Metody oceny jakości generatorów liczb losowych

Maciej Pawlik

Wydział fizyki i informatyki stosowanej, AGH Kraków
email: yaq@fatcat.ftj.agh.edu.pl
www: http://fatcat.ftj.agh.edu.pl/~yaq/

Streszczenie

W dokumencie przedstawiono wybrane metody oceny jakości generatorów liczb losowych i przykłady ich zastosowania. Dotychczas wykorzystanie liczb losowych w środowisku informatycznym wiązało się prawie wyłącznie ze specyficznymi metodami obliczeń numerycznych. Obecnie w epoce internetu i powszechnej wymiany informacji, często nieświadomie korzystamy z algorytmów opierających swoje działanie na właściwościach liczb losowych. Niezależnie od sposobu wykorzystania liczby losowe są bezużyteczne jeżeli nie są dobrej jakości, dla komputera - maszyny deterministycznej jest to zagadnienie szczególnie skomplikowane. Obecnie nie ma rozwiązań idealnych, są tylko lepsze lub gorsze , aby wybrać te, które dadzą nam największą szansę na dobre wyniki opracowano całe zestawy testów, z których wybrane zostaną omówione.

1 Wstęp

1.1 Liczby losowe

Liczby losowe są rezultatem działania określonego mechanizmu losującego. Istota losowości polega na tym, że dla liczb z przedziału $1...n$, szansa że otrzymamy w wyniku losowania konkretną liczbę wynosi $\frac{1}{n}$. Wynika z tego, że rezultat pojedynczego losowania jest niezależny od pozostałych, dlatego ciężko rozpatrywać kwestię losowości pojedynczej liczby. W przypadku idealnego losowania mówimy o generowaniu prawdziwych liczb losowych. Naturalne generatory liczb losowych to na przykład rzut monetą lub kostką do gry i tasowanie talii kart. Uzyskanie prawdziwych liczb losowych jest trudne, nie można stwierdzić z cała pewnością, że wyeliminowało się z układu generującego wszystkie czynniki które mogłyby zakłócić proces losowania. Często zdarza się, że potrzebujemy dużej ilości liczb losowych, a metody pozyskiwania liczb z czynników nie deterministycznych są dość wolne.

1.2 Liczby pseudo losowe

Liczby pseudo losowe to liczby wykazujące pewne właściwości liczb prawdziwie losowych, są efektem działania algorytmu, opracowanego w celu zbliżenia się do symulowania idealnego generatora. Ich wadą jest najczęściej niska jakość,

wymagania powszechnych standardów wobec liczb pseudolosowych nie są wygórowane. Generator pseudolosowy powinien zostać dokładnie przetestowany w docelowym środowisku przed konkretnym zastosowaniem wyników jego działania. Zalety, które decydują o tym, że w praktyce najczęściej stosujemy liczby pseudo losowe zamiast prawdziwie losowych to szybkość generowania, operując algorytmem ogranicza nas tylko moc obliczeniowa komputera i powtarzalność, kolejne liczby generowane są na podstawie poprzednich, możemy dowolna ilość razy powtarzać ciąg bez zapamiętywania jego wyników.

1.3 Generatory liczb losowych

Jak już zostało wspomniane wcześniej, do generowania liczb prawdziwie losowych najkorzystniej używać generatorów naturalnych, czyli obserwacje zjawisk, których wyników nie możemy precyzyjnie przewidzieć. Dobre generatory to np. rzut monetą, rejestracja „białego" szumu, czas pomiędzy uderzeniami w klawiaturę komputera. W zależności od zapotrzebowania liczby mogą być generowane na bieżąco, ale często obliczenia wymagają milionów liczb co przy standardowej szybkości generowania przez np. urządzenie `/dev/random` w linuksie może okazać się nie wystarczające.

1.4 Generatory liczb pseudo losowych

Generacją liczb pseudolosowych zajmują się odpowiednie algorytmy, obliczają kolejną liczbę na podstawie poprzedniej. Najczęściej zaleca się podanie pierwszej liczby (`srand()` w bibliotece standardowej C) tak aby przy każdym uruchomieniu generatora otrzymać inny ciąg. W praktyce rolę pierwszej liczby zastępuje aktualny czas w postaci tak zwanego „timestampu", w razie potrzeby można ustawić początkową liczbę na stałe, tak aby zawsze otrzymywać ten sam ciąg liczb, jest to pożądane w przypadku testowania algorytmów, gdzie zależy nam na takich samych warunkach przy każdym teście. Pewnym udoskonaleniem, jest okresowe wprowadzenia do generatora liczby prawdziwie losowej jako tej od której rozpocznie się generowanie kolejnych pseudo losowych, ale nie zawsze przynosi to oczekiwane efekty, ponieważ na czas okresu dalej jesteśmy przywiązani do z góry określonej sekwencji.

