

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.

Przykłady zastosowań generatorów liczb pseudolosowych:

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

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.

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

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. 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
6. Wichmann and Hill
7. Linear feedback shift register
8. RC4
9. Multiply-With-Carry

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`).

```
1 public static long getSeed() {  
2     SecureRandom random = new SecureRandom();  
3     byte seed[] = random.generateSeed(20);  
4     return Longs.fromByteArray(seed);  
5 }
```

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 $i \in 0, 1, 2, \dots, p - 1$. 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 ($p \equiv q \equiv 3 \pmod 4$),
2. mają możliwie mały $\text{NWD}(\phi(p-1), \phi(q-1))$, co zapewnia długi cykl.

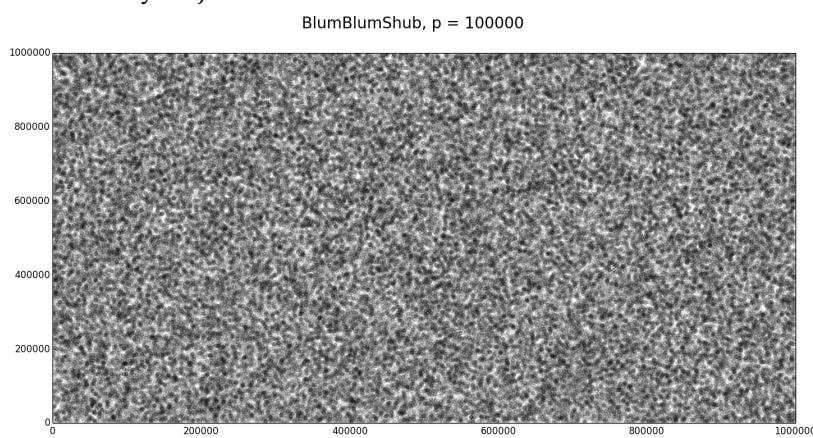
¹<http://pubs.siam.org/doi/abs/10.1137/0215025>

Wynikiem generatora jest kilka ostatnich bitów x_n .

```
1 private static final int p = 11;
2 private static final int q = 19;
3
4 public static long getRandomNumber() {
5     seed = (seed * seed) % (p * q);
6     return Math.abs(seed);
7 }
```

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. Mnożymytywne

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

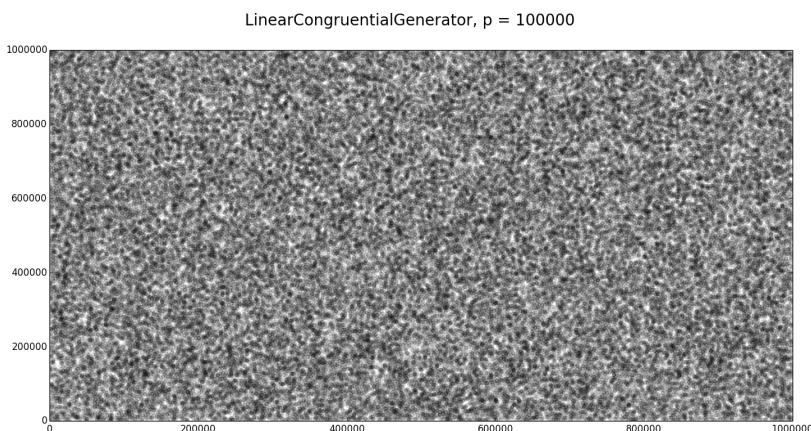
liczb już wygenerowanych, to wpadamy w cykl: $x_1, x_2, \dots, x_1, x_2$. Mimo bardzo prostej budowy i zasady działania, powstało wiele prac i analiz matematycznych, które mówią nam o najlepszym doborze parametrów m, a, c . Najlepszym, to znaczy takim w którym cykl będzie jak najdłuższy. Przykładowe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

