Wydział Informatyki i Telekomunikacji
Politechnika Wrocławska

Generowanie liczb losowych w oparciu o czas dostępu do zasobów sieciowych

Marek Bauer
Nr indeksu: 250076

Praca inżynierska napisana
pod kierunkiem
dr. hab. Szymona Żeberskiego

Politechnika
Wrocławska

Wrocław 2021

Streszczenie

Celem pracy jest stworzenie prostego generatora jednostajnych i niezależnych liczb losowych. Źródłem entropii dla niego będą czasy opóźnień, które powstają podczas przesyłania pakietów przez Internet. Zostaną zaprezentowane wyniki testów wydajności, jak i losowości tego podejścia. Losowość zostanie sprawdzona w oparciu o testy statystyczne zaproponowane w tej pracy, a wyniki niniejszego generatora porównane do trzech znanych na rynku rozwiązań różnej klasy.

Abstract

The purpose of this thesis is to create a simple random number generator, whose outputs meet the requirements of uniformity and independence. The source of entropy for our generator are delay times that arise during transmission of packets over the Internet. We present the results of efficiency and randomness tests of our approach. Randomness is checked using statistical tests that we propose. We compare our approach with three alternatives from the literature.

Spis treści

Cel pracy

Celem pracy jest stworzenie prostego systemu generującego liczby losowe, w tym celu zostaną wykorzystanie informacje o czasie dostępu do zasobów sieciowych (czas odpowiedzi ze strony serwera lub czas pobrania pliku z serwera itp.). Liczby losowe odnajdują zastosowanie między innymi w: kryptografii, statystyce, symulacjach komputerowych, jak również w tak przyziemnych branżach, jak gry komputerowe. Nie każdy ciąg zmiennych losowych jest jednak użyteczny w wyżej przedstawionych zastosowaniach. Powinien on spełniać dwie cechy:

jednostajność: w matematyce rozkładem jednostajnym określa się taki, w którym szansa wylosowania liczby z dowolnego przedziału $[a, b) \subseteq [0, 1)$ jest równa $b - a$, czyli mierze Lebesgue'a. Komputery są maszynami dyskretnymi i skończonymi zatem nasz system będzie generował liczby o rozkładzie dyskretnym jednostajnym, a więc liczby naturalne od 0 do $2^{64} - 1$ włącznie, z prawdopodobieństwem wylosowania każdej z nich równym $\frac{1}{2^{64}}$.

niezależność: ciąg niezależnych zmiennych losowych, charakteryzuje się tym że wszystkie zmienne losowe są niezależne parami, trójkami, czwórkami (...). W naszym przypadku oznacza to, że poprzednie wyniki w żaden sposób nie wpływają na kolejne. Znając więc dowolną liczbę poprzednich wyników, nie będziemy mogli w żaden lepszy sposób przewidzieć kolejnego niż to, że szansa wylosowania każdej liczby jest równa $\frac{1}{2^{64}-1}$. Jest to szczególnie ważny warunek w kryptografii, gdzie niemożliwość przewidzenia kolejnego wyniku jest podstawą bezpieczeństwa.

Te dwie cechy możemy uprościć do następującej formuły:

$$(\forall i \in \mathbb{N})(\forall k \in \{0, 1, \ldots, 2^{64} - 1\})P(X_i = k | X_{i-1}X_{i-2} \ldots X_0) = \frac{1}{2^{64}} \tag{1.1}$$

Mając ciąg je spełniający, w łatwy sposób wygenerować można niezależny ciąg zmiennych losowych o innych rozkładach, używając takich algorytmów jak Boxa-Mullera czy prostego odrzucania próbek. Stąd też te dwa warunki są wystarczające we wszystkich znanych zastosowaniach generatorów liczb losowych.

Analiza problemu

Rozdział ten skupia się na analizie dotychczasowych podejść do problemu generowania liczb losowych, jak również zarysowaniu koncepcji naszego podejścia do problemu. Dodatkowo zostanią tu również zaprezentowane podstawowe trudności wynikające z wybranej metody, wraz z ich konsekwencjami dla implementacji generatora.

2.1 Czy można stworzyć prawdziwy generator liczb losowych?

Zanim przejdziemy do stworzenia konceptu warto zadać sobie pytanie, czy w ogóle generator spełniający założenia pracy może istnieć? Naszą wątpliwość możemy sprowadzić do problemu, czy świat jest deterministyczny, czy też nie. Naturalnie w deterministycznym świecie każda akcja bezpośrednio wynika ze stanu początkowego. Znając go, w naszym przypadku stan całego wszechświata, moglibyśmy zasymulować wszystkie kolejne stany, a co za tym idzie również wyniki naszego systemu. Przy takich założeniach każdy generator byłby jedynie generatorem liczb pseudolosowych, z bardziej lub mniej złożoną zasadą działania. Jeśli natomiast nasz świat nie jest deterministyczny, możemy użyć tej niepewności do generowania prawdziwie losowego ciągu.

2.1.1 Debata nad wolną wolą

Jedynym z pierwszych miejsc, w których możemy znaleźć niepewność we wszechświecie jesteśmy my sami, a bardziej precyzyjnie - ludzka wolna wola, czyli zdolność do podjęcia w tych samych okolicznościach i z tymi samymi wspomnieniami, przekonaniami, myślami itp. dwóch różnych decyzji. Założenie wolnej woli jest dla ludzi bardzo intuicyjne; każdy z nas przecież czuje się zdolnym do podejmowania różnych decyzji. Jedną z nich był wybór przeczytania niniejszej pracy nad wszystkimi innymi możliwościami spędzenia tego czasu. Jednak oprócz naszego poczucia wolnej woli trudno znaleźć naukowe przesłanki świadczące o niej. Już od starożytności ludzkość zadaje sobie pytanie o to czy ma wpływ na swój własny los. Dzieła takie jak *Król Edyp* [7] autorstwa Sofoklesa przedstawiają wizję, w której wszytko jest z góry zdeterminowane, a ucieczka przed przeznaczeniem (determinizmem wszechświata) jest niemożliwa. Jednym z największych krytyków idei wolnej woli był francuski filozof z epoki oświecenia Paul d'Holbach, który w swoim dziele *Systemy przyrody* [10] stwierdził, że ludzki mózg poddaje się prawom fizyki jak każdy inny fizyczny obiekt, więc tak jak inne obiekty poddaje się determinizmowi, a nasze decyzje możemy sprowadzić do wypadkowej naszych przekonań, pragnień oraz naszego usposobienia. Uznał zatem, że istnienie wolnej woli stoi w sprzeczności z prawami natury.

2.1.2 Fizyka kwantowa jako źródło entropii

Od czasów dzieł Paula d'Holbacha dokonaliśmy sporego rozwoju naszego zrozumienia fizyki. Pojawiła się nieznana w jego czasach mechanika kwantowa. Obecnie konsensus naukowy mówi, że pozycja cząstek elementarnych nie jest jednym miejscem, a raczej funkcją prawdopodobieństwa [8], gdzie może się ona znajdować. Zakładając, iż rzeczywiście tak jest można by wykorzystać to źródło niepewności do generowania liczb losowych. Jednak fizyka wciąż się rozwija i istnieje spore ryzyko, że tak naprawdę ta losowość jest jedynie pewnym uproszczeniem bardziej skomplikowanego modelu, którego jeszcze nie rozumiemy. Dlatego też pytanie na temat determinizmu

świata nadal pozostaje otwarte i najprawdopodobniej nigdy nie poznamy na nie jednoznacznej odpowiedzi.

2.2 Dostępne rozwiązania problemu generowania liczb losowych

Problem generowania liczb losowych jest prawie tak stary jak same komputery. Pierwsze rozwiązania tego zagadnienia przypisujemy Johnowi von Neumannowi, który w 1949 roku opisał swój pomysł generowania liczb o trudnym do przewidzenia wzorcu (*metoda środkowego kwadratu* [23]). Od tego czasu powstały dwa różne podejścia do tego problemu: generatory liczb pseudolosowych oraz "prawdziwe" generatory liczb losowych.

2.2.1 Generatory liczb pseudolosowych

Są to metody algebraiczne polegające na generowaniu ciągu liczb opisanego pewnym wzorem. Jest on na tyle skomplikowany, aby wydawało się, iż elementy tego ciągu nie mają żadnego związku między sobą. Użycie następującego rozwiązania ma wiele zalet:

uniwersalność: z uwagi na to, że rozwiązanie opiera się jedynie na operacjach arytmetycznych może być ono zastosowane na każdej programowalnej maszynie liczącej, uwzględniając bardzo prymitywne mikrokontrolery. Umożliwia to również uruchamianie takiego programu w środowisku chronionym, czyli takim w którym program nie powinien mieć dostępu do informacji z zewnątrz. Cechy te sprawiają, iż praktycznie każdy język programowania ma wbudowany jakiś generator liczb pseudolosowych czyniąc je łatwym wyborem dla programistów.

wysoka wydajność: zdecydowana większość rozwiązań tego typu charakteryzuje się bardzo szybką metodą generowania kolejnych elementów ciągu. Wygenerowanie ciągu długości n zajmuje asymptotycznie $O(n)$ czasu i $O(1)$ pamięci. Kolejnym aspektem jest to, że przy tego typu generatorach ogranicza nas jedynie wydajność procesora. Dlatego też możemy łatwo wykorzystać cały potencjał naszego sprzętu przy rozwiązywaniu problemów wymagających dużej liczby liczb losowych. Dobrym przykładem są tutaj algorytmy wykorzystujące metodę Monte Carlo, wymagają one dużej ilości próbek, stąd też w większości decydują się na pseudolosowość.

jednostajność: jako że tego typu generatory są w pełni deterministyczne, możemy często udowodnić iż mają one oczekiwany jednostajny rozkład. Jest to duża zaleta w zastosowaniu tego podejścia w wyżej wspominanych algorytmach wykorzystujących metodę Monte Carlo. Pozwala ona uniknąć nieprawidłowości wynikających z nadreprezentacji pewnych liczb.

Pomimo tylu zalet rozwiązanie to nie jest pozbawione kilku istotnych wad:

problem stanu początkowego (seedu): ciągi pseudolosowe są deterministyczne, a więc dla tych samych warunków początkowych wygenerowane ciągi będą identyczne. Stąd pojawia się potrzeba wygenerowania stanu początkowego wykorzystując inną metodę generowana liczb losowych. Najczęściej w tym celu stosuje się czas uruchomienia, co może być jednak zgubne w przypadku programów uruchamianych cyklicznie. Zastosowanie generatorów tego typu do rozpoczęcia prawidłowej pracy wymaga odrobiny entropii z zewnątrz, na szczęście nie musi ona spełniać tak rygorystycznych wymagań jak nasz generator. Osłabia to jednak nieco uniwersalność tego rozwiązania.

brak niezależności: jest to największy problem tego podejścia. Jako iż nasz generator wykonuje jedynie szereg deterministycznych obliczeń, znając warunki początkowe i użyty

algorytm, możemy bez problemu wygenerować identyczny ciąg. Bardziej wyrafinowane współczesne generatory tego typu posiadają rozbudowany stan wewnętrzny, a więc wyznaczenie kolejnej liczby na podstanie jedynie kilku poprzednich jest niemożliwe. Jednak przy pozyskaniu odpowiedniej liczy próbek z generatora możliwe jest wyznaczenie jego stanu wewnętrznego, a co za tym idzie, również odtworzenie identycznego ciągu jaki produkuje nasz generator. Stąd też w zastosowaniach kryptograficznych nie zalecane jest używanie generatorów liczb pseudolosowych.

zapętlanie się: problem ten wynika z prostej obserwacji, iż w komputerze generator może mieć jedynie skończoną liczbę stanów wewnętrznych oraz tego, iż stan wewnętrzny w pełni determinuje kolejne uzyskiwane wyniki. Tak więc dla każdego generatora i każdego stanu początkowego istnieje taka liczba l, że $(\forall k \in \mathbb{N})X_k = X_{k+l}$. W bardziej zaawansowanych generatorach liczba l jest bardzo wysoka, rzędu 2^{100}, co niweluje ten problem.

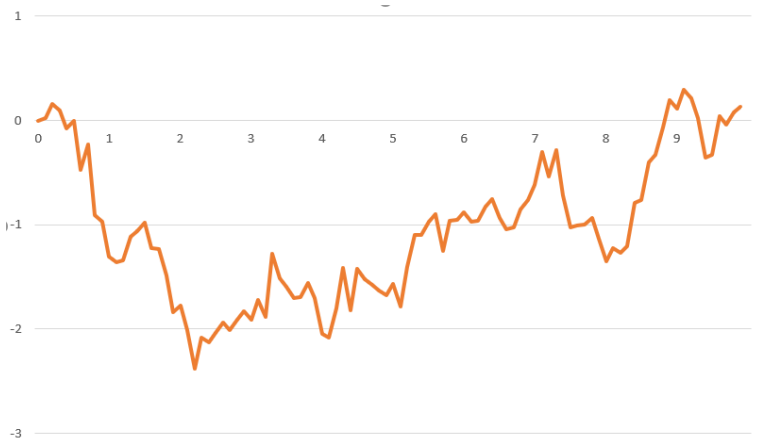
2.2.2 "Prawdziwe" generatory liczb losowych

Rozwiązania z tej kategorii opierają swoją losowość na informacjach pochodzących ze świata zewnętrznego. Ilość czynników jakie oddziałują na środowisko, w którym działa maszyna jest olbrzymia, stąd też istnieje niesamowita różnorodność podejść w tworzeniu tego typu generatorów. Jednymi z najpopularniejszych są: pomiar różnicy temperatury, wsłuchiwanie się w szum elektromagnetyczny lub, jeśli to możliwe, obserwowanie zachowania użytkownika systemu. Warto wymienić tutaj podstawową zaletę tej metody:

prawdziwa niezależność: to podejście pozwala nam spełnić drugie z wymienionych wymagań dotyczących generatorów liczb losowych. Nawet jeśli świat byłby deterministyczny, to zjawisko jakim jest efekt motyla generuje dostatecznie dużo niepewności, żeby żaden komputer stworzony przez człowieka nie był w stanie przewidzieć rezultatów. Najlepszym z "dowodów" na to jest fakt, iż pomimo lat starań i dużego zapotrzebowania na długoterminowe prognozy meteorologiczne nikt nie umie przewidzieć pogody na okres dłuższy niż 2 tygodnie. Wyobraźmy sobie generator oparty na temperaturze powietrza. Na potrzeby jego analizy nieco uprośćmy obraz świata zewnętrznego. Powiedzmy, że na zmianę temperatury wpływa jakaś ogromna liczba M binarnych zdarzeń takich jak to, czy motyl w parku lata czy nie. Stosując centralne twierdzenie graniczne wiemy, że rozkład prawdopodobieństwa temperatury w następnej chwili powinien być podobny do rozkładu normalnego, a co za tym idzie na odpowiednio małym wycinku czasu przypominać będzie jednowymiarowy ruch Browna. A więc na 50% temperatura powinna się podnieść i na 50% powinna się obniżyć.