2 Testowanie generatorów

2.1 Cele testowania

Głównym celem statystycznych testów generatorów jest odrzucenie dających wyniki nie losowe, przywiązuję się o wiele większą uwagę do tego aby nie zaakceptować generatora złego niż przez pomyłkę odrzucić dobry. Obecnie jest wiele metod generowania liczb a w przypadku złego wyboru skutki mogą być bardzo dotkliwe dla użytkownika. Podstawowe cechy dobrego generatora według NIST[1]:

1. Jednorodność. W każdym punkcie generowanego ciągu bitów prawdopodobnieństwo wystąpienia jedynki lub zera jest takie samo i wynosi $\frac{1}{2}$, oczekiwana liczba zer w całym ciągu wynosi około $\frac{n}{2}$ dla ciągu n bitów.
2. Skalowalność. Każdy podciąg ciągu bitów, który uzyskał pozytywny wynik testu jakości, poddany temu samemu testowi powinien również uzyskać wynik pozytywny.
3. Zgodność. Zachowanie generatora musi dawać podobne rezultaty niezależnie od początkowej wartości lub fizycznego zjawiska będącego źródłem „losowości".

Każdy z testów sprawdza zachowanie w pewnym środowisku (testowane jest wystąpienie konkretnej negatywnej cechy), dlatego aby w ogóle można było mówić o jakości generatora należy przetestować go przy użyciu pakietu lub specjalnego testu przygotowanego „pod" konkretne zastosowanie. Przykładowe zastosowania liczb losowych to:

- Obliczenia numeryczne. Do różnych metod obliczeniowych potrzebujemy dużej ilości liczb losowych o określonym rozkładzie, na przykład całkowanie metodą Monte-Carlo
- Kryptografia. Obecnie szyfrowanie wiadomości opiera się na ograniczonej mocy obliczeniowej komputerów i losowaniu liczb które bardzo trudno uzyskać starając się wyliczyć je z gotowego wyniku. W przypadku generowania klucza prywatnego, użytkownik najczęściej proszony jest o wykonanie ruchów myszką bądź napisanie krótkiego tekstu na klawiaturze, w celu zebrania losowych danych, daje to pewność że nikt nie będzie mógł dokonać dekompozycji zaszyfrowanej wiadomości znając zależności pomiędzy kolejnymi liczbami losowymi.
- Inne zastosowania, takie jak tworzenie złudzenia niepowtarzalności w grach, czyli losowych wydarzeń, generowanie identyfikatora sesji do uproszczonej kontroli dostępu do treści w przypadku aplikacji webowych. Tu możemy wymagać generatorów szybkich ,niekoniecznie wysokiej jakości, gracz co najwyżej zauważy drobne regularności, a sesja użytkownika trwa przeważnie tak krótko, że jest zbyt mało czasu na znalezienie odpowiedniej wartości identyfikatora.

2.2 Metody analizy wyników

Przedstawione testy pochodzą z pakietu DieHarder[2], który jest odpowiednikiem starszego i uznanego za jeden z najlepszych pakietu DieHard. Nowsza wersja wprowadza szereg udoskonaleń, dodatkowych testów (miedzy innymi z pakietu NIST[1] i jest dystrybuowana na licencji GPL, co pozwala przeanalizować kod, wprowadzać potrzebne zmiany w przypadku specyficznych środowisk i rozpowszechniać go dalej. Sam autor w dokumentacji stwierdza, że nie można ufać testom jeżeli nie można przeanalizować ich kodu źródłowego, ma to swoje odbicie w zachowaniu innych wydawców, praktycznie każdy publikuje kod źródłowy.