1. Parametr m powinien być duży, najlepiej wielokrotnością 2 (najczęściej 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.

```

1 private final static long a = 6364136223846793005L;
2 private final static long b = 1442695040888963407L;
3 private final static long m = (long) Math.pow(2, 64);
4
5 public static long getRandomNumber() {
6     seed = (a * seed + b) % m;
7     return seed;
8 }
```

Listing 3: Generowanie następnej liczby pseudolosowej przez LCG

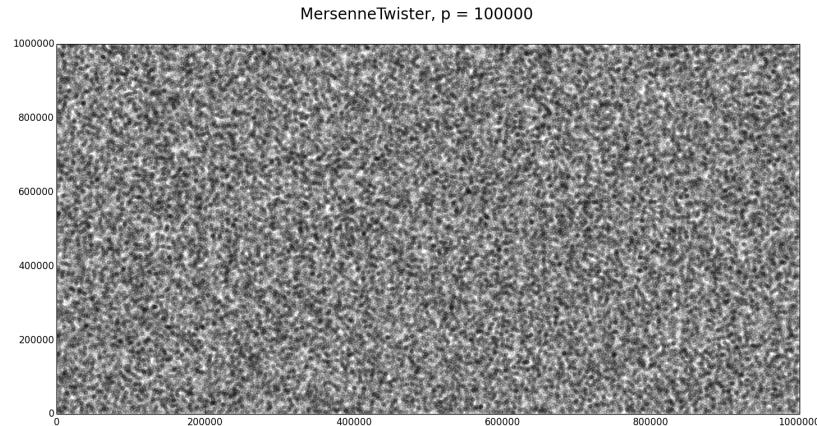


1.3 Mersenne twister

Mersenne twister został opracowany przez Makoto Matsumoto i Takuji Nishimura w 1997 roku². Generator ten dostarcza wysokiej jakości liczby pseudolosowe oraz jest bardzo 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 długi 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:

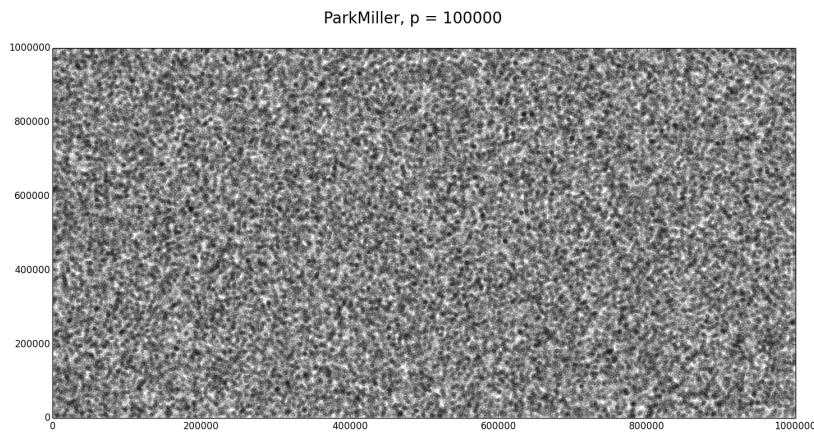
$$x_{n+1} = (a * x_n) \bmod M$$

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

```

1 private static final long max = ((long) 2 << 30) - 1;
2 private static final long a = 16807;
3
4 public static long getRandomNumber() {
5     seed = (a * seed) % max;
6     return seed;
7 }
```

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



1.5 Xorshift

Algorytm został zaproponowany przez George Marsaglia³ 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 najszyszych generatorów, spełniających silne wymogi testów statystycznych.