Podejście to jest nie jest jednak pozbawione wad:

wolne generowanie liczb: głównym problemem w tym podejściu jest fakt, że niezależnie od mocy obliczeniowej urządzenia możemy uzyskać tylko ograniczoną ilość próbek w ciągu sekundy. Pomimo założenia, iż parametry świata rzeczywistego mają charakter ciągły tzn. nasza temperatura płynnie zmienia się w czasie, to jednak komputery są maszynami skończonymi. Co za tym idzie nie są w stanie dokładnie odczytać parametru ze świata zewnętrznego, a jedynie jego zniekształcony cyfrowy obraz. Ma on tą cechę, iż rzutuje całe przedziały np. $[0, 1)$ na pojedynczą wartość 0. Jak możemy zaobserwować na rysunku 2.1 pomimo, iż wysokość wykresu ciągle się zmienia, to często wahania te pozostają w granicach przedziału precyzji. A więc wykonanie pomiaru w 6 i 7 sekundzie wyprodukuje identyczny rezultat. Stąd też wymagają czasu, aby temperatura zmieniała się na tyle, żeby nasz system był w stanie wykryć różnicę. Częstym problemem jest również sama ilość próbek, które urządzenie pomiarowe jest w stanie w ciągu sekundy wygenerować urządzenie pomiarowe.



Rysunek 2.1: Przykładowy jednowymiarowy proces Weinera (ruch Browna)

ryzyko manipulowania wynikami: świat zewnętrzy jest niemożliwy do kontrolowania z punku widzenia twórcy sytemu. Istnieje więc ryzyko, iż osoba posiadająca dostatecznie dużo wiedzy na temat działania pewnego systemu tworzącego liczby losowe będzie w stanie manipulować warunkami środowiskowymi na tyle, żeby wygenerował oczekiwany przez nią rezultat. Możemy np. ciągle ogrzewać czujnik temperatury tak, żeby nigdy nie pokazał, iż temperatura spada lub wręcz sprawić, żeby przekroczył on zakres swoich odczytów.

dostęp do świata zewnętrznego: w celu pobrania informacji o jakimś parametrze ze świata zewnętrznego konieczny jest czujnik, który jest w stanie odczytać jego wartość. Co oczywiście, nie jest możliwe zmierzenie temperatury bez termometru. W celu generowania odpowiedniej liczby próbek potrzebny jest jednak odpowiedniej klasy sprzęt pomiarowy podnosząc tym samym koszty produkcji urządzenia, który takowy posiada. Stąd tego typu moduły nie są popularne, zwłaszcza w tanich urządzeniach. Pewnym obejściem problemu jest wykorzystanie użytkownika jako źródła entropii dla systemu. Jest to jednak rozwiązanie relatywnie wolne i może być zastosowane jedynie w systemach, w których występuje interakcja z użytkownikiem.

niepewność jednostajności liczb losowych: oderwanie się od matematycznych wzorów powoduje, iż nie możemy przeprowadzić dowodu poprawności działania takich generatorów. Sprawia to, że nie możemy być pewni parametrów rozkładu naszych zmiennych losowych. Zapewnienie jego jednostajności jest jedną z największych trudności w tym podejściu.

2.3 Koncept generatora

Generator spełniający oba wymagania sprecyzowane w celu tej pracy musi należeć do kategorii ”prawdziwych” generatorów liczb losowych. Tak więc będzie on wykorzystywać informacje o świecie zewnętrznym celem generowania kolejnych liczb losowych.

2.3.1 Źródło entropii

W tym celu wykorzystamy opóźnienia w przesyłaniu pakietów przez sieć globalną. Jednym powodem ich powstawania na drodze naszego pakietu będzie obciążenie hosta docelowego. Jest on praktycznie zupełnie poza naszą kontrolą i wynika z liczby użytkowników chcących pobrać z niego informacje, a więc jest wypadkową tysięcy niezależnych od nas decyzji ludzkich jak i również zautomatyzowanych systemów. Pomiędzy naszym komputerem, a hostem docelowym

występuje pewna liczba komputerów (routerów) odpowiedzialnych za przekierowywanie pakietów do celu. Każdy z tych routerów również generuje opóźnienia wynikające z ich obecnego obciążenia. Tak więc nawet jeśli jako jedyni chcemy sprawdzić ceny akcji firmy *Apple*, to na czas uzyskania tych informacji ma również wpływ sąsiadka wyszukującą w Internecie filmiki z kotami. Ostatnim z źródeł opóźnień na drodze pakietu jest czas jego przetworzenia przez nasz komputer co więcej, że warunki środowiskowe takie jak temperatura powietrza, opady atmosferyczne czy nawet promieniowanie kosmiczne na trasie pakietu mają również pewien, choć bardzo niewielki wpływ na jego czas podróży.

2.3.2 Przekształcenie próbek na liczby losowe

Posiadając już odpowiednio losowe źródło entropii, którym dla naszego przypadku będzie dodatnia liczba naturalna określająca czas podróży pakietu w pewnej jednostce czasu, należy je jeszcze odpowiednio przetworzyć, aby zapewnić oczekiwany jednostajny rozkład liczb losowych. W przypadku tej pracy, w tym celu "obetniemy" każdą próbkę do najmniej znaczących bitów, używając w tym celu operacji modulo. Część z bitów naszej próbki powinna mieć rozkład statystycznie nieróżniący się od rozkładu jednostajnego.

2.4 Występowanie prawa Benforda w rozkładzie czasów dostępów

2.4.1 Definicja

Rozkład zmiennej losowej X spełnia prawo Benforda wtedy i tylko wtedy gdy $Lead(X)$ (funkcja zwracająca najbardziej znaczącą cyfrę) spełnia warunek 2.1:

$$P(\text{Lead}(X) = d) = \log_b(d+1) - \log_b(d) = \log_b\left(1 + \frac{1}{d}\right) \tag{2.1}$$

gdzie b oznacza podstawę systemu liczbowego w którym zapisana jest dana liczba.

2.4.2 Historia

Pierwszym opublikowanym artykułem na temat tego fenomenu jest *Note on the frequency of use of the different digits in natural numbers* (tłum. *Uwaga na temat częstotliwości występowania różnych cyfr w liczbach naturalnych*) [16]. W nim właśnie Simon Newcomb podzielił się obserwacją, iż najbardziej znaczące cyfry (1-9) nie występują w jednakowej proporcji. Swojego odkrycia dokonał, gdy zauważył że pierwsze strony książki zawierającej wyliczone logarytmy są bardziej zniszczone niż dalsze. Zaproponował on przedstawioną wyżej formułę opisującą to zjawisko. Kolejnym matematykiem zajmującym się tym problemem był Frank Benford. Na podstawie danych z kilkunastu dostępnych baz danych zauważył on, że około połowa z nich spełnia nazwane od jego nazwiska prawo Benfoda [5]. Formalizacji tego fenomenu dokonał amerykański matematyk Ted Hill [9].

2.4.3 Uogólnienie na dalsze cyfry

Posiadając podaną wyżej definicję prawa Benforda możemy łatwo uogólnić ją na dalsze cyfry. Aby tego dokonać należy zmienić podstawę systemu z b na b^k (gdzie k jest k-tą najbardziej znaczącą cyfrą). Teraz możemy obliczyć prawdopodobieństwo występowania każdej k cyfrowej kombinacji na początku. Należy jednak pamiętać, że dopuszczalne kombinacje muszą być większe bądź równe b^{k-1}, gdyż w przeciwnym wypadku na pierwszej pozycji takiej kombinacji będzie

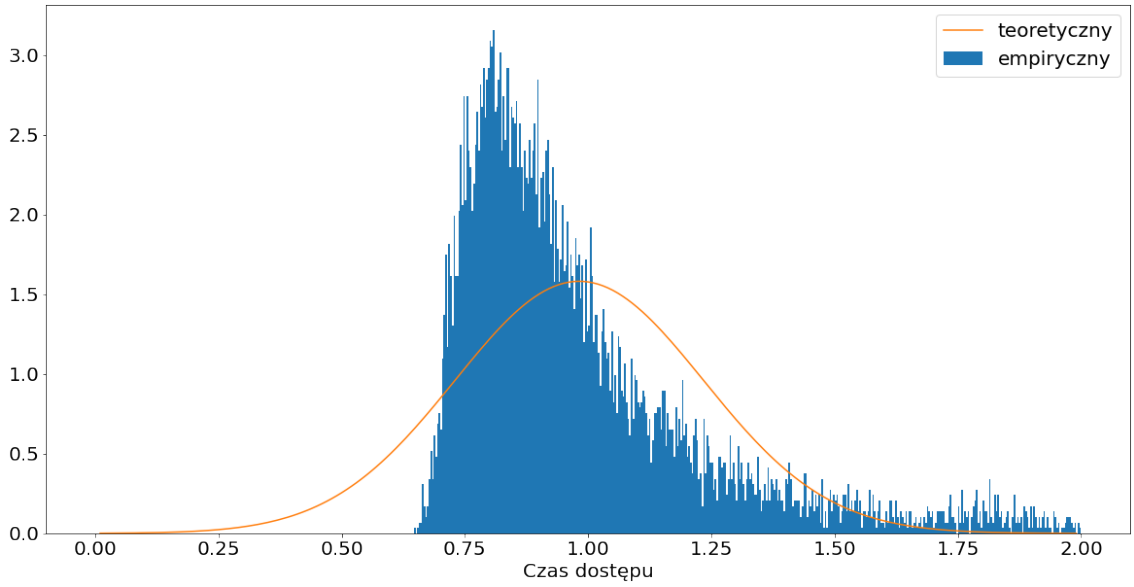
występowało zero, a więc nie będzie to najbardziej znacząca cyfra. Z tych obserwacji możemy wyciągnąć formułę 2.2.

$$\begin{aligned} &P(\mathrm{Lead}_k(X) = d) = \\ &= \log_b\left(1 + \frac{1}{b^{k-2} + d}\right) + \log_b\left(1 + \frac{1}{b^{k-2} + b + d}\right) + \cdots + \log_b\left(1 + \frac{1}{b^{k-1} - b + d}\right) = \\ &= \sum_{n=b^{k-2}}^{n=b^{k-1}-1} \log_b\left(1 + \frac{1}{bn + d}\right) \end{aligned} \tag{2.2}$$

Jak widzimy, obliczenie prawdopodobieństwa każdej kolejnej z cyfr jest wykładniczo bardziej złożone.

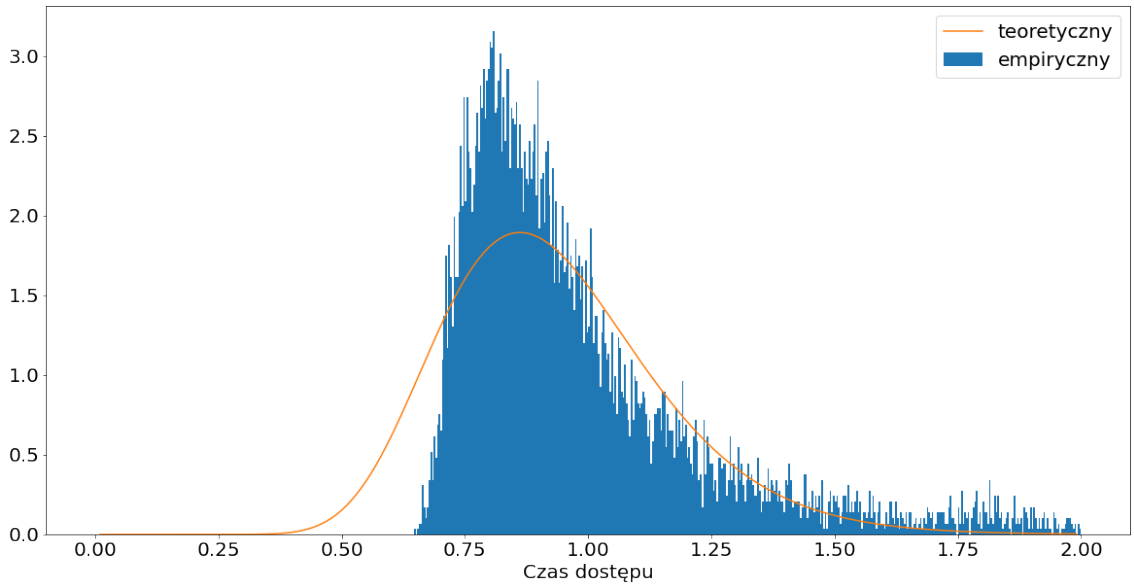
2.4.4 Obserwacja

Zgodnie z naszymi założeniami początkowymi moglibyśmy się spodziewać, że ze względu na ogromną ilość czynników mających wpływ na czas dostępu jego rozkład powinien dążyć do rozkładu normalnego ze względu na centralne twierdzenie graniczne. Problematycznym w tej tezie jest natomiast fakt, iż czas dostępu nie może być ujemny. Tak więc dla wartości oczekiwanej rozkładu M, prawdą jest, iż $P(X \in [-M, 0]) < P(X \in [2M, 3M])$. Być może jednak wartości tak oddalone od wartości oczekiwanej i tak są pomijalne w całym rozkładzie. Jak możemy zobaczyć



Rysunek 2.2: Porównanie histogramu czasu dostępu do strony *https://www.spaceneedle.com/* z gęstością rozkładu noramlnego

na rysunku 2.2, rozkład teoretyczny nie pokrywa się empirycznym. Musimy zatem zrewidować swoją tezę. Dokonujemy obserwacji, że $P(X \in [\frac{1}{2}m, m]) \approx P(X \in [m, 2m])$. Możemy zatem postawić hipotezę, iż mamy do czynienia z rozkładem logarytmicznym, precyzyjniej postawimy hipotezę o rozkładzie log normalnym. Jak widzimy na wykresie 2.3 rozkład log normalny lepiej pasuje do empirycznego histogramu. Możemy zatem sprawdzić, czy wygenerowane liczby spełniają prawo Benforda. Jako, iż mamy do czynienia z komputerami, będziemy pracować z systemem liczbowym o podstawie 2.



Rysunek 2.3: Porównanie histogramu czasu dostępu do strony *https://www.spaceneedle.com/* z gęstością rozkładu log noramlnego

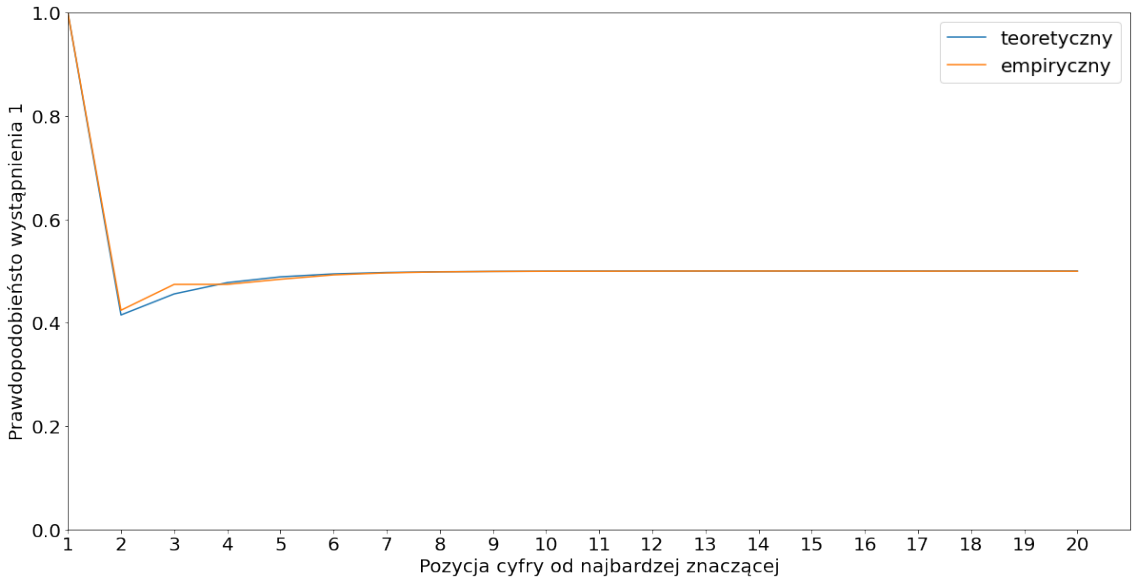
Pozycja	Prawdopodobieństwo wystąpienia 1
1	1
2	0.41503749927884376
3	0.4556794837761896
4	0.47755858383690225
5	0.48874172051300435
6	0.49436607559162077
7	0.49718243682965746
8	0.4985911432029018
9	0.4992955621971009
10	0.49982388981450276
...	...
20	0.4999996560346523
...	...
∞	0.5

Tablica 2.1: Rozkład prawdopodobieństwa występowania 1 na pozycji licząc od najbardziej znaczącej

W tablicy 2.1 możemy znaleźć teoretyczne prawdopodobieństwa występowania 1 na k-tym najbardziej znaczącym miejscu. Porównajmy zatem na rysunku 2.4 uzyskane próbki z teoretycznymi prawdopodobieństwami występowania jedynki na k-tej najbardziej znaczącej pozycji.