Pakiet DieHarder prezentuje wyniki większości testów w podobny i czytelny sposób. Poza testami, ze ściśle określonymi warunkami sukcesu stosuje się

analizę statystyczną. Badamy hipotezę zerową mówiącą, że „dany ciąg jest losowy" w zależności od testu otrzymujemy p-wartość obliczoną na podstawie odchylenia rozkładu od wartości oczekiwanej, która mówi nam z jakim prawdopodobieństwem mamy do czynienia z dobrym generatorem i wartość powyżej której uznajemy, że nie ma podstaw by odrzucić hipotezę, najczęściej jest to liczba rzędu 10^{-6}. Problem pojawia się przy bardziej skomplikowanych generatorach i dużych porcjach danych, gdzie pojedyncza p-wartość może nam nie wystarczyć aby ocenić generator (np. parokrotnie otrzymalibyśmy wynik w okolicach jednej liczby, to by oznaczało pewne uwarunkowania dotyczące liczb, powodujące generowanie ciągle tego samego wyniku) w tym celu każdy test jest przeprowadzany n razy (najczęściej $n = 100$). Otrzymane p-wartości poddajemy testowi *Kolmogorova-Smirnova* aby sprawdzić czy p-wartości przedstawiają rozkład równomierny, co oznaczało by ich losowy charakter, czyli otrzymujemy kolejna „finalną" p-wartość i sprawdzamy czy mieści się w zadanym przedziale. Podsumowując, nie testujemy tylko generatorów, ale testujemy też charakter testów, co pozwala nam przy względnie małej ilości przebiegów ocenić czy testy mówią prawdę. W celu lepszego odbioru danych prezentowany jest histogram p-wartości - dokonując modyfikacji w parametrach możemy obserwować wyniki w poszukiwaniu jakiś charakterystycznych cech, np. zmiany skupienia wartości w zależności od liczebności próbek - jeżeli zostanie to zaobserwowane możemy odrzucić hipotezę, że mamy do czynienia z dobrym generatorem.

2.2.1 Test Kolmogorova-Smirnova (λ Kołmogorova)

Test polega na znalezieniu maksymalnej wartości odchylenia wzorcowego rozkładu od wyników uzyskanych z doświadczenia:

$$D = max|K(a_i) - F(a_i)|$$

Dla każdego i dla którego istnieje wynik próby. Wartość λ obliczana jest jako iloczyn D i pierwiastka liczebności, następnie jest porównana z λ krytyczną i wynik pozwala nam stwierdzić zgodność rozkładów. Test *Kolmogorova-Smirnova* w DieHarder jest stosowany w „odmianie" *Kuipera*, gdzie badamy zgodność dystrybuanty otrzymanego rozkładu z rozkładem normalnym, na podstawie uśrednionych odchyleń, a nie jego maksymalnej wartości.

2.2.2 Test χ^2

W przypadku gdy sam test wymaga badania zgodności rozkładów najczęściej stosuje się test χ^2, dany:

$$\chi^2 = \sum_{i=1}^{n} \left(\frac{O_i - E_i}{\sigma_i} \right)^2$$

gdzie O_i to wartość doświadczalna, E_i to wartość wzorcowego rozkładu, σ_i to odchylenie standardowe, n to ilość pomiarów. Wykorzystując otrzymaną wartość odczytujemy ze stablicowanych wartości prawdopodobieństwo, że oba rozkłady są zgodne.

3 Testy generatorów

W tym punkcie zostaną przedstawione dostępne testy generatorów liczb losowych.

3.1 Diehard

Testy z orginalnego pakietu Diehard, w większości zmodyfikowane przez autora DieHarder.

3.1.1 Birthdays test

Test urodzin, polega na „paradoksie urodzin" czyli szansie, że w grupie osób znajdą się osoby które mają urodziny tego samego dnia w roku. Dla potrzeb testu grupa liczy 512 osób a dzień roku określa 24 bitowa liczba, czyli bierzemy pod uwagę 2^{24} dni. Zliczane są odległości pomiędzy datami urodzin, dla dobrego generatora rozkład powinien być podobny do rozkładu Poissona.

3.1.2 Overlapping 5-Permutations test

Test działa na milionie 32 bitowych liczb. Ciąg dzielimy na pięcioelementowe podciągi pokrywające się, każdy z ciągów może być w jednym ze 120 stanów wybranych pod względem kolejności liczb w ciągu (5! możliwości), wystąpienia poszczególnych stanów są zliczane. Potem wyniki są statystycznie porównywane do danych wzorcowych otrzymanych z rozkładu normalnego.

3.1.3 32x32 Binary Rank test

Test operuje na macierzach o wymiarach $31x31$. Z 31 wylosowanych liczb 32 bitowych wybiera się pierwsze bajty i zapełnia poszczególnymi bitami macierz. Następnie liczy się rząd macierzy i sumuje ich wystąpienia, wartości poniżej 28 są bardzo rzadkie i zliczane są razem z 28, następnie wykonuje się test χ^2 w odniesieniu do wartości generatora idealnego.

3.1.4 6x8 Binary Rank test

Test analogiczny do poprzedniego, główna różnica to rozmiar macierzy, odpowiednio dobiera się mniejsze porcje danych.

