Przegląd wybranych generatorów liczb pseudolosowych
i analiza ich widma spektralnego.

MATEUSZ SOŁTYSIK, ANDRZEJ KWAK, WIKTOR DYNGOSZ
Politechnika Wrocławska, *Wydział Podstawowych Problemów Techniki*
Czerwiec, 2014

Wstęp

Pseudo-Random Number Generator (PRNG) – program lub podprogram, który na podstawie niewielkiej ilości informacji (ziarno, ang. seed) generuje deterministyczny, potencjalnie nieskończony, ciąg liczb. Generatory liczb pseudolosowych nie generują ciągów prawdziwie losowych - generator inicjowany ziarnem, które może przyjąć k różnych wartości, jest w stanie wyprodukować co najwyżej k różnych ciągów liczb. Ponieważ rozmiar zmiennych reprezentujących wewnętrzny stan generatora jest ograniczony i może on znajdować się tylko w ograniczonej liczbie stanów, po pewnym czasie generator dokona pełnego cyklu i zacznie generować te same wartości. Teoretyczny limit długości cyklu wyrażony jest przez 2^n, gdzie n to liczba bitów przeznaczonych na przechowywanie stanu wewnętrznego. W praktyce, większość generatorów ma znacznie krótsze okresy.
Pożądane cechy generatorów:

1. trudne do ustalenia ziarno, choć znany jest ciąg wygenerowanych bitów
2. trudne do ustalenia kolejno generowane bity, choć znany jest ciąg bitów dotychczas wygenerowanych

Przykłady zastosowań generatorów liczb pseudolosowych:

1. Kryptografia
2. Całkowanie numeryczne (metoda Monte-Carlo)
3. Symulacja systemów masowej obsługi
4. Automaty do gier losowych

W tej pracy przedstawiamy opis, zasadę działania oraz analizę widma spektralnego wymienionych algorytmów:

1. Blum Blum Shub
2. Liniowy Generator Kongruentny (LCG)
3. Mersenne twister
4. Generator Parka-Millera
5. Xorshift

todo

1 Opis i analiza algorytmów PRNG

Informacje wstępne

Każdy - opisany w tej pracy - algorytm, powinien być inicjowany tzw. seedem. Przypomnijmy, że dany algorytm z ustalonymi parametrami i znaną wartością seed, generuje te same ciągi! Aby zapobiec przekłamaniom w tym zakresie postanowiliśmy, że nasza implementacja będzie korzystała z funkcji $getSeed()$, która przy procesie generacji ziarna wykorzystuje niedeterministyczne źródło bitów ($/dev/urandom$).

```
public static long getSeed() {
      SecureRandom random = new SecureRandom();
      byte seed[] = random.generateSeed(20);
      return Longs.fromByteArray(seed);
}
```

Listing 1: Przykładowa funkcja generująca seed.

Testy spektralne

Analiza spektralna algorytmu służącego do generowania liczb pseudolosowych polega na wygenerowaniu liczby próbek p par uporządkowanych kolejnych liczb (x_i, x_{i+1}) dla $i \in 0, 1, 2, \ldots, p-1$. W ramach tej pracy została zaimplementowana lista algorytmów oraz stworzony program GUI, który umożliwia przeprowadzanie owych testów.
Testy dla każdego algorytmu nie powinny przekraczać wartości 10^9, ponieważ zakres 32-bitowej liczby jest równy $\frac{32\log(2)}{\log(10)} \approx 9$.

1.1 Blum Blum Shub

Algorytm *Blum Blum Shub* został zaproponowany przez Lenore Blum, Manuela Blum'a oraz Michaela Shub'a wiosną, 1986. roku, w pracy: "A Simple Unpredictable Pseudo-Random Number Generator"[1]. Generator ten jest postaci:

$$x_{n+1} = (x_n)^2 \bmod N$$

gdzie x_{n+1} to kolejny stan generatora, N to iloczyn dwóch dużych liczb pierwszych p i q takich, że:

1. dają w dzieleniu przez 4 resztę 3 ($p \equiv q \equiv 3 \bmod 4$),
2. mają możliwie mały $NWD(\phi(p-1), \phi(q-1))$, co zapewnia długi cykl.