2.4.5 Konsekwencje

Jak możemy dostrzec na wykresie 2.4, prawo Benforda dość dobrze opisuje zaobserwowane odchylenie od rozkładu jednostajnego. Fakt ten rodzi problem przy próbie przekształcenia próbki na ciąg zmiennych losowych o binarnym rozkładzie ($P(X = 0) = \frac{1}{2} \wedge P(X = 1) = \frac{1}{2}$). Prosta konwersja liczby zmiennoprzecinkowej na ciąg bitów nie zapewni nam oczekiwanego rezkładu.



Rysunek 2.4: Porównanie teoretycznego prawdopodobieństwa występowania '1' na k-tej najbardziej znaczącej pozycji (linia niebieska), z empirycznym prawdopodobieństwem (linia pomarańczowa)

Aby zatem zniwelować ten problem podczas generowania liczb o rozkładzie jednostajnym, wymagane będzie odrzucenie pewnej ilości początkowych bitów, których rozkład będzie najbardziej zaburzony prawem Benforda. Spowoduje to oczywiste obniżenie wydajności generatora w zależności od ilości odrzuconych bitów początkowych z każdej próbki. Konieczne pozostanie zatem balansowanie pomiędzy jednostajnością rozkładu a wydajnością generatora.

2.5 Maksymalna precyzja pomiaru

2.5.1 Jak komputery liczą czas?

Współczesne komputery posiadają moduł *RTC* (zegar czasu rzeczywistego), który służy jednak głównie do wyświetlania aktualnego czasu, przez co jego precyzja nie jest wystarczająca aby mierzyć czas wykonywania operacji na komputerze. Dlatego też wciąż wykorzystuje się przerwania na procesorze wykonywane co pewną stałą ilość cykli, podczas których następuje inkrementacja licznika czasu. Nie mogą być one jednak zbyt częste, żeby nie obciążać niepotrzebnie maszyny. Oczywiście implementacja tego rozwiązania różni się mocno pomiędzy komputerami, stąd ten sam program może lepiej lub gorzej mierzyć czas w zależności od sprzętu, na którym jest uruchomiony. Dla procesorów o częstotliwości kilku giga Hertzów (10^9 operacji na sekundę) precyzyjne odmierzanie upływu nanosekund (10^{-9} części sekundy) zajęłoby większość taktów procesora. Stąd też na takich maszynach możemy spodziewać się jedynie precyzji rzędu mikrosekund (10^{-6} części sekundy).

2.5.2 Wpływ precyzji pomiaru na wydajność

Precyzja pomiaru w oczywisty sposób wpływa na ilość bitów jakie możemy uzyskać z jednej próbki. Jeśli przymniemy, że uzyskane czasy dostępu będą miały wartości od 16ms do 512ms, to przy precyzji rzędu mikrosekundy możemy uzyskać od 14 do 19 bitów informacji. Zakładając zatem wykorzystanie tylko jednego wątku do tego zadania możemy uzyskać do 875 bitów na sekundę, co jest niewielką wartością, zwłaszcza w porównaniu z generatorami liczb pseudolosowych, które generują kilka kilobajtów danych na sekundę.

2.5.3 Konsekwencje

Taka niewielka ilość bitów uzyskanych z jednej próbki w oczywisty sposób wymusza konieczność zebrania ich większej ilości w celu wygenerowania pojedynczej liczby losowej. Nawet przy założeniach, że wszystkie bity w tej próbce będą miały oczekiwany przez nas rozkład, przy 16 bitach informacji z każdej, będziemy potrzebować ich aż cztery, aby wygenerować 64 bitową liczbę. Każda próbka generuje obciążenie sieci. Jej przepustowość jest ograniczonym zasobem na każdej maszynie, co jest poważnym problemem w tej metodzie generowania liczb losowych.

Projekt i implementacja generatora

Rozdział ten jest poświęcony analizie technicznych trudności w implementacji generatora. Dodatkowo zostanie tutaj przedstawiony szczegółowy pseudokod umożliwiający implementację tego rozwiązania w praktycznie każdym języku programowania. Zaprezentowana zastanie również faktyczna implementacja w języku Python oraz analiza wydajności systemu.

3.1 Przekształcenie próbki na wektor bitów o jednostajnym rozkładzie

3.1.1 Sposób wyboru bitów z akceptowalnym rozkładem

Zakładając prawdziwość tezy, iż czasy dostępu spełniają prawo Benforda niemożliwe jest wygenerowanie ze skończonej liczby próbek jednego bitu losowego o rozkładzie jednostajnym. Stąd też obniżyć będziemy musieli nasze oczekiwania odnośnie rozkładu bitów. Zastosujemy prosty test statystyczny, który ocenia szansę uzyskania danej średniej z n próbek zakładając że pochodzą one z jednostajnego rozkładu bitowego. Przyjmiemy standardowy poziom istotności $\alpha = 0.05$, co oznacza, że w 95% przypadków ciąg bitów z docelowym rozkładem powinien go przejść.

3.1.2 Zastosowanie centralnego twierdzenia granicznego

Dla dużej liczby próbek $X_{i\in\{1,2,\dots,n\}}$ obliczenie dokładnego rozkładu zmiennej losowej $Y_k = \frac{1}{k}\sum_{i=1}^{n} X_i$ jest bardzo trudnym zadaniem. Dlatego też zastosujemy centralne twierdzenie graniczne. Mówi one, że dla rozkładu zmiennej losowej X, o skończonej wariancji:

$$\begin{aligned}\lim_{n\to\infty} \frac{\sqrt{n}}{\sigma n}\sum_{i=1}^{n}(X_i-\mu) &\sim \mathcal{N}(0,1)\\ \lim_{n\to\infty} \frac{\sqrt{n}}{\sigma n}\sum_{i=1}^{n}(X_i) &\sim \mathcal{N}(\mu,1)\\ \lim_{n\to\infty} \frac{1}{n}\sum_{i=1}^{n}(X_i) &\sim \mathcal{N}\left(\mu,\frac{\sigma}{\sqrt{n}}\right)\end{aligned} \tag{3.1}$$

W tej pracy będziemy pracować ze zmienną losową $X \in \{-1,1\}$ odpowiadającą dwóm stanom bitów:

$$P(X=1) = p \wedge P(X=-1) = 1-p \tag{3.2}$$

Szybko możemy obliczyć parametry tego rozkładu:

$$\begin{aligned}\mathbb{E}X &= 1p + (-1)(1-p) = 2p-1\\ Var(X) &= \mathbb{E}X^2 - [\mathbb{E}X]^2 = p + (1-p) - (2p-1)^2\\ &= 1 - 4p^2 + 4p - 1 = 4p(1-p)\\ \sigma &= 2\sqrt{p(1-p)}\end{aligned} \tag{3.3}$$

Zatem dla odpowiednio dużego n, będziemy przybliżać rozkład Y_n rozkładem normalnym o parametrach:

$$\mathcal{N}\left(2p-1, \frac{2\sqrt{p(1-p)}}{\sqrt{n}}\right) \tag{3.4}$$

3.1.3 Test statystyczny

Celem określenia, jak bardzo podobny jest rozkład bitu do rozkładu docelowego ($p = \frac{1}{2}$) obliczymy prawdopodobieństwo, iż średni wynik dla n zmiennych losowych z rozkładu będzie się mieścił w przedziale:

$$\left[\frac{1}{\sqrt{n}}\Phi^{-1}(0.025), \frac{1}{\sqrt{n}}\Phi^{-1}(0.975)\right]$$

gdzie Φ^{-1} oznacza funkcję odwrotną do dystrybuanty rozkładu normalnego o średniej 0 i odchyleniu standardowym 1. W tym celu skorzystamy z wzoru:

$$P(X \in (a, b]) = F_X(b) - F_X(a) \tag{3.5}$$

gdzie F_X to dystrybuanta rozkładu zmiennej X. Przekształcimy teraz dystrybuantę naszego rozkładu tak, aby zachować kształt rozkładu, ale zmienić wariancję i wartości oczekiwaną:

$$\begin{aligned} P(X \leqslant y) &= F(y) \\ P(\sigma X \leqslant \sigma y) &= F(y) \\ P(\sigma X + \mu \leqslant \sigma y + \mu) &= F(y) \\ P(\sigma X + \mu \leqslant \sigma y) &= F(y - \mu) \\ P(\sigma X + \mu \leqslant y) &= F\left(\frac{y-\mu}{\sigma}\right) \end{aligned} \tag{3.6}$$

Wykorzystując przekształcenie 3.6 możemy zapisać dystrybuantę $F_{p,n}$ naszego rozkładu dla ustalonego p i n, przybliżając ją za pomocą dystrybuanty rozkładu normalnego Φ:

$$F_{p,n}(y) \approx \Phi\left(\frac{\sqrt{n}(y-2p-1)}{2\sqrt{p(1-p)}}\right) \tag{3.7}$$

Dodatkowo oznaczmy jeszcze granice przedziału ufności:

$$L_n = -\frac{1}{\sqrt{n}}\Phi^{-1}(0.025) = \frac{1}{\sqrt{n}}\Phi^{-1}(0.975) \tag{3.8}$$

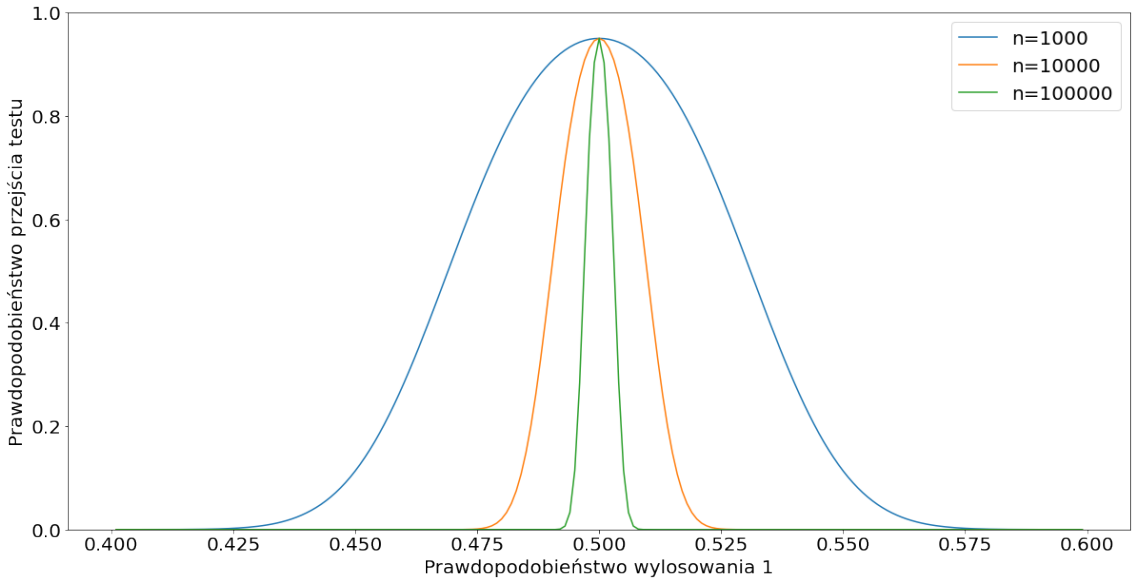
Korzystając zatem z 3.5 obliczamy szansę przejścia tego testu statystycznego dla dowolnego p:

$$f_n(p) = P\left(X \in [-L_n, L_n]\right) = F_{p,n}(L_n) - F_{p,n}(-L_n) \tag{3.9}$$

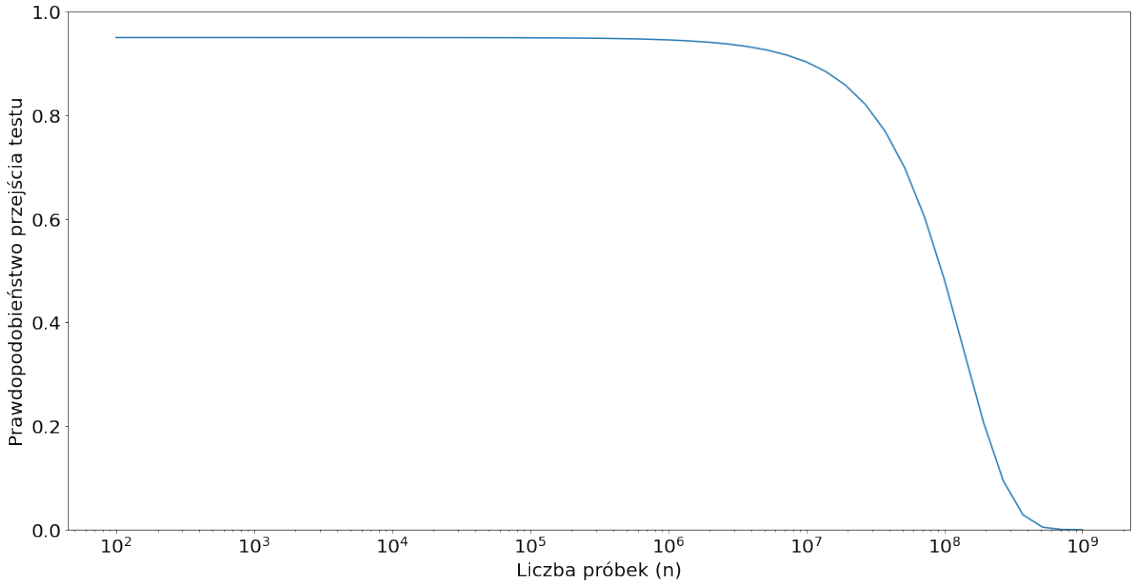
Po zaimplementowaniu tej funkcji, możemy zobaczyć jej wizualizację, przedstawioną na rysunku 3.1 oraz 3.2.

