cur){
5      cur^=(cur<<13);
6      cur^=(cur>>17);
7      cur^=(cur<<5);
8      return cur;
9  }
10
11 void hareAndTortoise(unsigned long seed){ ...
30 }
31
32 int main(){
33     hareAndTortoise(2463534242LL);
34 }
35

```

Figure 13. Hare and Tortoise Algorithm to Find Xorshift period

From that algorithm, our program finds 4294967295 as the period of xorshift 32 bit that Marsaglia proposed. Notice that this number is $2^{32} - 1$, because there are 2 graphs in xorshift representation: graph with only node 0 and its complement. This is because node 0 doesn't have anything that can be shifted, so a good xorshift cannot include 0 as it will create a generator with period 1.

D. C++11 minstd_rand0

In 1988, Park and Miller proposed a variant of Linear Congruential Generator with addition constant equals to 0 [6]. Later, C++11 implemented it under minstd_rand0 function. The following is its official implementation that used in its library.

```

1539  /**
1540   * The classic Minimum Standard rand0 of Lewis, Goodman, and Miller.
1541   */
1542  typedef linear_congruential_engine<uint_fast32_t, 16807UL, 0UL, 2147483647UL>
1543  minstd_rand0;
1544  ...

```

Figure 14 C++11 minstd_rand0 Implementation

(Taken from <https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/random.h> at December 9, 2018 at 22.33 UTC+7)

From that code, we can see that minstd_rand0 function that C++11 uses $LCG(seed, 16807, 0, 2^{31} - 1)$ as its routine. We will use our *Hare and Tortoise Algorithm* to find its period.

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  unsigned long findNext(unsigned long cur){
5      return (((Long Long)cur*16807UL)+0UL)%2147483647UL;
6  }
7
8  void hareAndTortoise(unsigned long seed){ ...
27 }
28
29 int main(){
30     hareAndTortoise(1);
31 }
32

```

Figure 15. Hare and Tortoise Algorithm to Find minstd_rand0 period

From that algorithm, our program finds 2147483646 as the period of C++11 minstd_rand0. Notice that this number is $2^{32} - 2$, because there are 2 graphs in xorshift representation: graph with only node 0 and its complement. This is because we can't use 0 as initial seed in multiplicative version of LCG, as 0 will result in itself if we try to generate its next number.

E. C++11 minstd_rand

Few years after they propose the use of $LCG(seed, 16807, 0, 2^{31} - 1)$, Park, Miller, and Stockmeyer advocated the new alternative of multiplicative version of LCG [7]. This version uses $LCG(seed, 48271, 0, 2^{31} - 1)$ as its routine, as can be seen in the official implementation of C++11

minstd_rand.

```

1545  /**
1546   * An alternative LCR (Lehmer Generator function).
1547   */
1548  typedef linear_congruential_engine<uint_fast32_t, 48271UL, 0UL, 2147483647UL>
1549  minstd_rand;
1550

```

Figure 16. C++11 minstd_rand Implementation

(Taken from <https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/bits/random.h> at December 9, 2018 at 22.51 UTC+7)

We will use our Hare and Tortoise to check for its period.

```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4  unsigned long findNext(unsigned long cur){
5      return (((Long Long)cur*48271UL)+0UL)%2147483647UL;
6  }
7
8  void hareAndTortoise(unsigned long seed){ ...
27 }
28
29 int main(){
30     hareAndTortoise(1);
31 }
32

```

Figure 17. Hare and Tortoise Algorithm to Find minstd_rand period

From that algorithm, our program finds 2147483646 as the period of C++11 minstd_rand.

F. Compiler PRNG Comparison

From previous experiment, we can create a table each random implementation method with its period.

Compiler	Type	Modulus/Range	Period
glibc	LCG 32 bit	2^{31}	2147483648
Delphi Pascal	LCG 32 bit	2^{32}	4294967296
-	Xorshift 32 bit	2^{32}	4294967295
C++11 minstd_rand0	LCG 32 bit	$2^{31} - 1$	2147483646
C++11 minstd_rand	LCG 32 bit	$2^{31} - 1$	2147483646

Figure 18. Comparison of each compiler RNG period

As we can see, every compiler that we test have a full or near full period of its LCG implementation. This is because the need of RNG implementation of each language is very high, that's why everyone that contribute to these languages always try to find the best RNG implementation that compatible with the language they use.

V. CONCLUSION

There are many things that must be tested to judge whether a generator is good enough to be used for critical application that require true randomness. Even so, checking the period of Pseudorandom Number Generator is one of the simplest ways to check the quality of the generator. Floyd's Cycle Finding Algorithm with its linear time complexity and constant memory is a good choice for finding its period. Furthermore, with its ease of implementation and easily customizable program, we can create a more variative test related to generator period, such as choose a random number as seed and check whether this number have cycle with more than certain threshold.

VII. ACKNOWLEDGMENT

I would like to express my deep gratitude to Mr. Rinaldi Munir, Mr. Judhi Santoso, and Mrs. Harlili as our lecturer in Discrete Mathematics Course for the knowledge that they shared upon us. I also would like to thank my family and friends for help and support while this paper is created. I also would like to thank all programming language creator that open-source their language implementation, as their contribution to make their source-code open helps me test my algorithm implementation.

REFERENCES

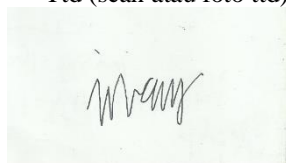
- [1] West, Douglas (2002), "Introduction to Graph Theory", Singapore: Pearson, pp. 2-3
- [2] Pommerening, K. (2016) *Bitstream Ciphers, week 3 notes*. Retrieved from https://www.staff.uni-mainz.de/pommeren/Cryptography/Bitstream/1_Classic/LCG.pdf
- [3] Briñón-Arranz, Lara. (2011). Cooperative control design for a fleet of AUVs under communication constraints.
- [4] Cormen, T. H., & Cormen, T. H. (2001). *Introduction to algorithms*. Cambridge, Mass: MIT Press, pp 604-605
- [5] George Marsaglia. 2003. Xorshift RNGs. *J. Stat. Softw.* 8, 14 (2003), 1--6.
- [6] S. K. Park, K. W. Miller, "Random number generators: good ones are hard to find", *Communications of the ACM*, v.31 n.10, p.1192-1201, Oct. 1988
- [7] Stephen K. Park; Keith W. Miller; Paul K. Stockmeyer (1988). "Technical Correspondence: Response" (PDF). *Communications of the ACM*. 36 (7): 108–110.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 9 Desember 2017

Ttd (scan atau foto ttd)



Adyaksa Wisanggeni 13517091