3.1.5 Bitstream test

Test analizuje ciąg liczb losowych jako ciąg znaków o alfabecie 0 i 1, ciąg dzieli się na dwudziestoliterowe pokrywające się słowa. Test liczy ilość słów które nie wystąpiły w ciągu składającym się z 2^{21} pokrywających się słów, przy istniejących 2^{20} możliwych kombinacjach tworzących słowa. Dla generatora prawdziwych liczb losowych suma takich słów powinna podlegać rozkładowi normalnemu o wartości oczekiwanej 141909 i odchyleniu standardowym 428. Otrzymana

wariancja jest szukaną p-wartością:

$$p = \frac{\text{wynik} - 141909}{428}$$

3.1.6 Overlapping Pairs Sparse Occupance test

Test rozpatruje dwu literowe słowa z alfabetu o 1024 literach, każda litera jest wyznaczana z dziesięciu bitów wybranych z 32 bitowej liczby losowej z testowanego ciągu. Generowane jest 2^{21} pokrywających się słów i obliczana jest liczba brakujących możliwych do utworzenia słów. Wyniki powinny podlegać rozkładowi normalnemu o parametrach $\mu = 141909$ i $\sigma = 290$ więc wartość p jest równa:

$$p = \frac{\text{wynik} - 141909}{290}$$

3.1.7 Overlapping Quadruples Sparce Occupancy test

Podobnie jak poprzednio, używamy słów i alfabetu, ale zmienia się liczba liter w słowie, na 4 i liczba liter na 32. Każda litera odpowiada wybranym pięciu bitom z testowanego ciągu, którego elementami są 32-bitowe liczby. P-wartość wyraża się wzorem:

$$p = \frac{\text{wynik} - 141909}{295}$$

3.1.8 DNA test

Test operuje na alfabecie czteroliterowym: C, G, A, T, (stąd analogia do DNA) litera jest ustalana na podstawie dwóch bitów z testowanego ciągu, każde słowo ma 10 liter, podobnie jak poprzednio w ciągu o długości 2^{21} pokrywających się słów zliczamy te, które nie wystąpiły, parametry rozkładu normalnego powinny wynosić: $\mu = 141909$ i $\sigma = 339$.

$$p = \frac{\text{wynik} - 141909}{339}$$

3.1.9 Count the 1s (stream) test

Kolejny test operujący na słowach i alfabecie, ale inaczej tworzonych. Traktuje ciąg liczb jako strumień bajtów z których każdemu przyporządkowuje literę w zależności od tego ile dany bajt zawiera jedynek.

ilość jedynek	prawdopodobieństwo wystąpienia	litera
0	$\frac{1}{256}$	A
1	$\frac{8}{256}$	A
2	$\frac{28}{256}$	A
3	$\frac{56}{256}$	B
4	$\frac{70}{256}$	C
5	$\frac{56}{256}$	D
6	$\frac{28}{256}$	E
7	$\frac{8}{256}$	E
8	$\frac{1}{256}$	E

Otrzymujemy ciąg pokrywających się słów składających się z pięciu liter od A do E o prawdopodobieństwie wystąpienia odpowiednio $\frac{37}{256}$, $\frac{56}{256}$, $\frac{70}{256}$, $\frac{56}{256}$ i $\frac{37}{256}$. Istnieje 5^5 możliwych kombinacji tworzących słowa, z ciągu 256000 słów wyliczamy częstotliwość z jaką pojawia się każde ze słów. Następnie na podstawie danych wzorcowych przeprowadzamy test χ^2.

3.1.10 Count the 1s (byte) test

Test analogiczny do poprzedniego. Jedyna różnica to wybór danych, analizowany bajt wybiera się jako jeden z czterobajtowej liczby. Bajt wybierany z liczby jest cyklicznie zmieniany na inny.

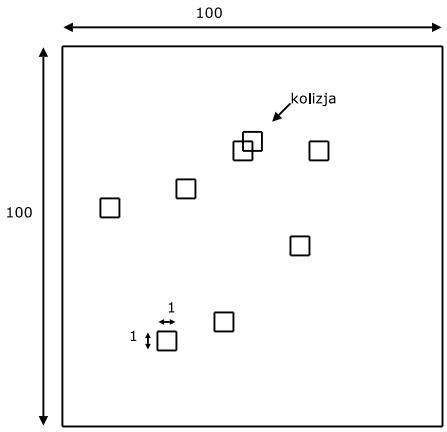
3.1.11 Parking Lot test

Test o dość intrygującej nazwie, polega na próbie zaparkowania kwadratowego samochodu o długości boku równej jednej jednostce na parkingu o wymiarach 100x100. Liczby losowe są interpretowane jako współrzędne gdzie ma zostać zaparkowany samochód. Wyniki liczbowe to n liczba prób zaparkowania i k liczba prób udanych gdzie nie „rozbiliśmy" samochodu o już zaparkowany.