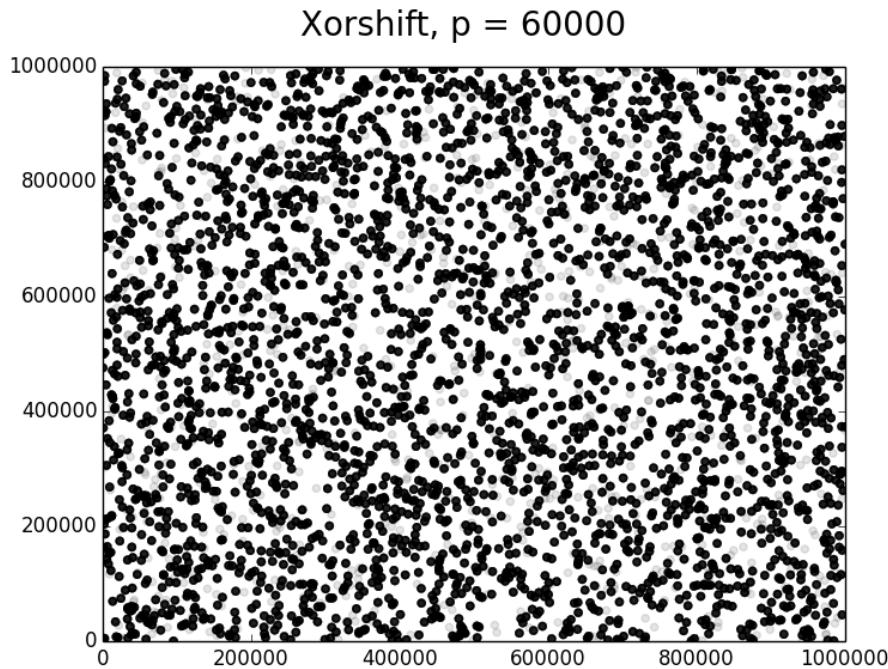
```

1 public static long getRandomNumber() {
2     seed ^= seed >> 12;
3     seed ^= seed << 25;
4     seed ^= seed >> 27;
5     seed = (seed * 2685821657736338717L) % max;
6     return Math.abs(seed);
7 }
```

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>



1.6 Wichmann and Hill

Algorytm został zaproponowany przez autorów w 1982 roku. Jego przybliżony okres wynosi 2^{43} z tego powodu generator stał się bardzo popularny. Algorytm wykorzystuje kombinacje trzech liniowych generatorów kongruentnych (LCG). Na początku inicjalizowane są trzy wartości I_x, I_y, I_z które przyjmują wartości od 1 do 30,000. Następnie wykonywane są następujące operacje:

$$I_x := 171 \times (I_x \bmod 177) - 2 \times (I_x : 177)$$

$$I_y := 172 \times (I_y \bmod 176) - 35 \times (I_y : 176)$$

$$I_z := 170 \times (I_z \bmod 178) - 63 \times (I_z : 178)$$

if $I_x < 0$ **then**

$$I_x := I_x + 30269;$$

if $I_y < 0$ **then**

$$I_y := I_y + 30307;$$

if $I_z < 0$ **then**

$$I_z := I_z + 30323;$$

$$W := I_x/30269.0 + I_y/30307.0 + I_z/30323.0$$

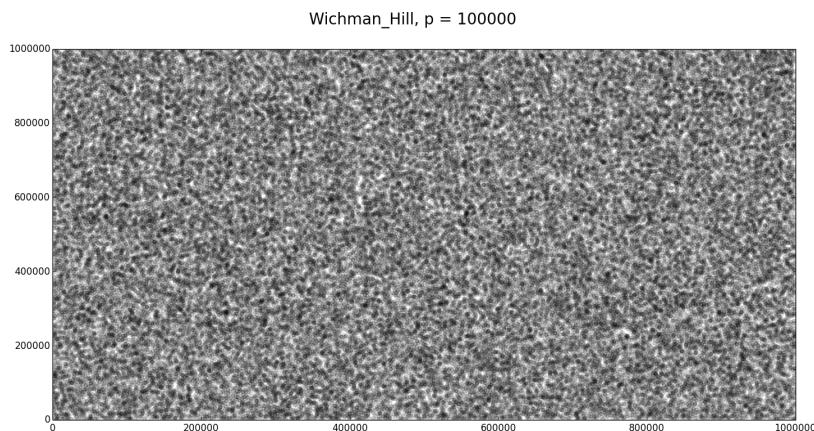
W pracy przy implementacji algorytmu skorzystano z faktu, że WH RNG może być zapisany jako prosty generator kongruentny za pomocą wzoru:

$$X_{n+1} = 16555425264690 \times X_n \bmod 27817185604309$$

znanego także jako rekursja Zeisela.

```
1  public long getRandomNumber() {
2      seed = (16555425264690L * seed) % 27817185604309L;
3      return Math.abs(seed);
4 }
```