[1] http://epubs.siam.org/doi/abs/10.1137/0215025

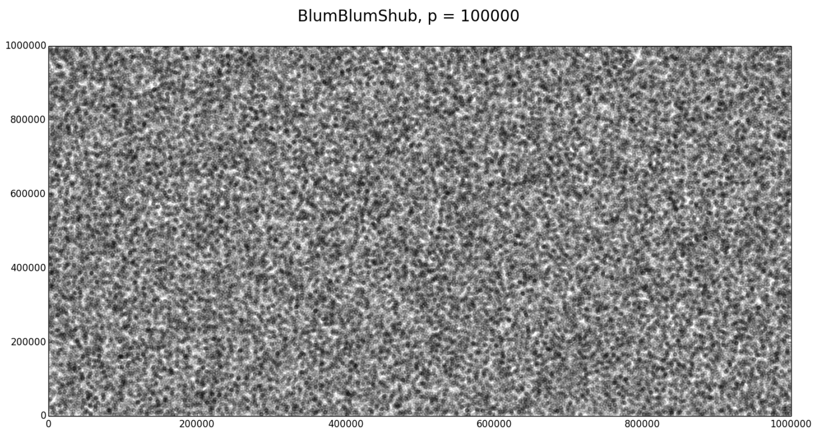
Wynikiem generatora jest kilka ostatnich bitów x_n.

```
private static final int p = 11;
private static final int q = 19;

public static long getRandomNumber() {
    seed = (seed * seed) % (p * q);
    return Math.abs(seed);
}
```

Listing 2: Generowanie następnej liczby pseudolosowej przez BBS

Generator ten jest nie jest najszybszy, jednakże jest bardzo bezpieczny. Przy odpowiednich założeniach, odróżnienie jego wyników od szumu jest równie trudne co faktoryzacja N.



1.2 Liniowy Generator Kongruentny (LCG)

Liniowe Generatory Kongruentne to najbardziej rozpowszechnione w praktyce algorytmy PRNG. Wyróżniamy dwa rodzaje tych generatorów:

1. Addytywne
   W postaci: $x_{i+1} = (a * x_i + c) \bmod m$
2. Multiplikatywne
   W postaci: $x_{i+1} = (a * x_i) \bmod m$, gdzie:

x_{i+1} to kolejna liczba pseudolosowa,
$m, 0 < m$ - zakres generowanej liczby
$a, 0 < a < m$ - współczynnik mnożenia
$c, 0 \leq c < m$ - współczynnik inkrementacji
Dla pewnych kombinacji parametrów generowany ciąg jest prawie losowy, dla innych bardzo szybko staje się okresowy. Cechą tych generatorów jest również fakt, iż jeśli kolejna wygenerowana liczba powtarza się w ciągu

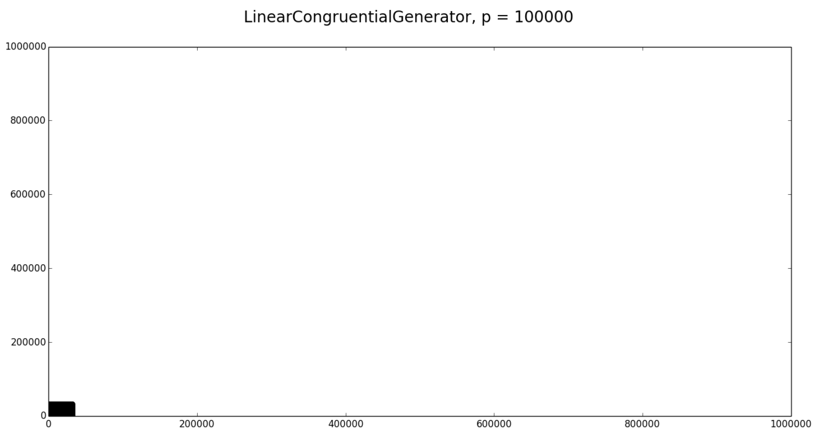
liczb już wygenerowanych, to wpadamy w cykl: $x_1, x_2, \ldots, x_1, x_2$. Mimo bardzo prostej budowy i zasady działania, powstało wiele prac i analiz matematycznych, które mówią nam o najlepszym doborze parametrów m, a, c. Najlepszym, to znaczy takim w którym cykl będzie jak najdłuższy. Przykładowe zalecenia:

1. Parametr m powinien być duży, najlepiej wielokrotnością 2 (najczęsciej spotyka się generatory LCG, w których $m = 2^{32}$ lub (w nowszych) $m = 2^{64}$)
2. Parametr a powinien być o jeden rząd mniejszy niż m.
3. Dobrą wartością dla a jest liczba postaci $t21$, gdzie t - liczba parzysta.

```
private final static long a = 25173;
private final static long b = 13849;
private final static long m = 32768;

public static long getRandomNumber() {
    seed = (a * seed + b) % m;
    return seed;
}
```

Listing 3: Generowanie następnej liczby pseudolosowej przez LCG

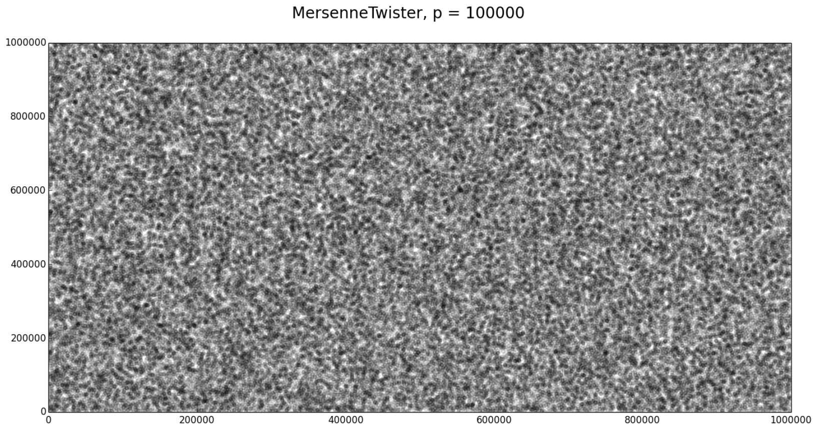


1.3 Mersenne twister

Mersenne twister został opracowany przez Makoto Matsumoto i Takuji Nishimura w 1997 roku[2]. Generator ten dostarcza wysokiej jakości liczby pseudolosowe oraz jest bardo szybki. Nazwa pochodzi od tego, że na długość okresu została wybrana pierwsza liczba Mersenne'a. Algorytm, mimo

[2]http://doi.acm.org/10.1145/272991.272995

swoich zalet, nie nadaje się do zastosowań kryptograficznych. Stosunkowo obserwacja niewielkiej liczby iteracji (624) pozwala przewidzieć wszystkie kolejne. Kolejną kwestią jest dlugi czas inicjalizacji algorytmu - w porównaniu do generatora Fibonacciego lub liniowego generatora kongruencyjnego.



1.4 Generator Park–Miller

Generator jest odmianą multiplikatywnego liniowego generatora kongruencyjnego, który określamy wzorem:

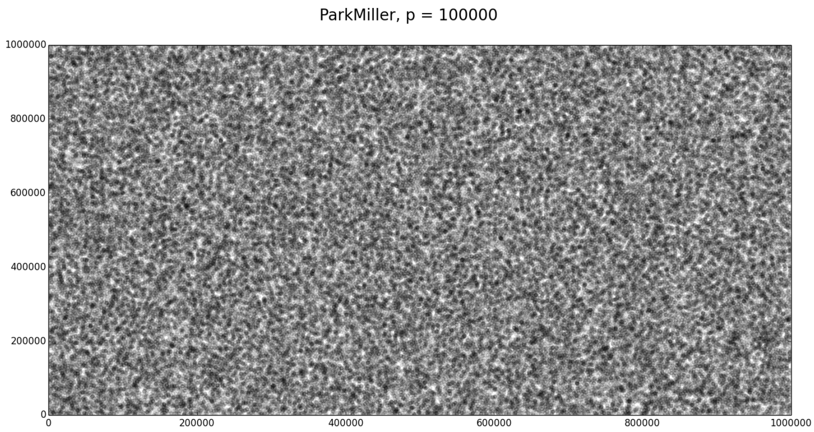
$$x_{n+1} = (a * x_n) \bmod M$$

gdzie, x_{n+1} to następna liczba pseudolosowa, a - współczynnik generujący kolejną liczbę, M - współczynnik określający zakres generowanych liczb (od 0 do $M - 1$).

```
private static final long max = ((long) 2 << 30) - 1;
private static final long a = 16807;

public static long getRandomNumber() {
    seed = (a * seed) % max;
    return seed;
}
```

Listing 4: Generowanie następnej liczby pseudolosowej przez algorytm Parka-Millera.



1.5 Xorshift

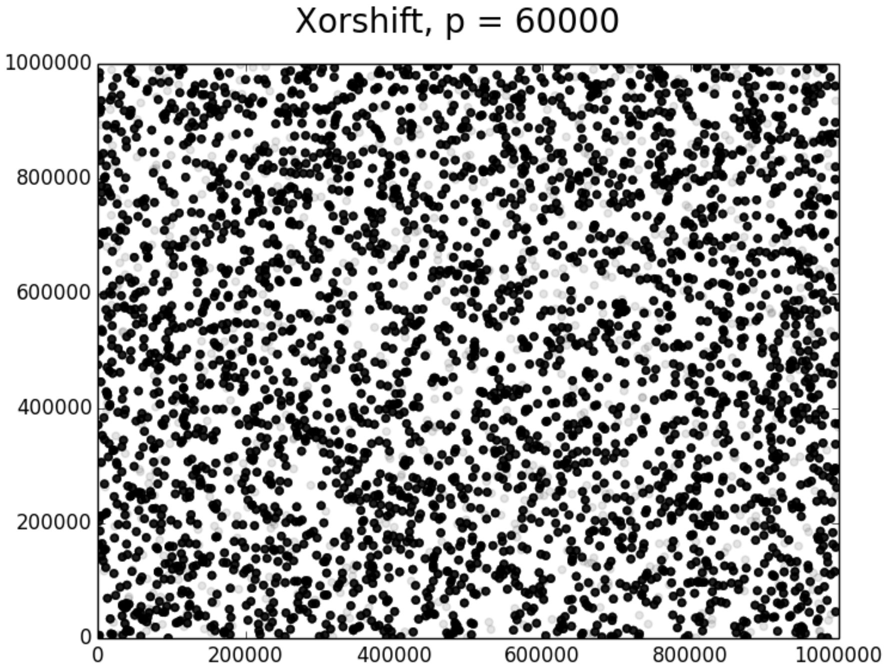
Algorytm został zaproponowany przez George Marsaglia[3] w 2003 roku. Liczba x_{n+1} jest generowana poprzez wielokrotną różnicę symetryczną liczb x_n i przesuniętej bitowo x_n. Wykorzystanie funkcji *XOR* sprawia, że ten algorytm jest niezwykle szybki na współczesnych komputerach.
Algorytm ten jest szybki ale nie niezawodny i z pewnością nie nadaje się do zastosowań kryptograficznych. Jednak, połączenie go z nieliniowym generatorem - jak pierwotnie sugerował autor - prowadzi do jednego z najszybszych generatorów, spełniających silne wymogi testów statystycznych.

```
public static long getRandomNumber() {
    seed ^= seed >> 12;
    seed ^= seed << 25;
    seed ^= seed >> 27;
    seed = (seed * 2685821657736338717L) % max;
    return Math.abs(seed);
}
```

Listing 5: Generowanie następnej liczby pseudolosowej przez Xorshift

Silnik przeglądarki internetowej Webkit korzysta z tego algorytmu przy wywoływaniu $Math.random()$ w języku JavaScript.

[3] http://www.jstatsoft.org/v08/i14/paper


Xorshift, p = 60000
1000000
800000
600000
400000
200000
0
0
200000
400000
600000
800000
1000000

Wnioski