Test nie został przeanalizowany całkowicie analitycznie i wzorcowe wartości zostały wyznaczone eksperymentalnie: $n = 12000$, k: $\mu = 352$, $\sigma = 21.9$ rozkład k powinien przypominać rozkład normalny.

3.1.12 Minimum Distance (2D Spheres) test

W teście wybierane jest $n = 8000$ losowych punktów w kwadracie o boku 10000. Obliczone jest d - minimalna odległość pomiędzy $\frac{n^2-n}{2}$ parami punktów. Jeżeli punkty są rozmieszczone niezależnie losowo, wtedy wartość d^2 - kwadratu minimalnej odległości powinna mieć rozkład podobny do eksponencjalnego z wartością oczekiwaną $m = 0.995$. Co po znormalizowaniu przy średniej $1 - e^{\frac{-d^2}{0.995}}$ powinno przedstawiać rozkład równomierny.



Rysunek 1: Parking Lot test

3.1.13 3D Spheres (minimum distance) test

Test pod niektórymi względami bardzo podobny do poprzedniego. Wybiera $n = 4000$ losowych punktów w trójwymiarowej przestrzeni w obrębie sześcianu o boku 1000. W każdym punkcie umieszcza sferę o promieniu równym odległości do najbliższego punktu. Objętość najmniejszej sfery podlega rozkładowi podobnemu do eksponencjalnego z wartością oczekiwaną $\frac{120\pi}{3}$. Sześcian promienia ma rozkład eksponencjalny z wartością oczekiwaną 30 (zostało to ustalone doświadczalnie). Po przekształceniu otrzymujemy do badania rozkład równomierny o średniej $1 - e^{\frac{-r^3}{30}}$.

3.1.14 Squeeze test

Liczby losowe w postaci liczb całkowitych są przekształcane na liczby zmiennoprzecinkowe ($n = \frac{n}{n_{max}+1}$, co efektywnie daje liczby z przedziału $[0, 1)$). Zaczynając od $k = 2^{31}$, test szuka wartości j, czyli liczby iteracji potrzebnych do „zredukowania" k do wartości 1, używając wyrażenia:

$$k = \lceil k \cdot n \rceil$$

Taki cykl jest powtarzany 1000 razy, podczas tego zliczana jest ilość wystąpień poszczególnych końcowych wartości j, które potem są porównywane do danych otrzymanych doświadczalnie wobec generatora idealnego.

3.1.15 Sums test

Test jest analogiczny do poprzedniego, losowe liczby całkowite są przekształcane na liczby zmiennoprzecinkowe $U(1), U(2) \ldots$, nastepnie wylicza sie sumy pokrywających się podciągów składających się ze 100 elementów ciągu: $S(1) =$

$U(1) + \ldots + U(100)$, $S(2) = U(2) + \ldots + U(101), \ldots$. Otrzymany ciąg sum powinien mieć rozkład normalny.

3.1.16 Runs test

Test podobnie jak poprzednio najpierw przekształca ciąg liczb losowych całkowitych na liczby zmiennoprzecinkowe, potem analizuje otrzymany ciąg, przechodząc od pierwszej do ostatniej liczby, liczy przejścia gdzie przechodził z wartości mniejszej na większą (*run up*) i odwrotnie (*run down*). Przejścia są zliczane dla sekwencji 10000 liczb. Idealny przypadek zakłada równą ilość przejść w górę jak i w dół.

3.1.17 Craps test

Test rozgrywa 200000 gier „w kości". Rzut kostką przebiega następująco: liczby losowe są zamieniane na zmiennoprzecinkowe jak poprzednio, mnożone przez 6, następnie dodaje się do wyniku 1 i usuwa część ułamkową, otrzymując liczbę całkowitą z zakresu $[1, 6]$. Oblicza się sumę zwycięstw i liczbę rzutów potrzebnych do zakończenia gry. Rozkład liczby zwycięstw powinien być podobny do rozkładu normalnego o wartości oczekiwanej $\mu = 200000p$ z $\sigma = 200000p(1-p)$ gdzie $p = \frac{244}{495}$.