Listing 6: Generowanie następnej liczby pseudolosowej przez algorytm Wichmanna-Hilla na podstawie rekursji Zeisela.



1.7 Linear feedback shift register

Linear feedback shift register albo inaczej Rejestr przesuwający z liniowym sprzężeniem zwrotnym to generator, którego bit wejściowy jest funkcją jego poprzedniego stanu. Wartość początkowa rejestrów jest jego ziarnem. Następnie ustalone funkcje konkretnych komórek rejestrów XORują wynik bitu wychodzącego.

```
1 // wygeneruj liczbe losowa
2 int next = 0;
3 for (int i = 0; i < M; i++) {
4     next |= (bits[i] ? 1 : 0) << i;
5 }
6
7 // przesun na zero jesli liczba ujemna
8 if (next < 0) next++;
9
10 // wylicz ostatni rejestr ze stanow okreslonych rejestrów
11 bits[M] = false;
12 for (int TAP : TAPS) {
13     bits[M] ^= bits[M - TAP];
14 }
15
16 // przesuwanie rejestrów
17 System.arraycopy(bits, 1, bits, 0, M);
```

```

18
19     return Math.abs(next);

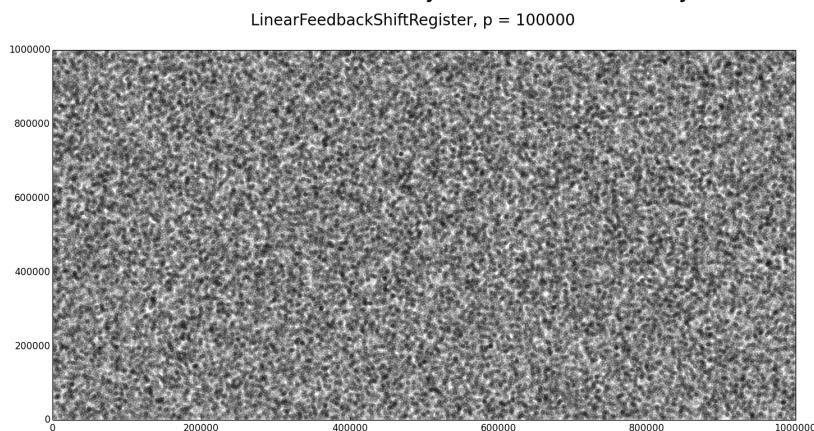
```

Listing 7: generowanie liczby za pomocą LFSR i przesuwanie komórek.

Jak widzimy na powyższym listingu liczba generowana jest za pomocą obecnych stanów tablicy. Po wylosowaniu wyrzucany jest ostatni bit i jest xorowany z określonymi przez nas komórkami. Następnie tablica jest przesuwana o 1.

Każdy LFSR jest związany z określonym wielomianem z pierścienia wielomianów $R[t]$, gdzie R jest ciałem skończonym Z_2 . Okres rejestru jest ograniczony przez stopień stwarzyszonego z nim wielomianu i wynosi maksymalnie $2^d - 1$, gdzie d jest stopniem wielomianu (ciało Z_2 ma charakterystykę równą 2). Okres danego LFSR jest maksymalny jeżeli stwarzyszony z nim wielomian jest wielomianem pierwotnym. Rejestr taki, nazywamy rejestrem maksymalnej długości.

Zaletą LFSR jest niesamowita szybkość wynikająca z operacji logicznych przez co znalazłem zastosowanie w szyfrach strumieniowych.