3.1.4 Wybranie akceptowalnych rozkładów

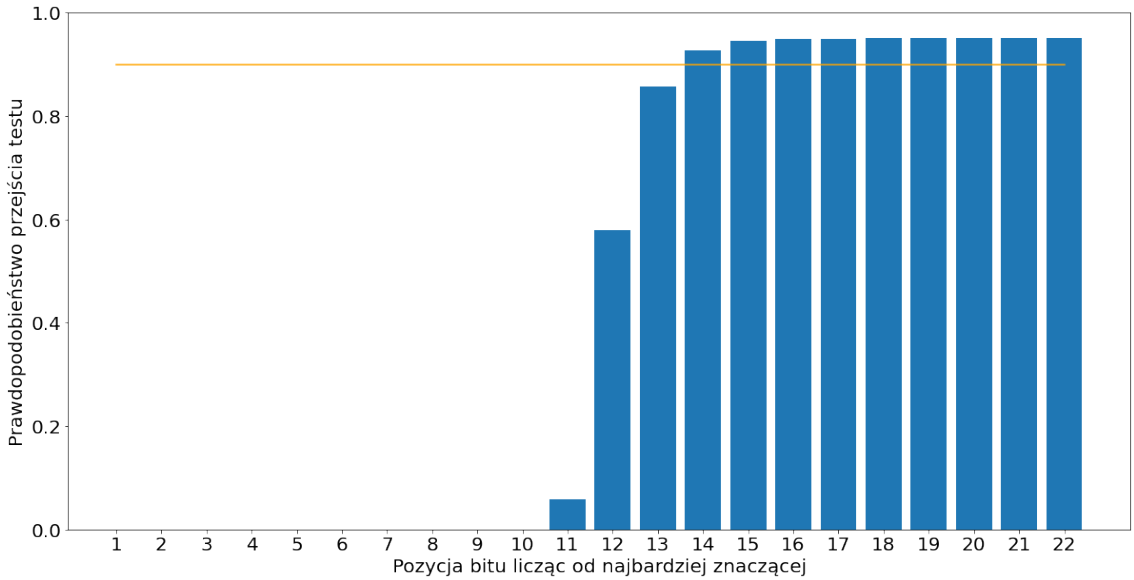
Zakładając prawdziwość tezy o rozkładzie spełniającym prawo Benforda sprawdzimy ile najbardziej znaczących bitów próbki powinniśmy odrzucić. Przyjmijmy za kryterium przejście testu powyżej w 90% przypadków dla średniej z 10^8 bitów ($n = 10^8$). Na wykresie 3.3 widzimy, iż bity będące na co najwyżej 14. najbardziej znaczącej pozycji spełniają nasze wymagania. Wcześniejsze niestety będziemy zmuszeni odrzucić ze względu na niekompatybilność rozkładu.



Rysunek 3.1: Porównanie prawdopodobieństwa przejścia testu dla różnych n



Rysunek 3.2: Porównanie prawdopodobieństwa przejścia testu dla różnych n (skala logarytmiczna) przy błędzie rozkładu 10^{-3}



Rysunek 3.3: Prawdopodobieństwo przejścia testu statystycznego dla k-tego najbardziej znaczącego bitu w próbce spełniającej prawo Benforda (niebieski) oraz próg akceptowalności na poziomie 0.9 (pomarańczowy)

3.2 Pseudokod generatora

Na podstawie konceptu naszego generatora oraz rozpoznanych potencjalnych problemów możemy przygotować jego pseudokod. Wykorzystamy w nim paradygmat programowania obiektowego, który jest obecnie dostępny w większości najpopularniejszych języków programowania. W pseudokodzie 3.1 wykorzystamy następujące typy:

SyncQueue - kolejka synchroniczna. Jest strukturą generyczną, a więc może przechowywać wartości dowolnego określonego typy. Posiada ona bufor oraz metodę *push* przyjmującą jeden argument, który zostanie dodany do kolejki. W przypadku zapełnienia bufora zatrzymuje wątek do czasu zwolnienia miejsca. Dodatkowo, posiada metodę *pop* która, jeśli bufor nie jest pusty, zdejmuje jeden element z kolejki i go zwraca, a w przypadku pustego bufora zatrzymuje wątek do czasu wprowadzenia do niego danych.

Thread - typ reprezentujący wątek w programie. W konstruktorze przyjmuje funkcję, którą będzie wykonywał po jego uruchomieniu. Wywołanie metody *start* nie zatrzymuje obecnego wątku, a jedynie tworzy nowy.

List - implementacja listy uporządkowanej.

Int - prosty typ liczbowy 64-bitowy.

Bit - typ liczbowy mogący przyjąć dwie wartości: 0 lub 1.

String - typ przechowujący łańcuch znaków.

Jedynym polem klasy jest *queue* typu *SyncQueue<Bit>*. Przechowuje ona wygenerowane do tej pory bity, które zostaną wykorzystane w metodzie *get*. Oprócz konstruktora, w którym możemy określić liczbę wykorzystywanych przez nasz generator wątków, mamy dwie metody:

produce - odpowiada ona za ciągłe pobieranie czasu dostępu do wskazanego w argumencie *url* adresu, a następnie umieszczanie pozyskanych w ten sposób użytecznych bitów w kolejce *queue*.

Pseudokod 3.1 Pseudokod klasy naszego generatora

```
class GENERATOR
    queue : SYNCQUEUE<BIT>                          ▷ kolejka synchroniczna bitów

    constructor(number_of_threads : INT)
        addresses:LIST<STRING> ←load()              ▷ wczytujemy adresy
        for i ← i to number_of_threads do
            thread : THREAD ←Thread(addresses[i])   ▷ tworzymy nowy wątek
            thread.start()                          ▷ uruchamiamy wątek produkujący bity
        end for
    end constructor

    procedure PRODUCE(url : STRING)
        loop
            time : INT ←ping(url)                   ▷ pobieramy czas w ns "pinga"
            bits : INT ←ceil(log2(time))            ▷ obliczamy długość liczby
            for _ ← 14 to bits do
                queue.push(time mod 2)              ▷ dodajemy ostatni bit do kolejki
                time ← time div 2                   ▷ dzielimy całkowicie przez 2
            end for
        end loop
    end procedure

    function GET
        result : INT ← 0
        for _ ← 0 to 63 do
            bit:BIT ← queue.pop()                   ▷ pobieramy bit z kolejki
            if bit == 1 then
                result ← result * 2 + 1             ▷ dodajemy 1 na koniec
            else
                result ← result * 2                 ▷ dodajemy 0 na koniec
            end if
        end for
        return result                               ▷ zwracamy 64-bitową liczbę
    end function
end class
```

get - umożliwia pobieranie liczby losowej z naszego generatora. Bierze ona z kolejki *queue* 64 bity i tworzy z nich pojedynczą liczbę losową spełniającą założenia tej pracy.

3.3 Wybór protokołu do mierzenia czasów połączeń

Ważnym z punktu widzenia wydajności systemu jest wybór odpowiedniego protokołu sieciowego, który wykorzystamy w naszym generatorze. Nie powinien mieć wpływu na losowość naszych liczb, stąd też możemy koncentrować się na wydajności rozwiązania i wygodzie jego użycia.

3.3.1 Przegląd dostępnych rozwiązań

Przedstawimy tutaj znalezione metody pozyskiwania czasów dostępu do zasobów sieciowych, a mianowicie:

ping *ICMP* (*Internet Control Message Protocol*) - najbardziej oczywiste podejście do problemu. Jest on obsługiwany przez zdecydowaną większość hostów i powoduje możliwie najmniejsze obciążenie, co pozwala przesłać jednocześnie większą liczbę zapytań z jego wykorzystaniem. Posiada on jednak podstawową wadę - wymaga dodatkowych uprawnień w celu jego wysłania na współczesnych systemach operacyjnych. Oznacza to konieczność uruchamiania programów z tym generatorem z uprawieniami administratora. Ponadto, zupełnie uniemożliwia jego wykorzystanie w niektórych przypadkach, takich jak front-end aplikacji webowych.

***HTTP* (*HyperText Transfer Protocol*)** - niezwykle popularny protokół. Wykorzystuje on *TCP*. Z uwagi na to, że wykorzystują go serwisy webowe znalezienie hostów, z którymi można się połączyć nie stanowi żadnego problemu. Ogromną zaletą tego protokołu jest znacząca liczba bibliotek, które go wspierają. Pozwalają one w większości języków pobrać zawartość strony w jednej linijce kodu. Problemem tego podejścia jest natomiast stosunkowo duży rozmiar pakietów, który zależy od wybranego hosta. Dla przykładu kod źródłowy strony *http://www.google.com* waży 128 KB, co nie jest relatywnie ogromną wartością, lecz w przypadku konieczności pobrania milionów takich stron mocno obciąża sieć.

ping *TCP* (*Transmission Control Protocol*) - inne dobre rozwiązanie umożliwiające mierznie czasu potrzebnego na nawiązanie sesji w protokole *TCP*. Naturalnie nie został on zaprojektowany do mierzenia opóźnień, stąd też nie jest tak lekki jak *ICMP*, a więc wywołuje większe obciążenie sieci. Zaletą tego rozwiązania jest brak konieczności posiadania specjalnych uprawnień, przez co możemy go wykorzystać w prawie każdej sytuacji. Wadą jest jednak fakt, iż pingowanie używając tej metody nie jest powszechne. Dlatego też nie zawsze znajdziemy proste w użyciu biblioteki to umożliwiające. Na szczęście, prawie każdy język ma wbudowane wsparcie dla *socketów*. Mając taką bibliotekę wystarczy napisać funkcję nawiązującą połączenie i natychmiast je zrywającą, zmierzyć potrzebny na to czas. Dodatkową wadą jest to, że ciągłe nawiązywanie i zrywanie połączenia nie jest naturalnym zachowaniem w sieci, stąd też po kilku takich próbach, następujących w krótkich odstępach, czas odpowiedzi hosta będzie mocno się wydłużał.

3.3.2 Wybór odpowiedniego protokołu

Jak widzimy, każda z 3 dostępnych opcji niesie z sobą kilka wad, jak i zalet. Na potrzeby tej pracy zastosujemy ostatnie rozwiązanie - tak zwany ping *TCP*. Generator do testów napiszemy w języku Python. Posiada on wsparcie dla tworzenia *socketów*. W przypadku jego implementacji na front-endzie aplikacji webowej, musielibyśmy skorzystać z *web-socketów*. Konieczność nadawania uprawnień administratora dla każdego uruchomienia programu jest w zbyt dużym utrudnieniem dla użytkownika aby zdecydować się na ten wariant z protokołem *ICMP*. Zmiana wybranego protokołu w już zaimplementowanym generatorze nie jest trudna ze względu na jego prostą strukturę. Bez problemu można użyć bibliotek takich jak: *pyping* (dla pingu *ICMP*), *requests* (dla pobierania zawartości stron) oraz *tcping* (dla pinga w oparciu o protokół *TCP*).

3.4 Implementacja w języku Python

Poniżej znajdziemy kod źródłowy 3.1 naszego generatora. Ze względu na jego stosunkowo małą długość pozwolimy sobie omówić go w całości.

Kod źródłowy 3.1: Klasa generatora w języku Python

```
1  import math
2  import time
3  import socket
4  import queue
5  import random
6  import threading
7
8  TIMER_RESOLUTION_IN_NS = 100  # rozdzielczość czasu
9  MOST_SIGNIFICANT_BITS_TO_REJECT = 13  # liczba bitów do odrzucenia
10
11
12 def ping(address):
13     """Procedura nawiązująca i zrywająca połączenie"""
14     s = socket.socket()
15     s.connect((address, 80))
16     s.shutdown(socket.SHUT_RD)
17
18
19 def load_file(file_name):
20     """Funkcja wczytująca dane z pliku i zapisująca je do listy"""
21     res = []
22     f = open(file_name, "r")
23     for x in f:
24         res.append(x[0:-1])
25     f.close()
26     return res
27
28
29 def elapsed(fn):
30     """Funkcja zwracająca czas potrzebny na wywołanie funkcji fn"""
31     start = time.perf_counter_ns()
32     fn()
33     stop = time.perf_counter_ns()
34     return (stop - start) // TIMER_RESOLUTION_IN_NS
35
36
37 class Pinger(threading.Thread):
38     """Klasa wątku odpytującego kolejne adressy z listy"""
39
40     def __init__(self, output, addresses):
41         threading.Thread.__init__(self)
42         self._output = output
43         self._addresses = addresses
44         self._index = random.randint(0, len(addresses))
45         self._stop_event = threading.Event()
46
47     def run(self):
48         """Procedura generująca bity"""
49         while not self._stop_event.is_set():
50             try:
51                 eclipsed = elapsed(lambda: ping(self._addresses[self._index]))
52                 length = math.ceil(math.log2(eclipsed))
53                 for _ in range(length - MOST_SIGNIFICANT_BITS_TO_REJECT):
54                     self._output.put(eclipsed % 2)
55                     eclipsed //= 2
56             except:
57                 print("Connection error with " + self._addresses[self._index])
```

```
58              finally:
59                  self._index = (self._index + 1) % len(self._addresses)
60
61      def stop(self):
62          """Procedura zatrzymująca wątek"""
63          self._stop_event.set()
64
65
66  class Generator:
67      """Klasa generatora przyjmująca liczbę wątków"""
68
69      def __init__(self, threads):
70          self._queue = queue.Queue()
71          addresses = load_file("addresses")
72          self._threads = []
73          for _ in range(threads):
74              thread = Pinger(self._queue, addresses)
75              thread.start()
76              self._threads.append(thread)
77
78      def get(self):
79          """Funkcja zwracająca 64-bitową liczbę losową"""
80          result = 0
81          for _ in range(64):
82              bit = self._queue.get()
83              if bit == 0:
84                  result = result * 2
85              else:
86                  result = result * 2 + 1
87          return result
88
89      def stop(self):
90          """Funkcja zatrzymująca generator"""
91          for thread in self._threads:
92              thread.stop()
```

Warto zwrócić uwagę na dwie użyte stałe:

TIMER_RESOLUTION_IN_NS - stałą użyta w linijce 8. Określa ona rozdzielczość czasu dla funkcji *perf_counter_ns*, a więc najmniejsza zmiana, która powoduje zwrócenie innego wyniku. Wpływa ona bezpośrednio na ilość bitów informacji możliwych do uzyskania z pojedynczej próbki.

MOST_SIGNIFICANT_BITS_TO_REJECT - stała zastosowana w lilijce 9. Oznacza ona liczbę najbardziej znaczących bitów z każdej próbki, którą będziemy odrzucać ze względu na występowanie prawa Benforda.

Dodatkowo napisaliśmy funkcje pomocnicze:

ping - w linijce 12. Tworzy ona domyślny *socket*, a następnie łączy się z hostem z argumentu *address* na porcie 80 używanym przez protokół *HTTP*. Po ustanowieniu połączenia go zrywa.

load_file - w linijce 19. Wczytuje ona kolejne linie z pliku o nazwie pobranej z argumentu *file_name* i zapisuje je w liście, która jest zwracana na końcu.

elapsed - w linijce 29. Mierzy ona czas potrzebny na wykonanie funkcji przekazanej w argumencie *fn*, a następnie zwraca wynik w jednostce zapewniającej maksymalną ilość informacji o upłyniętym czasie.

Kod zawiera dwie główne klasy:

Pinger - którą możemy znaleźć w linijce 37. rozszerzającą klasę wątku. Jej głównym zadaniem jest kolejne odpytywanie adresów z argumentu *addresses*, a następne wysyłanie bitów o odpowiednim rozkładzie za pomocą kanału *output*. Dodatkowo w linijce 61. znajduje się procedura *stop*, która pozwala zatrzymać wykonywanie wątku.