3.1.18 Marsaglia and Tsang GCD test

Uzyskuje się 10^7 par liczb losowych u i v. Za pomocą algorytmu Euklidesa oblicza się ich największy wspólny dzielnik:

$$NWD(a,b) = \begin{cases} a & \text{dla} \quad b = 0 \\ NWD(b, a \bmod b) & \text{dla} \quad b \geqslant 1 \end{cases}$$

i liczbę kroków potrzebną do zakończenia algorytmu. Generowane są tablice wyrażające częstość występowania poszczególnych wartości NWD i liczby kroków. Rozkład NWD jest dany rozkładem $P(i) = \frac{6}{i^2\pi^2}$, wartość liczby kroków jest dana rozkładem dwumianowym i jest porównywana z wzorcowymi wynikami.

3.2 Robert G. Brown's Tests

Testy są częścią pakietu DieHarder, zostaną omówione w oddzielnym podpunkcie ze względu na to, że nie są uznawane za „klasyczne" czyli nie są aż tak popularne jak wcześniejsze.

3.2.1 Timing test

Test szybkości generowania liczb losowych, pokazuje uśrednione wyniki pomiaru czasu generowania jednej liczby i ilość generowanych liczb na sekundę. Pewną prawidłowością jest to, że o wiele szybciej generowane są liczby pseudolosowe niż prawdziwie losowe, wynika to z tego, że liczby pseudolosowe są ograniczane przez szybkość algorytmu i maszyny na której pracuje, natomiast

liczby prawdziwie losowe muszą być zbierane z fizycznych zjawisk, których mierzalne skutki nie zawsze następują szybko lub często.

3.2.2 Bit Persist test

Test operuje na 256 sekwencjach 32 bitów, tworząc „maski" oznaczające, że w poprzednim kroku bit się nie zmienił, jeżeli pod koniec sekwencji maska zawiera jedynkę oznacza to, że jeden z bitów cały czas pozostawał taki sam, co źle świadczy o generatorze, który możemy odrzucić jako nie dający dobrych rezultatów. Test powstał jako odpowiedź na pojawienie się pewnych genratorów, które przechodziły popularne testy podczas gdy w środowisku docelowym wykazywały brak podstawowych własności.

3.2.3 Bit Distribution Test

Test oblicza częstości występowania wszystkich nie pokrywających się sekwencji bitowych (o możliwych długościach) w liście danych liczb losowych i porównuje z teoretycznie wygenerowanym rozkładem dwumianowym, wykonując test χ^2. Pozwala ocenić pewien „stopień" losowości generatora.

3.3 Statistical Test Suite[1]

Dwa testy nie mające bezpośrednio odpowiedników w oryginalnym diehard.

3.3.1 Monobit test

Test sumuje wystąpienia jedynki w ciągu testowanych liczb losowych, oblicza wartość p bezpośrednio, korzystając z funkcji błędu, zakładając że rozkład powinien mieć cechy rozkładu normalnego.

3.3.2 Runs test

Test analogiczny do *Runs test* z pakietu DieHarder, ale operujący na bitach.

3.4 Przykładowe testy

Jednym z założeń podczas tworzenia DieHarder była prostota obsługi i umożliwienie łatwego rozszerzania pakietu (a dokładniej jego funkcjonalności) o nowe testy i generatory. Podczas prac nad pakietem twórcy GSL[5] skontaktowali się z Robertem Brownem i udzielając swojej pomocy zaproponowali kierunek rozwoju DieHarder, który miał uzupełniać funkcjonalność GSLa co zaowocowało tym, że pakiet w wersji podstawowej udostępnia w przeważającej większości generatory liczb losowych zawarte w pakiecie GSL. Jak zostało wspomniane wcześniej testy pozwalając określić, który generator najprawdopodobniej nie jest dobry, dokładniejsza interpretacja wyników jest trudna, otrzymywana p-wartość może powodować odrzucenie generatora, ale nie świadczy bezpośrednio o jego jakości,

dobre generatory pseudolosowe często zbliżają się do p-wartości 1, podczas gdy cechą naturalnych generatorów jest to, że rzadko zbliżają się do tej wartości.

3.4.1 Testowane generatory

Przetestowanie wszystkich generatorów wszystkimi testami zajęło by zbyt wiele czasu, dlatego wybrałem generatory które pokażą „przekrój" przez jakość obecnie dostępnych.