1.8 RC4

RC4 to szyfr strumieniowy, wymyślony przez Rona Rivesta w 1987 roku. Algorytm opiera się na 256 elementowej tablicy i dwóch wskaźnikach. Początkowo tworzona jest permutacja identycznościowa. Następnie w 256 iteracjach tablica jest przetwarzana i mieszana z bajtami klucza.

```

1     String key = Long.toHexString(seed);
2     int j = 0;
3     for(int i = 0; i < 256; i++){
4         j = (j + list.get(i) + (int) key.charAt(i%key.length()
5             )) % 256;
6         int tmp = list.get(i);

```

```

6         list.set(i, list.get(j));
7         list.set(j, tmp);

```

Listing 8: Inicjalizowanie RC4.

Następnie do generowania kolejnych bitów liczby, ponownie tablica jest przetwarzana.

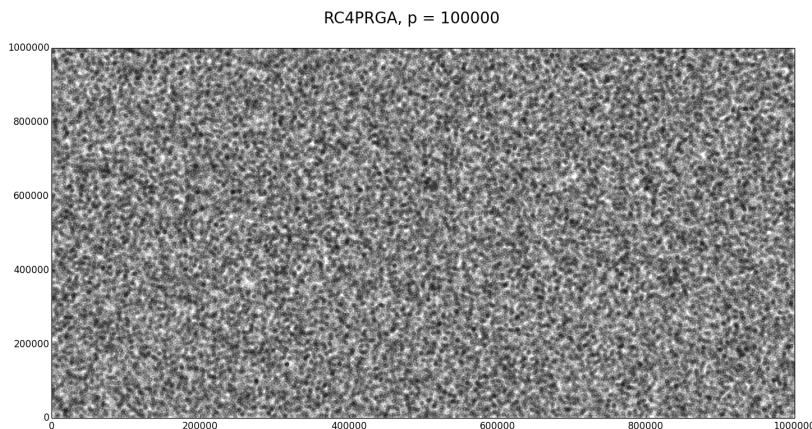
```

1  private String prga() {
2      int i, j;
3      i = j = 0;
4      for(int k = 0; k < 256; k++) {
5          i = (i+1) % 256;
6          j = j + list.get(i) % 256;
7
8          int tmp = list.get(i);
9          list.set(i, list.get(j));
10         list.set(j, tmp);
11     }
12     int t = list.get((i+j)%256);
13     return Integer.toHexString(list.get(t));
14 }

```

Listing 9: Generowanie losowego bajtu liczby za pomocą RC4.

Dużą zaletą tego szyfru jest jego szybkość. Generator znacznie wydajniej generuje bity od liniowych generatorów.



1.9 Multiply-With-Carry

Algorytm wymyślony przez George'a Marsaglia. Jego głównymi zaletami jest szybkość i wysoki okres (od ok. 2^{60} do $2^{2000000}$). MWC opiera się na operacji modulo liczby b (zazwyczaj wynosi ona 2^{32} dlatego, że w większości komputerów standardowa operacja) Generator wymaga podstawy

b mnożnika a , zbioru x losowych wartości takich że r zawiera się w b i początkowego przeniesienia $c < a$. Algorytm wygląda następująco:

$$(1) \quad t := a \times x + c$$

$$(2) \quad x := t \bmod b$$

$$(3) \quad c := (\text{int}) \frac{t}{b}$$

Warto zauważyć, że przy arytmetyce $b = 2^{32}$ wartość c będzie generowana przez pierwsze 32 bity liczby, natomiast kolejna wartość x przez dolne 32 bity. Mnożnik a jest dobrany w taki sposób aby $ab - 1$ i $(ab - 1)/2$ były liczbami pierwszymi.

Generator MWC przechodzi testy statystyczne, których generatory LCG nie przechodzą. Przez ten fakt MWC rozważany był jako algorytm kryptograficznie bezpieczny. Jednak badania pokazują, że bity generowane są lekko stronniczo, co przekreśla go w zastosowaniach kryptograficznych.