Generator - znajduje się w linijce 66. Jej zadaniem jest generowanie liczb 64-bitowych na podstawie czasów dostępu do zasobów sieciowych. W konstruktorze przyjmuje ona jeden parametr *threads*, który określa ile wątków będzie wykorzystywał generator do pozyskiwania losowych bitów z czasów dostępu do zasobów sieciowych. Dodatkowo konstruktor wczytuje zawartość pliku zawierającego adresy, do których można się połączyć. Funkcja *get*, którą znajdziemy w linijce 78. pobiera bity z kolejki *_queue*, a następnie tworzy z nich jedną liczbę losową. Klasa zawiera również metodę *stop* w linijce 89., której zadaniem jest zakończenie pracy generatora oraz wykorzystywanych przez niego wątków.

3.5 Ocena wydajności generatora

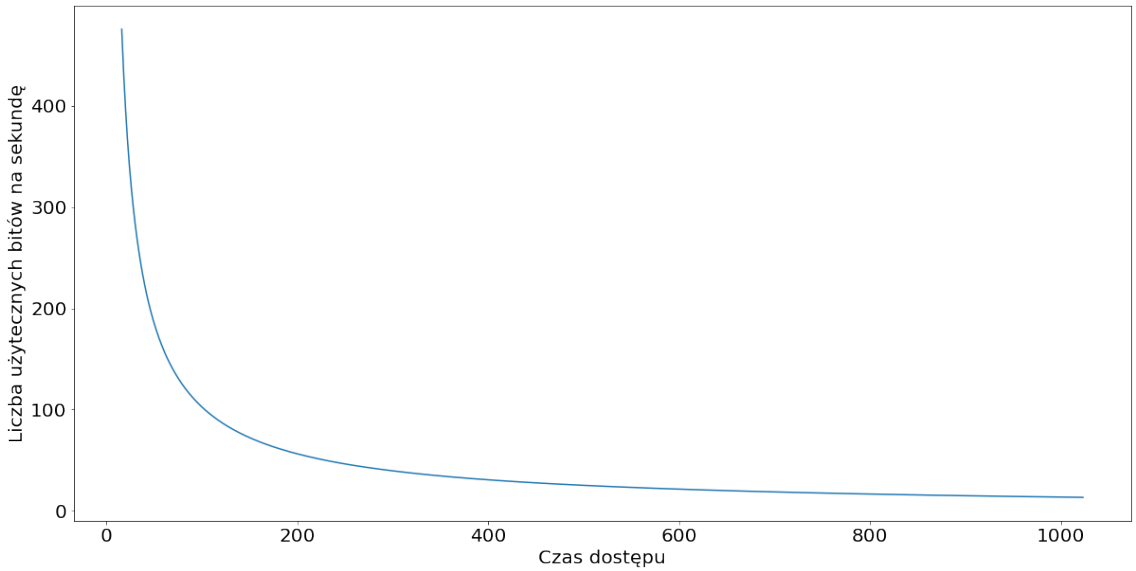
Po zaimplementowaniu generatora możemy dokonać analizy jego wydajności, a więc określić ile losowych bitów będzie w stanie wyprodukować w ciągu sekundy.

3.5.1 Ocena wydajności dla pojedynczego wątku

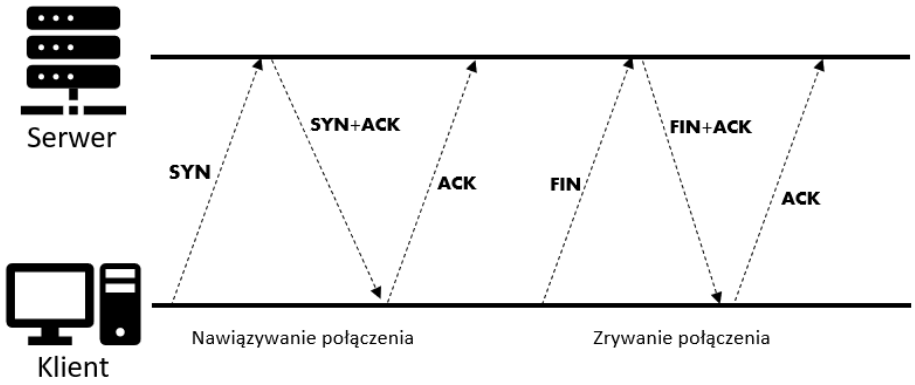
Jak możemy zobaczyć w kodzie w linijce 29. metoda *elapsed* wykorzystuje metodę *time.pref_counter_ns*. Według dokumentacji [3] dla systemu Windows zapewnia ona rozdzielczość czasu na poziomie 100 nanosekund. Oznacza to, że z każdymi upłyniętymi 100 ns funkcja *time.pref_counter_ns* zwróci inny wynik. Dzieląc go przez 100 możemy uzyskać liczbę składającą się z maksymalnej ilości użytecznych bitów. Każdy wątek czeka do czasu otrzymania odpowiedzi z serwera, zatem ilość wykonanych pingów w ciągu sekundy zależy od czasu oczekiwania na odpowiedź. Z drugiej strony im dłuższy czas tym więcej użytecznych bitów otrzymujemy. Zdecydowaliśmy się odrzucać 13 pierwszych bitów ze względu na ich rozkład, a więc próbki poniżej 800 μs nie niosą żadnej informacji. Na rysunku 3.4 widzimy wydajność pojedynczego wątku dla różnych czasów pinga. W praktyce nie udaje się uzyskać czasów lepszych niż 16ms, jeśli chcemy połączyć się do urządzenia spoza naszej sieli lokalnej. W takim przypadku z pojedynczego wątku możemy uzyskać nawet 60 B/s. Jest to relatywnie dobry wynik, choć wypada niezwykle blado w porównaniu z generatorami liczb pseudolosowych. W praktyce jednak średni czas dla pojedynczego pigna w testach wynosił ok. 121 ms. W tym przypadku możemy spodziewać się wydajności rzędu 11 B/s, czyli ok. trzy 64-bitowe liczby losowe na sekundę.

3.5.2 Ograniczenie przez przepustowość sieci

Aby oszacować jakie obciążenie sieci spowoduje wysłanie jednego pinga *TCP* musimy najpierw zrozumieć jego podstawy. *TCP* jest protokołem połączeniowym, zatem pojedynczy ping wymaga nawiązania połączenia, a następne jego zerwania. Jak widzimy na rysunku 3.5 ta operacja wymaga wysłania 4 pakietów, przy czym dodatkowo powinniśmy otrzymać 2 pakiety od serwera. Każdy z nich waży od 60 do 80 bajtów. Wykorzystując górne oszacowanie widzimy, że pojedynczy ping wymaga wysłania 320 bajtów oraz pobrania 160 bajtów. Większość dostawców internetu sprzedaje pakiety, których przepustowość wysyłania jest mniejsza od przepustowości pobierania, zatem skupimy się na limicie przepustowości wysyłania. Zakładając, że mamy do dyspozycji łącze o przepustowości wysyłania 1 Mbps, teoretycznie powinniśmy mieć możliwość wykonania 390 pingów na sekundę. Dodatkowym czynnikiem, jaki należy wziąć pod uwagę, jest koszt wykorzystania usługi *DNS* (*domain name system*). Tłumaczy ona adresy domenowe takie



Rysunek 3.4: Liczba akceptowalnych bitów uzyskiwanych na jedną milisekundę (oś y), przy stałym czasie pinga w milisekundach (oś x)

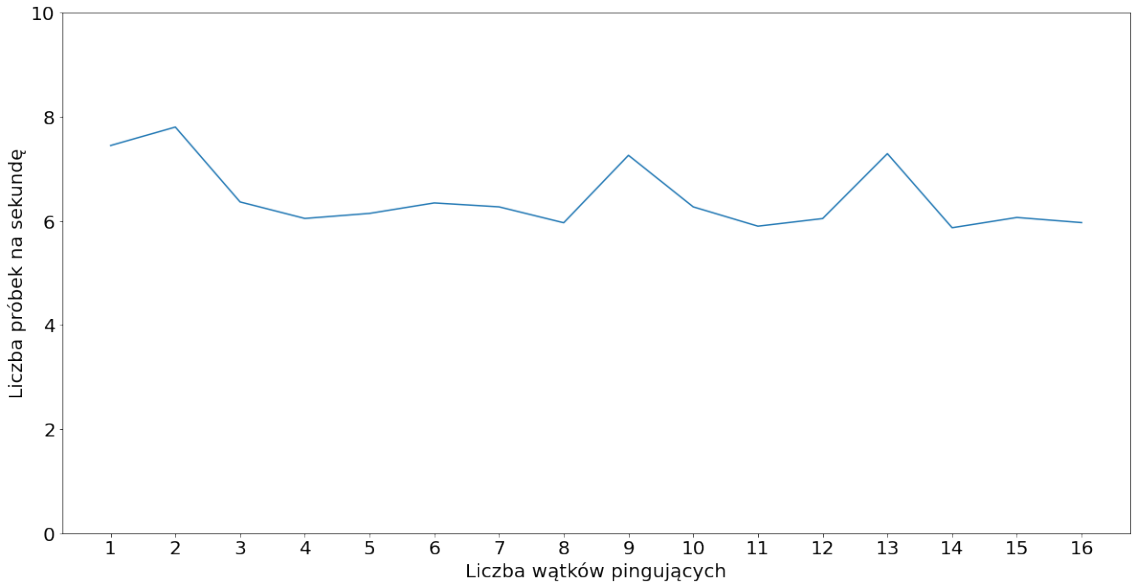


Rysunek 3.5: Diagram wysyłanych pakietów przy nawiązywaniu i zrywaniu połączenia w protokole TCP

jak *www.google.pl* na adresy IP. Jeśli chcemy wykorzystywać adresy w formie domenowej, będziemy umieli do każdego pinga doliczyć ok. 120 bajtów wysłanych oraz 120 bajtów pobranych. Zatem tym samym łączem o przepustowości wysyłania 1 Mbps będziemy teoretycznie mogli wykonać 284 pingów na sekundę.

3.5.3 Eksperymentalne wyniki w środowisku testowym

Znając teoretyczne ograniczenia wydajności naszego generatora warto sprawdzić jak w stosunku do teoretycznych ograniczeń wypada jego wydajność w środowisku testowym. Testy zostały wykonane na prywatnym komputerze w niedzielę o godzinie 18:00. Według serwisu *https://www.speedtest.net* wykorzystane połączenie miało 83.75 Mbps przepustowości pobierania oraz 24.29 Mbps przepustowości wysyłania. Zbadaliśmy liczbę pingów na sekundę przy różnej liczbie jednoczesnych pingów. Jak możemy zobaczyć na rysunku 3.6, nie została wykazana



Rysunek 3.6: Wykres zestawiający liczbę jednoczesnych wątków (oś x) z liczbą pingów na sekundę (oś y)

żadna istotna korelacja między liczbą wątków a ogólną wydajnością generatora. Spore wahania w wydajności dla różnej liczby wątków zostały prawdopodobnie spowodowane różnicami w obciążeniu sieci w momencie testu. Brak zwiększonej wydajności w przypadku generatora z większą liczbą wątków jest najprawdopodobniej spowodowany limitem ilości przesyłanych pakietów przez lokalny router lub dostawcę internetu. Dalsze badania są wymagane w celu jednoznacznego ustalenia przyczyny problemów z straconymi pakietami w przypadku testów z dużą liczbą jednoczesnych pingów.

Testy losowości

Rozdział ten poświęcony będzie testowaniu losowości wyników w celu ustalenia, czy liczby losowe otrzymane z naszego generatora spełniają warunek jednostajności oraz niezależności. Sprawdzimy, jak wypada on w stosunku do znanych rozwiązań. Opiszemy również teorię dotyczącą testowania losowości ciągów bitowych.

4.1 Jak udowodnić losowość?

Krótka odpowiedź brzmi na zadane w tytule niniejszego podrozdziału pytanie mówi, iż nie jest to możliwe. Aby móc powiedzieć z pewnością, jakimi parametrami cechuje się testowany generator, musielibyśmy poznać nieskończoność wygenerowanych przez niego wyników. Załóżmy, że wygenerował on do tej pory ciąg N jedynek. Moglibyśmy rzucić zatem tezę, iż nie spełnia on założenia jednostajności. Wiemy że:

$$\begin{gathered} P(X = \mathbb{1}^N) = 2^{-N} \\ \lim_{N\to\infty} P(X = \mathbb{1}^N) = \lim_{N\to\infty} 2^{-N} = 0 \end{gathered} \tag{4.1}$$

Istnieje zatem szansa, że prawidłowy generator wyprodukuje taki ciąg. Żyjemy w skończonym świecie, a więc empiryczne sprawdzenie granicy dla N dążącego do nieskończoności jest niemożliwe. Co za tym idzie, nie możemy matematycznie udowodnić, iż nie spełnia założeń.

4.1.1 Redukcja oczekiwań

Jesteśmy zmuszeni zatem zredukować nasze oczekiwania odnośnie testowania generatorów, z matematycznego dowodu, do statystycznego prawdopodobieństwa. Nawiązując do poprzedniego przypadku wiemy, że uzyskanie wektora samych jedynek było zawsze możliwe, jednak szansa na to jest bardzo mała. Uzyskanie dowolnego n-elementowego wektora binarnego jest jednakowo prawdopodobne przy założeniu jednostajności i niezależności. Musimy zatem znaleźć jakąś funkcję mierzalną (w naszych zastosowaniach: w sensie Lebesqua), która nie będzie różnowartościowa i zmapuje wiele podobnych wektorów na tę samą wartość. Dobrym przykładem będzie tutaj np. stosunek zer do wszystkich cyfr. Jak nietrudno policzyć, dla generatora spełniającego założenie jednostajności, stosunek ten powinien wynosić $\frac{1}{2}$. Istnieją również tylko dwa najbardziej oddalone od tej wartości wektory: $\mathbb{1}^N$ i $\mathbb{0}^N$. Wiemy więc, że prawidłowy generator ma prawdopodobieństwo wylosowania tak skrajnie rzadko występującego przypadku równe $\frac{1}{2^{n-1}}$.

4.2 Teoria testów statystycznych

Test statystyczny jest to narzędzie pozwalające ocenić prawdopodobieństwo hipotezy statystycznej na podstawie skończonej próbki z pewnej populacji.

4.2.1 Hipotezy statystyczne

Każdy test zaczynamy od sformułowania hipotez:

hipoteza zerowa (H_0) - podstawowa hipoteza mówiąca o tym, że próbka nie odbiega w żaden statystycznie istotny sposób od przyjętych założeń. W naszym przypadku oznacza to, że próbka została wygenerowana przez generator spełniający założenie jednostajności lub niezależności, albo oba.

hipoteza alternatywna (H_1) – zaprzeczenie hipotezy zerowej. W naszym przypadku hipoteza alternatywna będzie mówiła, że nasza próbka różni się w statystycznie istotny sposób od takich, które wygenerowałby generator spełniający założenie jednostajności lub niezależności, albo oba.