- IBM RANDU którego następna liczba losowa jest generowana na podstawie wzoru:

  $$x_{n+1} = (a \cdot x_n) \bmod m$$

  gdzie $a = 65539$, $m = 2^{31}$. Okres po którym wartości zaczynają sie powtarzać wynosi 2^{29}, sam generator jest uznawany za podręcznikowy przykład złego generatora.
- `/dev/urandom` - czyli generator liczb pseudo losowych linuksa. W zależności od wersji jądra mogą wystąpić drobne zmiany w działaniu, ale jego główne założenie opiera się na wykorzystaniu liczb prawdziwie losowych z `/dev/random`, które są dostępne ale w ograniczonej ilości. W przypadku wyczerpania puli, generator zaczyna generować liczby na podstawie algorytmu i w raz z uzupełnianiem puli w `random` pobiera z niego wartości aby nadać generowanemu ciągowi lepszą jakość.
- glibc2 - czyli generatory odpowiedzialne za funkcję `random()` w bibliotece standardowej języka C. Ich implementacja jest dana wzorcem:

  ```
  static unsigned long next = 1;
  int rand(void) {
      next = next * 1103515245 + 12345;
      return((unsigned)(next/65536) % 32768);
  }
  ```

  Implementacja w glibc jest podobna, ale autorzy twierdzą, że nie występuje tam problem słabej losowości mniej znaczących bitów w generowanej 32 bitowej liczbie.
- rand48 czyli ulepszona wersja podstawowego generatora glibc. Operująca na 48 bitowych liczbach całkowitych, stąd oczywista poprawa parametrów, wzór jest następujący:

  $$X_{n+1} = (a \cdot X_n + c) \bmod m, \quad \text{gdzie } n \geqslant 0$$

  gdzie $a = 25214903917$, $c = 11$, $m = 2^{48}$. W razie potrzeby zwracana wartość jest przycinana do pierwszych 32 bitów.
- ranlxd2 - jeden z najlepszych generatorów w bibliotece GSL. Został dogłębnie przeanalizowany matematycznie, stwierdzono że generuje prawdziwie nieskorelowane kolejne wartości. Jego okres wynosi 10^{171}.

3.4.2 Test szybkości

generator	czas generowania jednej liczby [ns]	ilość liczb generowana na sekundę
randu	$2.849990e+01$	$3.508784e+07$
urandom	$1.993932e+03$	$5.015216e+05$
rand	$5.997700e+01$	$1.667306e+07$
rand48	$5.555700e+01$	$1.799953e+07$
ranlxd2	$5.850280e+02$	$1.709320e+06$

Oczywiście dokładne liczby nie są interesujące, najważniejszym czynnikiem jest rząd wartości, który może nam sugerować, że test generatora `urandom` może być znacznie dłuższy niż *randu*. Najwolniejszym jest generator nie całkowicie pseudo losowy czyli `urandom`, najprawdopodobniej pobieranie wartości z `random` zabiera mu dużo czasu. Wśród pozostałych generatorów wyróżnia się jeszcze *ranlxd2*, który jest najbardziej skomplikowany, ale w zamian (według dokumentacji) daje najlepszą jakość.

3.4.3 Test urodzin

Test przyjmuje przedział ufności równy 5%.

generator	finalna p-wartość	wynik
randu	0.839485	PASSED
urandom	0.461496	PASSED
rand	0.796819	PASSED
rand48	0.531307	PASSED
ranlxd2	0.409497	PASSED

Wszystkie generatory uczestniczące w teście zdały go.

3.4.4 Count the 1s (stream)

Test przyjmuje przedział ufności równy 5%.

generator	finalna p-wartość	wynik
randu	0.000000	FAILED
urandom	0.960652	PASSED
rand	0.000000	FAILED
rand48	0.449355	PASSED
ranlxd2	0.433876	PASSED

Test generatora *rand* został powtórzony parokrotnie dając ten sam rezultat, czyli poprzeczka stawiana przez ten test przepuściła tylko „lepsze” generatory, w tym najkorzystniejszy wynik uzyskał generator korzystający z liczb nie pseudolosowych.

3.4.5 Craps Test

Test przyjmuje przedział ufności równy 5%.

generator	finalna p-wartość	wynik
randu	0.000000	FAILED
urandom	0.514265	PASSED
rand	0.985002	PASSED
rand48	0.999260	PASSED
ranlxd2	0.924036	PASSED

Tutaj odpadł tylko najsłabszy generator.

3.4.6 Bitstream Test

Test przyjmuje przedział ufności równy 5%.

generator	finalna p-wartość	wynik
randu	0.000001	FAILED
urandom	0.596022	PASSED
rand	0.000001	FAILED
rand48	0.404916	PASSED
ranlxd2	0.869964	PASSED

Analogicznie jak w jednym z poprzednich testów, odpadły najsłabsze generatory.