4.2.2 Podstawowe definicje

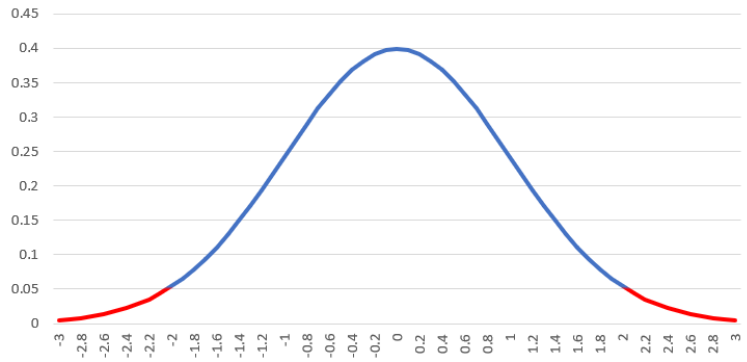
Wprowadzimy kilka podstawowych pojęć i symboli:

alfa (α) - wartość określająca, jaka jest szansa, że generator spełniający hipotezę zerową nie przejdzie testu. Z reguły ustalona na poziomie 5%.

beta (β) – wartość określająca szansę, że generator spełniający hipotezę alternatywą uzyska pozytywny wynik testu.

przedział ufności – przedział w którym, zgodnie z rozkładem teoretycznym powinno znaleźć się $1-\alpha$ próbek. Tak jak na rysunku 4.1 powinien on pokrywać najbardziej prawdopodobną część teoretycznego rozkładu.

W przypadku testów statystycznych bardzo często dokonuje się uproszczenia modelu wykorzystując centralne twierdzenie graniczne zakładając, że populacja ma rozkład normalny. Ułatwia to znacząco wyznaczenie zakresów dla przedziałów ufności.



Rysunek 4.1: Przedziały ufności(niebieski) dla testu statystycznego z cechą populacji mającą rozkład normalny z $\alpha = 0.05$

4.2.3 Rodzaje błędów

Przyswoiwszy znaczenie parametrów α i β możemy powiedzieć kilka słów o rodzajach błędów, które możemy popełnić w naszym teście.

Jak widzimy w tablicy 4.1, wyróżniamy błąd pierwszego rodzaju (jeśli błędnie odrzucimy hipotezę zerową). Oraz błąd drugiego rodzaju (jeśli przymniemy hipotezę zerową pomimo iż byłą ona fałszywa). Z tablicy 4.2 możemy wyczytać iż α określa prawdopodobieństwo popełnienia błędu pierwszego rodzaju, a β prawdopodobieństwo popełnia błędu drugiego rodzaju. O ile z łatwością możemy kontrolować α (wybierając inny przedział ufności, co możemy zobaczyć na rysunku 4.1), to kontrolowanie, a nawet wyliczenie β jest w wielu przypadkach niemożliwe. Generatorów produkujących ciągi bitowe jest przynajmniej continuum. Dla każdej liczby rzeczywistej $p \in [0, 1]$ istnieje taki generator, że $(\forall i \in \mathbb{N})P(X_i = 1) = p$. Załóżmy, iż mamy do

		Rzeczywistość	
		H_0	H_1
Wynik	H_0	Ok	Błąd II rodzaju
	H_1	Błąd I rodzaju	Ok

Tablica 4.1: Przedstawienie rodzajów błędów w teście statystycznym

		Rzeczywistość	
		H_0	H_1
Wynik	H_0	1 - α	β
	H_1	α	1 - β

Tablica 4.2: Przedstawienie prawdopodobieństwa popełnienia błędu w teście statystycznym.

czynienia z niezależnym generatorem o jakimś p oraz, że p zostało wybrane z rozkładem ciągłym. W takiej sytuacji, szansa wybrania $p = \frac{1}{2}$ wynosi dokładnie 0, a więc $\beta = 1$. Takie rozumowanie prowadzi jednak do poważenia zasadności testów statystycznych, przez co nie wyliczamy β, a jedynie skupiamy się na statystycznie istotnych różnicach. Możemy jednak stwierdzić, że dla coraz większej α zmniejsza się β. W skrajnym przypadku, gdzie α wynosi 1, β będzie równa 0. Nie jest to jednak najbardziej użyteczny test, gdyż zawsze odrzuca on hipotezę zerową. Co więcej, o czy, nie należy zapominać, β będzie zmniejszać się wraz z zwiększaniem się rozmiaru naszej próbki.

4.2.4 Wartość p

Kluczowym pojęciem w testach statystycznych jest wartość p (P_{value}). Określa ona, w jakim miejscu teoretycznego rozkładu przy założeniu hipotezy zerowej znajduje się nasza próbka. Im większa wartość p, tym większe prawdopodobieństwo, iż została ona wzięta z populacji spełniającej hipotezę zerową, a w naszym przypadku: iż została wygenerowana przez jednostajny i niezależny generator. Obliczenie prawdopodobieństwa tego zdarzenia jak wiemy, jest jednak niezwykle trudne. Otrzymanie skrajnej wartości p nie oznacza, że hipoteza zerowa jest fałszywa; oznacza jedynie, że jest mało prawdopodobna. Co więcej, przy założeniu prawdziwości hipotezy zerowej wartość oczekiwana wartości p to $\frac{1}{2}$, gdyż dla $P_{\text{value}}(X) \sim \mathcal{U}(0,1)$, gdzie $\mathcal{U}(0,1)$ to rozkład jednostajny na odcinku $(0,1)$.

4.3 Narzędzia testowania losowości

Problem określania czy dany wektor bitów jest losowy czy nie, jest relatywnie starym problemem. Od czasów wykorzystywania liczb losowych w kryptografii pojawiło się wiele narzędzi do testowania oraz paczek testów.

4.3.1 Przegląd paczek testów losowości

Ent - łatwa w użyciu paczka 5 testów losowości stworzona przez Johna Walkera w 1998 roku. Jest dostępna w dystrybucji Ubuntu. Zawiera dojść prymitywne testy, z których nie każdy posiada wyliczaną wartość p. Za pomocą tego narzędzia możemy przetestować wektor losowy dowolnej długości.

Diehard - pierwotnie wydany na płycie CD w 1995 roku przez George'a Marsaglię zestaw 12 testów na losowość stanowiący, jak na tamte czasy najlepszy tego typu pakiet. Jego użycie

jest również proste. Nie zawiera on jednak satysfakcjonującego uzasadnienia matematycznego wykonywanych obliczeń oraz teoretycznego rozkładu, na podstawie którego wyliczane są wartości p. Narzędzie możemy znaleźć pod adressem [14].

NIST test suite - zestaw 15 testów przygotowanych przez *National Institute of Standards and Technology* w celu sprawdzenia generatorów liczb pseudolosowych do zastosowań kryptograficznych. Używane są jako standard dla kryptografii. Każdy z testów został opisany wa pracy *A statistical test suite for random and pseudorandom number generatorsfor cryptographic applications* [4]. Zawarte w niej opisy testów nie zawierają sposobu wyliczenia teoretycznego rozkładu. Narzędzie to możemy znaleźć pod adresem [2].

TestU01 - najnowsza wersja tej paczki testów pochodzi z 2009 roku, została stworzona przez Pierre L'Ecuyera. Jest ona również powszechnie używana, jednak została przygotowana do testowania generatorów liczb pseudolosowych i wymaga wbudowania swojego generatora w napisany języku w C framework testujący. Narzędzie możemy znaleźć pod adresem [12].

Dieharder - narzędzie do testowania losowości stworzone przez Roberta G. Browna. Jego najnowsza wersją powstała w 2020 roku. Paczka zawiera testy z narzędzia *dierhard*, jak również większość przygotowanych przez *NIST* oraz autorskie testy. Jest narzędzie to jest dostępne w dystrybucji Ubuntu oraz pod adresem [6]. Większość z nich wymaga dużej ilości danych powyżej 100MB.

4.3.2 Wybór generatorów do porównania wyników

Nasze wyniki porównamy z trzema innymi znanymi generatorami różnej klasy. Przyjrzymy się jak wypada on na tle następujących rozwiązań:

LCG (Linear congruential generator) - jedno z najstarszych i najbardziej znanych podejść do problemu generowania liczb losowych. Logika stojąca za tym generatorem jest łatwa do zrozumienia i sprowadza się do jednego wzoru:

$$X_{n+1} = (aX_n + b) \bmod m \tag{4.2}$$

Pomimo swojej prostoty daje on zadowalające wyniki, stąd też jest używany do dziś między innymi w bibliotece *rand* języka C. Do testów użyjemy generatora o parametrach: $m = 2^{32}$, $a = 22695477$ oraz $b = 1$.

Tarus - rozwiązanie bardzo zbliżone do poprzedniego stworzone przez Pierre L'Ecuyer'a. Posiada bardziej rozbudowany stan wewnętrzny składający się z trzech zmiennych: s^1, s^2 oraz s^3. Zasadę działania generatora opisuje wzór:

$$\begin{aligned} X_n &= (s_n^1 \oplus s_n^2 \oplus s_n^3) \\ s_{n+1}^1 &= ((s_n^1 \,\&\, 4294967294) \ll 12) \oplus (((s_n^1 \ll 13) \oplus s_n^1) \gg 19) \\ s_{n+1}^2 &= ((s_n^2 \,\&\, 4294967288) \ll 4) \oplus (((s_n^2 \ll 2) \oplus s_n^2) \gg 25) \\ s_{n+1}^3 &= ((s_n^3 \,\&\, 4294967280) \ll 17) \oplus (((s_n^3 \ll 3) \oplus s_n^3) \gg 11) \end{aligned} \tag{4.3}$$

gdzie & oznacza operator iloczynu bitowego (*and*), $\oplus$ oznacza bitową sumę rozłączną (*xor*), a $\ll$ oraz $\gg$ oznaczają operację przesunięcia bitowego, odpowiednio w lewo i prawo.

CRNG (Cryptographic random number generator) - generator liczb losowych do zastosowań kryptograficznych. W tej pracy używamy rozwiązania z pakietu *secrets* z języka Python. Wykorzystuje on zebraną przez system operacyjny entropię w celu wygenerowania liczb "prawdziwie" losowych.

4.4 Testy losowości

Ze względu na niezwykle ubogie matematyczne uzasadnienie poprawności testów w znalezionych gotowych narzędziach testujących, jak również ich przystosowanie do testowania generatorów liczb pseudolosowych, wykorzystamy w tej pracy własne testy. Będą on mocno inspirowane pracą *NITS* [4] wzbogacone o odpowiednie uzasadnienie matematyczne.

4.4.1 Test częstotliwości

Cel testu

Ten test skupia się na sprawdzeniu, czy wektor bitów X ma rozkład jednostajny, a więc czy stosunek zer do jedynek dąży do równej proporcji. Jest to jeden z prostszych testów. Polega on na wyliczeniu wspominanej proporcji, a następnie obliczeniu szansy jej otrzymania przy założeniu jednostajności generatora.

Matematyczne podstawy testu

Pierwszym krokiem będzie różnowartościowe zmapowanie wektora X na zbiór $\{-1, 1\}$, tworząc zmienną losową $Y_i = 2X_i - 1$. Jest ona specjalnym przypadkiem rozkładu 3.2, z p równym $\frac{1}{2}$. Korzystając z centralnego twierdzenia granicznego możemy użyć aproksymacji 3.4 w postaci:

$$\mathcal{N}\left(0, \frac{1}{\sqrt{n}}\right) \tag{4.4}$$

Oznaczmy sumę po wszystkich wartościach Y_i jako S:

$$S = \sum_{i=1}^{n}(2X_i - 1) = \sum_{i=1}^{n} Y_i \tag{4.5}$$

Oznaczmy dodatkowo Φ jako dystrybuantę standardowego rozkładu normalnego:

$$\Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}} dt = \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right) \tag{4.6}$$

gdzie, *erf* jest funkcją błędu Gaussa, zdefiniowaną jako:

$$\operatorname{erf}(z) = \frac{1}{\sqrt{\pi}} \int_{0}^{z} e^{-t^2} dt \tag{4.7}$$

Z centralnego twierdzenia granicznego wiemy, że w granicy, przy $n \to \infty$:

$$\begin{aligned} \frac{S}{n} &\sim \mathcal{N}\left(0, \frac{1}{\sqrt{n}}\right) \\ \frac{S\sqrt{n}}{n} &\sim \mathcal{N}(0, 1) \\ \frac{S}{\sqrt{n}} &\sim \mathcal{N}(0, 1) \end{aligned} \tag{4.8}$$

Możemy zatem, przy $n \to \infty$, skorzystać z dystrybuanty rozkładu normalnego jako dystrybuanty $\frac{S}{\sqrt{n}}$:

$$\Phi(x) = P(\frac{S}{\sqrt{n}} \leqslant x) = \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2n}}\right)\right) \tag{4.9}$$

Standardowy rozkład normalny jest symetryczny ($\Phi(x) = 1 - \Phi(-x)$). Możemy uprościć nieco formułę używając wartości bezwzględnej:

$$P\left(\frac{|S|}{\sqrt{n}} \geqslant x\right) = P\left(\frac{S}{\sqrt{n}} \geqslant x\right) + P\left(\frac{S}{\sqrt{n}} \leqslant -x\right) = 1 - P\left(\frac{S}{\sqrt{n}} \leqslant x\right) + P\left(\frac{S}{\sqrt{n}} \leqslant -x\right) = \\ = 1 - \Phi(x) + \Phi(-x) = 1 - \Phi(x) + (1 - \Phi(x)) = 2 - 2\Phi(x) = 2\Phi(-x) \tag{4.10}$$

Znamy zatem przybliżony teoretyczny rozkład zmiennej $\frac{|S|}{\sqrt{n}}$.

Wartość p

Dla wektora $X_{i\in\{1,\ldots,n\}}$, który przyjmuje wartości 0 lub 1 wartość p wynosi:

$$P_{\text{value}} = 1 - \text{erf}\left(\frac{|\sum_{i=1}^{n} X_i|}{\sqrt{2n}}\right) \tag{4.11}$$

Przyjmując standardowe $\alpha = 0.05$ jeśli $P_{\text{value}} > 0.05$ test uznajemy za zaliczony.

4.4.2 Test jedynek w bloku

Cel testu

Test skupia się na ilości występowania jedynek w M-bitowych blokach. Zlicza, ile jest bloków o k jedynkach, a następnie ocenia, jak bardzo uzyskany wynik pokrywa się z teoretycznym rozkładem.

Matematyczne podstawy testu

Mając n bitów, możemy podzielić je na $N = \lfloor \frac{n}{M} \rfloor$ nienachodzących się na siebie bloków. Każdy z nich może mieć od 0 do M jedynek. Możemy zatem zastosować test χ^2 Petersona. Oznaczmy:

$$\chi^2 = \sum_{i=0}^{M} \frac{(O_i - E_i)^2}{E_i} \tag{4.12}$$

gdzie:

O_i - liczba bloków z i jedynkami,

E_i - oczekiwana liczba bloków z i jedynkami, która wynosi $N\frac{1}{2^M}\binom{M}{i}$.

Dowód testu Petersona o najlepszym dopasowaniu pominiemy w tej pracy. Zainteresowani mogą go znaleźć jednak na stronie *MIT* [18]. Przedstawimy jedynie jego zarys, żeby zrozumieć zasadę działania tego testu. Krokiem pierwszym w dowodzie byłoby pokazanie, iż zgodnie z centralnym twierdzeniem granicznym dla każdego i $\frac{(O_i - E_i)^2}{E_i}$ dąży do rozkładu normalnego $\mathbb{Z}_i$. Następny etap to wykazanie, że dowolnie wybrany zbiór $\mathbb{Z}_i$ mocy M będzie niezależny, przy czym ostatnia zmienna będzie oczywiście funkcją pozostałych. Możemy zatem przybliżyć rozkład naszego χ^2 za pomocą rozkładu χ^2 o M stopniach swobody. Dystrybuantę tego rozkładu oznaczmy literą Ψ:

$$\Psi_k(x) = \frac{1}{\Gamma(1/k)}\gamma\left(\frac{k}{2}, \frac{x}{2}\right) \tag{4.13}$$

gdzie:

$$\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}dt \\ \gamma(z,x) = \int_0^x t^{z-1}e^{-t}dt \tag{4.14}$$

Wartość p

Test rozpoczynamy od wyznaczenia wartości O_i dla każdego i, a następnie obliczamy wartość p:

$$P_{\text{value}} = 1 - \Psi_M \left(\sum_{i=0}^{M} \frac{\left(O_i - \frac{N}{2^M} \binom{M}{i} \right)^2}{\frac{M}{2^N} \binom{M}{i}} \right) \tag{4.15}$$

Za pozytywy wynik uznajemy $P_{\text{value}} > 0.05$.

4.4.3 Test nieprzerwanych ciągów

Cel testu

Skupia się on na liczbie występujących w wektorze, nieprzerwanych ciągach tych samych bitów. Każdy nieprzerwany ciąg zer jest poprzedzony jedynką, nie zawiera żadnej jedynki, a następnym bitem po nim jest jedynka (analogicznie dla ciągu jedynek). Porównamy następnie nasz wynik do teoretycznego rozkładu liczby nieprzerwanych ciągów.

Matematyczne podstawy testu

Liczba ciągów zależy bezpośrednio od liczby jedynek (n_1) i liczby zer (n_0). Przyjętymi przez nas oznaczeniem n będzie $(n_0 + n_1)$. Możemy stosunkowo łatwo wyznaczyć rozkład zmiennej losowej (R) liczby nieprzerwanych ciągów w n-bitowym wektorze:

$$\begin{aligned} P(R = 2x) &= \frac{2\binom{n_0-1}{x-1}\binom{n_1-1}{x-1}}{\binom{n_0+n_1}{n_1}} \\ P(R = 2x+1) &= \frac{\binom{n_0-1}{x}\binom{n_1-1}{x-1} + \binom{n_0-1}{x-1}\binom{n_1-1}{x}}{\binom{n_0+n_1}{n_1}} \end{aligned} \tag{4.16}$$

Pierwszym spostrzeżeniem pozwalającym na sporządzenie wzorów 4.16 jest fakt, iż każdy ciąg nieprzerwanych jedynek jest przedzielony ciągiem zer, a więc liczba ciągów jedynek nie może się różnić od liczby ciągów zer o więcej niż 1. Tak więc w przypadku parzystej liczby nieprzerwanych ciągów mamy tyle samo nieprzerwanych ciągów zer, jak i jedynek. Przejdźmy zatem do obliczenia, ile wektorów ma dokładnie x nieprzerwanych ciągów. W przypadku parzystej liczby nieprzerwanych ciągów wiemy, że mamy $\frac{x}{2}$ nieprzerwanych ciągów jedynek. Obliczmy więc na ile sposobów możemy podzielić n_1 bitów na $\frac{x}{2}$ nieprzerwanych ciągów. Zadanie to jest analogiczne ze znalezieniem $\frac{x}{2} - 1$ początków ciągów, gdyż pierwszy musi zaczynać się od pierwszej jedynki. W takim razie zmuszeni jesteśmy wybrać $\frac{x}{2} - 1$ jedynek spośród $n_1 - 1$, co wynosi oczywiście $\binom{n_1-1}{\frac{x}{2}-1}$. Analogicznie sytuacja wygląda dla zer i nieparzystej liczby nieprzerwanych ciągów.

Aby móc zastosować centralne twierdzenie graniczne, pozostaje nam jedynie obliczyć wartość oczekiwaną oraz wariancję. Niestety, pomimo podobieństwa do rozkładu hipergeometrycznego zadanie to nie jest łatwe, gdyż mamy dwa osobne wzory dla liczb parzystych i nieparzystych. Stąd też pominiemy je, wykorzystując wyliczenia zawarte w pracy dostępnej pod adresem [1].

$$\mathbb{E}R = 2\frac{n_0 n_1}{n} + 1 \tag{4.17}$$

Pomimo skomplikowania rozkładu wartość oczekiwana z wzoru 4.17 jest relatywnie prosta. Możemy się pokusić o jej uzasadnienie. Okazuje się bowiem, że z prawdopodobieństwem $\frac{n_0}{n}$ poprzedni bit był zerem, a więc nowy ciąg otrzymamy, jeśli następny będzie jedynką. Mamy na to $\frac{n_1}{n}$ szans, co uzasadnia fragment $\frac{n_0 n_1}{n}$. Analogiczne wygląda sytuacja w przeciwnym przypadku, stąd mnożymy człon $\frac{n_0 n_1}{n}$ razy 2. Wariancja jest jednak dużo bardziej skomplikowana:

$$Var(R) = \frac{2n_0 n_1(2n_0 n_1 - n)}{n^2(n-1)} \tag{4.18}$$

Dla wzoru 4.18 brak niestety łatwych intuicji. Ważną obserwacją jest natomiast symetryczność tego rozkładu.

Wartość p

Test rozpoczynamy od wyznaczenia liczby bloków 01 i 10 w danym nam losowym wektorze, oznaczenia jej jako ρ, oraz obliczenia wartości n_1 i n_0. Następnie wyliczamy wartość oczekiwaną $\mathbb{E}R$ oraz wariancję $Var(R)$ korzystając z 4.17 oraz 4.18, po czym wyliczamy wartość p:

$$P_{\text{value}} = 1 - \text{erf}\left(\frac{|\rho + 1 - \mathbb{E}R|}{\sqrt{2Var(R)}}\right) \tag{4.19}$$

Za pozytywny wynik uznajemy $P_{\text{value}} > 0.05$.

4.4.4 Test najdłuższego ciągu w bloku

Cel testu

Skupia się on na wyznaczeniu najdłuższego ciągu jedynek w każdym M-bitowym bloku, a następnie ustaleniu, z użyciem testu χ^2 Petersona, jakie jest prawdopodobieństwo, że jednostajny i niezależny generator dał taki rezultat.

Matematyczne podstawy

Wyznaczenie prawdopodobieństwa, że losowy wektor n-bitowy będzie miał najdłuższy ciąg jedynek długości k jest relatywnie trudnym zadaniem. Zacznijmy od oznaczenia funkcji $A_i(j)$, której wartością będzie liczba wszystkich j-bitowych ciągów, a których to długość ciągów jedynek nie przekracza i. W oczywisty sposób dla $j \leqslant i$ $A_i(j) = 2^j$, natomiast w przypadku gdy $j > i$, możemy zaobserwować pewną właściwość. Dla zilustrowania przykładu ustalmy, że $i = 2$, wygenerujemy następujące przedrostki '0', '10' oraz '110' a następnie "dokljmy" do nich wektory z ciągami jedynek nie przekraczającymi i. Jak możemy zauważyć, każdy ciąg wyprodukowany w ten sposób różni się na którymś bicie przedrostka, a więc nie zliczymy żadnego z nich dwa razy. Możemy zatem sformułować tę właściwość następującym wzorem:

$$A_i(j) = \begin{cases} \sum_{t=j-i-1}^{j-1} A_i(t) & j > i \\ 0 & j < 0 \\ 2^i & \text{w przeciwnym przypadku} \end{cases} \tag{4.20}$$

Dokładny dowód możemy znaleźć w artykule Schillinga, umieszczonym w bibliografii [21]. Korzystając ze wzoru 4.20 możemy wyznaczyć prawdopodobieństwa wytopienia wektora z najdłuższym ciągiem jedynek długości k. Niestety metoda ta jest wymagająca obliczeniowo, a wyznaczenie prawdopodobieństw dla bloku rozmiaru M ma złożoność $O(M^2)$. Oznaczmy zatem liczbę bloków z najdłuższym ciąganiem jedynek długości k występującą w wektorze jako O_k. Mając wyznaczone prawdopodobieństwa oraz znając wszystkie wartości O możemy znów zastosować test χ^2 Petersona o najlepszym dopasowaniu.

$$\chi^2 = N \sum_{t=0}^{M} \frac{(\frac{O_t}{N} - p_t)^2}{p_t} \tag{4.21}$$

gdzie:

N - liczba bloków, która wynosi $\lfloor \frac{n}{M} \rfloor$,

p_t - prawdopodobieństwo wytopienia wektora z najdłuższym ciągiem jedynek długości t, równe $\frac{A_t(M) - A_{t-1}(M)}{2^M}$.

Wartość p

Test rozpoczynamy od wyznaczenia wartości O_t oraz p_t, dla $t \in \{0, \ldots, M\}$, po czym następnie obliczamy wartość p:

$$P_{\text{value}} = 1 - \Psi_M \left(N \sum_{t=0}^{M} \frac{(\frac{O_t}{N} - p_t)^2}{p_t} \right) \tag{4.22}$$

Za pozytywy wynik uznajemy $P_{\text{value}} > 0.05$.

4.4.5 Test nieokresowości

Cel testu

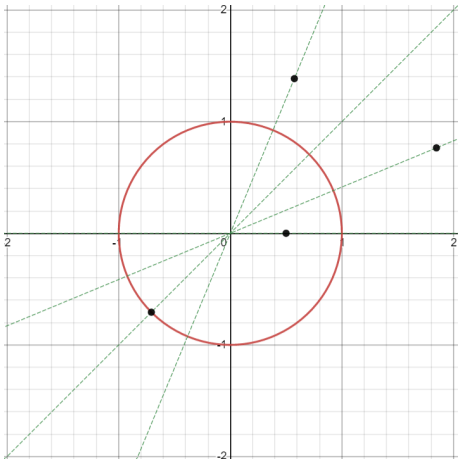
Test skupia się na sprawdzeniu czy wektor t-bitowy zawiera jakieś zależności cykliczne (np. czy co 5-ty bit ma większe szanse być zerem niż jedynką). Wykorzystamy w tym celu transformatę Fouriera, która pozwala wyróżnić z funkcji okresowej poszczególne występujące w niej częstotliwości, a następnie porównamy otrzymany wynik z teoretycznym rozkładem współczynników transformaty.

Matematyczne podstawy

Na początek przypomnijmy, czym jest transformata Fouriera. Dla wektora liczb $X_{n\in\{0,\ldots,N-1\}}$ generuje ona następujący wektor liczb zespolonych:

$$Y_k = \sum_{n=0}^{N-1} X_n e^{-\frac{i2\pi}{N}kn} = \sum_{n=0}^{N-1} X_n \left(\cos\left(\frac{2\pi}{N}kn\right) + i \sin\left(\frac{2\pi}{N}kn\right) \right) \tag{4.23}$$

gdzie $i = \sqrt{-1}$. Jej idea opiera się na umieszczeniu punktów wokół początku układu współ-



Rysunek 4.2: Wizualizacja powstawania transformaty Fouriera dla $X = [0.5,\ 2,\ -1,\ 1.5,\ \ldots]$, wykonana w aplikacji Desmos

rzędnych tak, żeby miary kątowe między kolejnymi punktami były identyczne, a odległości od początku układu współrzędnych równe wartościom wektora X (jak możemy zobaczyć na rysunku 4.2), a następnie obliczeniu średniej pozycji tych punktów. W celu przeprowadzenia testu musimy przyjąć następujące twierdzenia:

- Zarówno $\sqrt{\frac{2}{t}} Im(Y_k)$ oraz $\sqrt{\frac{2}{t}} Re(Y_k)$ przy $t \to \infty$ mają rozkład normalny $\mathcal{N}(0, 1)$,
- Zarówno $\sqrt{\frac{2}{t}} Im(Y_k)$ oraz $\sqrt{\frac{2}{t}} Re(Y_k)$ przy $t \to \infty$ są od siebie niezależne,

gdzie $Re(z)$ zwraca część rzeczywistą liczby zespolonej, a $Im(x)$ oznacza wartość urojoną liczby. Dowód tych twierdzeń odnaleźć można w pracy *Randomness evaluation with the discrete fourier transform test basedon exact analysis of the reference distribution* [17]. Mając te dwa rezultaty możemy w łatwy sposób wywnioskować, iż:

$$\frac{2}{n}|Y_k|^2 \sim \chi^2_2(\text{rozkład chi kwadrat z 2 stopniami swobody}) \tag{4.24}$$

Znając dokładny rozkład dla $\frac{2}{n}|Y_k|^2$, przy $n \to \infty$ możemy się podjąć się wyznaczenia wartości krytycznej κ:

$$\begin{aligned} P(|Y_k| \leqslant \kappa) &= 0.95 \\ \int_0^{\frac{2}{n}\kappa^2} \frac{1}{2} e^{-\frac{y}{2}} dy &= 0.95 \\ 1 - e^{-\frac{\kappa}{n}} &= 0.95 \\ \kappa &= \sqrt{-n \ln(0.05)} \end{aligned} \tag{4.25}$$

Ze względu na własność $|Y_k| = |\overline{Y_{N-k}}|$, wykonujemy testy jedynie dla $k \leqslant \frac{n}{2}$. Ponieważ ten test składa się z $\frac{n}{2}$ testów szansa, że wszystkie z nich przejdą pomyślnie test wynosi $0.95^{\frac{n}{2}}$, co szybko dąży do 0. Dlatego też przy ocenie rezultatu testu musimy pamiętać o tym, iż 5% z nich powinno uzyskać wynik negatywny. Mamy więc do czynienia z rozkładem Bernuliego.

$$Z \sim \mathcal{B}(\frac{n}{2}, 0.95) \tag{4.26}$$

Jest to znany rozkład, którego parametry wynoszą:

$$\begin{aligned} \mathbb{E}Z &= \frac{n}{2} 0.95 \\ Var(Z) &= \frac{n}{2}(0.05)(0.95) \end{aligned} \tag{4.27}$$

Możemy zatem korzystając z centralnego twierdzenia granicznego przybliżyć go rozkładem normalnym:

$$\mathcal{N}\left(\frac{n}{2}0.95, \frac{n}{2}(0.05)(0.95)\right) \tag{4.28}$$

Wartość p

Celem ustalenia wartości p najpierw dokonamy szybkiej transformaty Fouriera na wektorze bitów zmapowanym funkcją $f(x) = 2x - 1$ (a więc zmieniającą 0 na -1), po czym wyliczamy wektor r:

$$r_k = [\![|Y_k| < \sqrt{-n \ln(0.05)}]\!] \tag{4.29}$$

gdzie $[\![wyrażenie]\!]$ jest funkcją prawdy i przyjmuje 1 (jeśli zdanie jest prawdziwe) i 0 (w przeciwnym wypadku).

$$P_{\text{value}} = 1 - \text{erf}\left(\frac{|\sum_{j=0}^{\frac{n}{2}} r_k - \mathbb{E}Z|}{\sqrt{2Var(Z)}}\right) \tag{4.30}$$

Test uznajemy za zaliczony jeśli $P_{\text{value}} > 0.05$.

4.4.6 Test nakładających się wzorców

Cel testu

Celem testu jest sprawdzenie, czy wstępowanie danego wzorca $B = b_1 b_2 b_3 \ldots b_m$ w otrzymanym wektorze pokrywa się z teoretycznym rozkładem przy założeniu jego jednostajności oraz niezależności.