3.4.7 Przykład wyników szczegółowych

Dla każdego przebiegu testu pakiet prezentował szczegółowe wyniki w następującej postaci:

```
#=================================================================
#          Diehard Count the 1s (stream) (modified) Test.
# ...
# <tutaj znajduje się krótki opis testu>
# ...
#=================================================================
#                         Run Details
# Random number generator tested: rand48
# Samples per test pvalue = 256000 (test default is 256000)
# P-values in final KS test = 100 (test default is 100)
#=================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#     20|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |    |
#     18|    |    |    |    |    |    |    |    |    |    |
#       |    |    |    |    |    |    |    |    |    |****|
#     16|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |    |    |    |    |    |    |****|
#     14|    |    |    |    |    |    |    |    |    |****|
#       |    |    |    |****|    |    |    |    |    |****|
#     12|    |    |    |****|    |    |    |    |****|****|
#       |    |    |    |****|    |    |    |****|****|****|
#     10|    |****|    |****|****|    |    |****|****|****|
#       |    |****|    |****|****|    |****|****|****|****|
#      8|****|****|    |****|****|    |****|****|****|****|
#       |****|****|    |****|****|    |****|****|****|****|
#      6|****|****|****|****|****|    |****|****|****|****|
#       |****|****|****|****|****|    |****|****|****|****|
#      4|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#      2|****|****|****|****|****|****|****|****|****|****|
#       |****|****|****|****|****|****|****|****|****|****|
#       |-----------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#=================================================================
#                           Results
# Kuiper KS: p = 0.302501
# Assessment: # PASSED at > 5% for Diehard Count the 1s (stream) Test
```

Prezentowany histogram pozwala stwierdzić samemu w jakim stopniu generator prezentuje oczekiwany ciągły rozkład p-wartości dla 100 powtórzeń testu. W

przypadku odnalezienia zależności kształtu histogramu od parametrów testu, możemy odrzucić generator jako generujący liczby pseudo losowe zależne od środowiska, czyli niskiej jakości. Pokazywany test wypadł pozytywnie, z wyników prezentowanych przez program możemy odczytać informacje, że test dotyczył generatora *rand48*, działał na próbce 256000 wartości losowych na test i został powtórzony 100 razy. Przykład negatywnego wyniku wygląda następująco:

```
#==================================================================
#                         Run Details
# Random number generator tested: random-glibc2
# Samples per test pvalue = 256000 (test default is 256000)
# P-values in final KS test = 100 (test default is 100)
#==================================================================
#               Histogram of p-values
# Counting histogram bins, binscale = 0.100000
#    100|    |    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     90|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     80|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     70|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     60|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     50|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     40|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     30|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     20|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#     10|****|    |    |    |    |    |    |    |    |    |
#       |****|    |    |    |    |    |    |    |    |    |
#       |-------------------------------------------------
#       | 0.1| 0.2| 0.3| 0.4| 0.5| 0.6| 0.7| 0.8| 0.9| 1.0|
#==================================================================
#                           Results
# Kuiper KS: p = 0.000000
# Assessment: FAILED at < 0.01% for Diehard Count the 1s (stream) Test
```

Testowany generator *randu* nie spełniał wymogów pojedynczych testów (minimalna p-wartość), co w efekcie dało histogram zapełniony w jednym przedziale, który nie jest zgodny z rozkładem równomiernym.

3.4.8 Podsumowanie wyników

Wyniki sugerują, że najlepsze z generatorów pseudolosowych to *ranlxd2* i `urandom`, które niestety są co najmniej o rząd wolniejsze niż pozostałe generatory. Dobrym rozwiązaniem zdaje się być *rand48*, który zakończył pozytywnie wszystkie testy, przy tym nie ustępując pod względem wydajności najszybszym.

References

1. **NIST** Special Publication 800-22, *http://csrc.nist.gov/rng/* 15 maj, 2001.
2. Robert G. Brown: **DieHarder**: A Random Number Test Suite, *http://www.phy.duke.edu/~rgb/*
3. **ENT** A Pseudorandom Number Sequence Test Program http://www.fourmilab.ch/random/
4. William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery: Numerical Recipes in C, The Art of Scientific Computing Second Edition *CAMBRIDGE UNIVERSITY PRESS*
5. **GSL** - GNU Scientific Library, *Free Software Foundation, Inc.* http://www.gnu.org/software/gsl/