Matematyczne uzasadnienie

Obliczenie dokładnego prawdopodobieństwa dla poszczególnej liczby występowania wzorca B w losowym wektorze $X = x_1x_2x_3 \dots x_n$ jest bardzo trudnym zadaniem. Możemy poznać dokładne wartości dla wzorców długości 4 [22], jednak samo wyliczenie wszystkich prawdopodobieństw jest niełatwe ze względu na złożoność obliczeniową. Stąd też użyjemy oszacowania rozkładem normalnym. Żeby to zrobić, musimy najpierw wyznaczyć wartość oczekiwaną oraz wariancję. Aby je obliczyć potrzebujemy nowego wektora zmiennych losowych.

$$Y_k = [\![x_{k-m+1}x_{k-m+2} \dots x_k = b_1b_2 \dots b_m]\!] \tag{4.31}$$

Wektor Y z definicji 4.31 informuje nas, czy na k-tej pozycji kończy się poszukiwany przez nas wzorzec. Z łatwością możemy obliczyć, że dla każdego $k \in \{m, \dots, n\} Y_k = \frac{1}{2^m}$. Oznaczmy zmienną losową liczbę występowania wzorca B w losowym wektorze jako Q. Zatem:

$$\mathbb{E}Q = \mathbb{E}\sum_{k=m}^{n} Y_k = \sum_{k=m}^{n} \mathbb{E}Y_k = \sum_{k=m}^{n} \frac{1}{2^m} = \frac{n-m+1}{2^m} \tag{4.32}$$

Wzór 4.32 wynika z faktu, iż wartość oczekiwana sumy jest zawsze sumą wartości oczekiwanych, przez co nie musimy przejmować się problemem zależności wektora Y. Prawo wariancji sumy jest bardziej skomplikowane:

$$Var(A+B) = Var(A) + Var(B) + cov(A,B) \tag{4.33}$$

Jak widzimy na wzorze 4.33, aby policzyć sumę wariancji konieczne jest ustalenie kowariancji zmiennych losowych. Kowariancję definiujemy wzorem 4.34.

$$cov(A,B) = \mathbb{E}[(A - \mathbb{E}A)(B - \mathbb{E}B)] = \mathbb{E}[AB] - \mathbb{E}A\mathbb{E}B \tag{4.34}$$

Aby wyliczyć wartość wariancji dla $\sum_{k=m}^{n} Y_k$, zastosujemy proces iteracyjny. Najpierw wyliczymy $Var(Y_m + Y_{m+1})$, następnie $Var((Y_m + Y_{m+1}) + Y_{m+2})$, żeby na samym końcu poznać wartość $Var(\sum_{k=m}^{n-1} Y_k + Y_n)$. Musimy być zatem w stanie ustalić wartość wyrażenia 4.35 dla każdego t.

$$\mathbb{E}\left[Y_t \sum_{k=m}^{t-1} Y_k\right] = \mathbb{E}\left[\sum_{k=m}^{t-1} Y_tY_k\right] = \sum_{k=m}^{t-1} \mathbb{E}\left[Y_tY_k\right] \tag{4.35}$$

Ponieważ Y jest wektorem zer i jedynek oraz $0x = 0$ możemy wzór 4.35 zapisać w formie wyrażenia 4.36, pomijając zbędne zera.

$$\sum_{k=m}^{t-1} \mathbb{E}\left[Y_tY_k\right] = \sum_{k=m}^{t-1} \mathbb{E}\left[Y_k | Y_t = 1\right] P(Y_t = 1) = \sum_{k=m}^{t-1} \mathbb{E}\left[Y_k | x_{k-m+1} \dots x_k = b_1 \dots b_m\right] P(Y_t = 1) \tag{4.36}$$

Widzimy zatem, iż musimy obliczyć oczekiwaną liczbę wzorców B w wektorze długości $t-1$ znając ostanie $m-1$ pozycji tego wektora. Dla $Y_m, \dots, Y_{t-m}$ wartość oczekiwana się nie zmienia i wynosi $\frac{1}{2^m}$. Dla pozostałych $Y_{t-m+1}, \dots, Y_{t-1}$ natomiast koniecznym będzie wyliczenie wartości, które zależą one od samego wzorca B. Kończąc, przemnożymy otrzymaną wartość oczekiwaną przez prawdopodobieństwo, że $Y_t = 1$, a więc przez $\frac{1}{2^m}$.

Wartość p

Wzorzec B jest nam dany. Pierwszym krokiem będzie wyliczenie jego wartości oczekiwanej $\mathbb{E}Q$, wariancji $Var(Q)$ oraz liczby wystąpień wzorca (do której ustalenia najlepiej zastosować algorytm Knutha-Morrisa-Pratta), oznaczonej Q. Podstawiając do wzoru, otrzymujemy:

$$P_{\text{value}} = 1 - \operatorname{erf}\left(\frac{|Q - \mathbb{E}Q|}{\sqrt{2Var(Q)}}\right) \tag{4.37}$$

Test uznajemy za zaliczony jeśli $P_{\text{value}} > 0.05$.

4.4.7 Test złożoności liniowej

Cel testu

W tym teście skupimy się na sprawdzeniu jak dobrze złożoność liniowa M-bitowych bloków wektora losowego pokrywa się z oczekiwanymi rezultatami. Złożoność linową zdefiniujemy jako najkrótszy rejestr przesuwający z liniowym sprzężeniem zwrotnym (dalej *LFSR*) który może wygenerować ten ciąg. W celu obliczenia złożoności liniowej bloku wykorzystamy algorytm Berlekampa–Masseya [15].

Matematyczne podstawy testu

Wyznaczmy teoretyczny rozkład złożoności linowej Λ przy założeniu jednostajności i niezależności. Obliczenia zostały oparte na pracy R. A. Rueppela [20]. Dla każdego ciągu n-bitów s^n możemy wyliczyć jego złożoność linową $\Lambda(s^n)$. Korzystając z pracy J. Masseya [15] wiemy, że zachodzi następująca zależność rekurencyjna:

$$\Lambda(s^{n+1}) = \begin{cases} \Lambda(s^n) & \text{jeśli najkrótszy LFSR dla } s^n \text{ produkuje odpowiedni bit} \\ \Lambda(s^n) & \text{jeśli najkrótszy LFSR dla } s^n \text{ nie jest odpowiedni, ale } \Lambda(s^n) \geqslant \frac{n}{2} \\ n - \Lambda(s^n) & \text{jeśli najkrótszy LFSR dla } s^n \text{ nie jest odpowiedni i } \Lambda(s^n) < \frac{n}{2} \end{cases} \tag{4.38}$$

Oznaczmy teraz liczbę ciągów długości n o złożoności liniowej L jako $N_n(L)$. Ciąg s^{n+1} składa się z ciągu s^n z dodanym jednym bitem. Jeśli ten dodatkowy bit odpowiada już istniejącemu najkrótszemu *LFSRowi*, jego złożoność się nie zmienia. Dla przypadku kiedy $\Lambda(s^{n+1}) = L' < \frac{n}{2}$, ponieważ złożoność liniowa nie może spaść po dodaniu bitu, do tej kategorii zaliczymy jednie ciągi n bitów, którym dodano odpowiadający bit, zatem $N_{n+1}(L') = N_n(L')$. Ciągi, którym dodano nieodpowiadający bit zmienią swoją złożoność. Dlatego też w przypadku $L' > \frac{n+1}{2}$ liczba takich ciągów wyniesie $N_n(n - L')$ (są to te którym wzrosła złożoność) plus $2N_n(L')$ (ponieważ dla $\Lambda(s^n) \geqslant \frac{n+1}{2}$ złożoność się nie zmienia). W przypadku $L' = \frac{n+1}{2}$ ciągi o długości n z dodanym dowolnym bitem zachowają swoją złożoność, więc $N_{n+1}(L') = 2N_n(L')$.

$$N_{n+1}(L) = \begin{cases} 2N_n(L) + N_n(n-L) & \text{jeśli } L > \frac{n+1}{2} \\ 2N_n(L) & \text{jeśli } L = \frac{n+1}{2} \\ N_n(L) & \text{jeśli } L < \frac{n+1}{2} \end{cases} \tag{4.39}$$

Wzór 4.39 możemy uprościć do formy 4.40, której dowód znaleźć możemy w pracy R. A. Rueppela [20].

$$N_n(L) = \begin{cases} 1 & \text{jeśli } n \geqslant L > 0 \\ 2^{\{2n-2L, 2L-1\}} & \text{w przeciwnym przypadku} \end{cases} \tag{4.40}$$

Ze względu na dużą dysproporcję w ilości ciągów dla odpowiednich złożoności, zastosujemy grupowanie wyników w T-przedziałów o różnej długości dla M=1024 ($[0, 255]$, ..., $[510, 511]$, $[512]$, $[513, 514]$, ..., $[786, 1024]$). Zastosujemy zatem test χ^2 Petersona:

$$\chi^2 = N \sum_{k=0}^{T-1} \frac{(\frac{O_k}{N} - p_k)^2}{p_k} \tag{4.41}$$

gdzie:

N - liczba M-bitowych rozłącznych przedziałów,

T - liczba kategorii,

O_k - liczba przedziałów zaliczonych do k-tej kategorii,

p_k - prawdopodobieństwo przynależności do k-tej kategorii.

Wartość p

Test rozpoczynamy od wyznaczania prawdopodobieństwa p dla każdej z kategorii. Następnie dzielimy wektor X na M-bitowe bloki. Korzystając z algorytmu Berlekampa–Masseya wyliczamy ich złożoności i dodajemy do odpowiedniej kategorii O. Następnie obliczamy wartość p:

$$P_{\text{value}} = 1 - \Psi_{(T-1)}\left(N \sum_{k=0}^{T-1} \frac{(\frac{O_k}{N} - p_k)^2}{p_k}\right) \tag{4.42}$$

Za pozytywy wynik uznajemy $P_{\text{value}} > 0.05$.

4.4.8 Test ostatniego przecięcia

Cel testu

Test skupia się na sprawdzeniu, kiedy po raz ostatni losowa ścieżka wygenerowana na podstawie wektora losowego X długości n osiągnie wartość zero (innymi słowy kiedy po raz ostatni wyrówna się ilość zer i jedynek w wektorze X), a następnie porównaniu tej wartości do teoretycznego rozkładu.

Matematyczne podstawy

Pierwszym krokiem w wyznaczeniu ostatniego punku osiągnięcia 0 będzie zdefiniowanie wektora Y, który określa naszą pozycję w kroku t:

$$Y_t = \sum_{k=0}^{t-1}(2X_k - 1) \tag{4.43}$$

Naszym celem jest wyznaczenie największego t takiego, że $Y_t = 0$. Dla znanego wektora X obliczenie tej wartości jest prostym zadaniem o złożoności obliczeniowej $O(n)$. Aby przeprowadzić test musimy jednak poznać również teoretyczny rozkład ostatniego przecięcia dla wektorów długości o n. Posłużymy się zatem centralnym twierdzeniem granicznym, które pozwoli nam oszacować ten ruch jednowymiarowym ruchem Browna (ściślej: procesem Weinera). Dla takich ruchów możemy zastosować drugie prawo *arcus sinusa*, które brzmi:

$$P(L \leqslant x) = \frac{2}{\pi}\arcsin(\sqrt{x}) \tag{4.44}$$

gdzie L oznacza ostatni punkt w którym ruch Browna przyjmuje wartość 0. Dowód twierdzenia 4.44 możemy znaleźć w pracy *Arcsine laws and its simulation and application* [13]. Rozkład ten jest dojść płaski, a wartość oczekiwana nie pokrywa się z modą. W związku z tym dla $x \in [0, 1]$ podzielimy wektor X na N 2047-bitowych rozłącznych wektorów. Dodatkowo pogrupujemy wyniki na 16 kategorii, po czym zastosujemy test χ^2 Petersona.

$$\chi^2 = N \sum_{k=0}^{15} \frac{(\frac{O_k}{N} - p_k)^2}{p_k} \tag{4.45}$$

gdzie:

N - liczba 2047-bitowych rozłącznych przedziałów, a więc $\lfloor \frac{n}{2047} \rfloor$,

O_k - liczba przedziałów zaliczonych do k-tej kategorii,

p_k - prawdopodobieństwo przynależności do k-tej kategorii.

Wartość p

Rozpoczynamy obliczając prawdopodobieństwo dla każdej z kategorii:

$$p_k = \frac{2}{\pi}\left(\arcsin\left(\sqrt{\frac{128(k+1)}{2048}}\right) - \arcsin\left(\sqrt{\frac{128k}{2048}}\right)\right) \tag{4.46}$$

W kolejnym kroku dzielimy wektor na 2047-bitowe przedziały zliczając, ile wektorów należy do każdej kategorii. Następnie obliczamy wartość p:

$$P_{\text{value}} = 1 - \Psi_{15}\left(N\sum_{k=0}^{15}\frac{(\frac{O_k}{N} - p_k)^2}{p_k}\right) \tag{4.47}$$

Za pozytywy wynik uznajemy $P_{\text{value}} > 0.05$.

4.4.9 Test największej sumy

Cel testu

Test skupi się na wyznaczeniu maksymalnej wartości losowej ścieżki wygenerowanej w oparciu o n-bitowy wektor losowy X, po czym otrzymany wynik zostanie porównana do teoretycznego rozkładu tej zmiennej.

Matematyczne podstawy

Na początku przedstawimy kilka użytecznych definicji których użyjemy w dalszym rozumowaniu. Całość jest inspirowana notatkami z wykładu dostępnego pod adresem [19].

$$\begin{aligned} W_t &= \sum_{k=1}^{t}(2X_k - 1) \\ T_a &= \min\{t \leqslant 0 : W_t = a\} \\ M_t &= \max_{0\leqslant k\leqslant t} W_k \end{aligned} \tag{4.48}$$

Ze względu na jednostajność rozkładu oraz niezależność, możemy dodatkowo powiedzieć, że:

$$P(W_t > a | T_a < t) = P(W_t < a | T_a < t) = \frac{1 - P(W_t = a)}{2} \tag{4.49}$$

Równanie 4.49 mówi, że jeśli wiemy, że w jakimś punkcie T_a osiągnęliśmy wartość a to szansa, że jesteśmy poniżej tej wartości jest identyczna jak ta, że jesteśmy powyżej jej. Dodatkowo, z definicji możemy wyczytać, że:

$$(\forall t \in \mathbb{N})(\forall \in \mathbb{N}) M_t \geqslant a \iff T_a \leqslant t \tag{4.50}$$

czyli, że jeśli maksimum do punktu t jest większe od a to w którymś punkcie przed t nasz ruch musiał przyjąć wartość a.

$$\begin{aligned} &P(T_a \leqslant t) = \\ &= P(T_a \leqslant t \wedge W_t > a) + P(T_a \leqslant t \wedge W_t < a) + P(T_a \leqslant t \wedge W_t = a) = \\ &= 2P(T_a \leqslant t \wedge W_t > a) + P(T_a \leqslant t \wedge W_t = a) = \\ &= 2P(M_t \geqslant a \wedge W_t > a) + P(W_t = a) = \\ &= 2P(W_t > a) + P(W_t = a) \end{aligned} \tag{4.51}$$

W równaniu 4.51 sprowadziliśmy problem znalezienia rozkładu największej wartości w naszej ścieżce do znalezienia rozkładu wartości w punkcie t. Ma on znany rozkład dwumianowy Bernoulliego. My natomiast przybliżmy go rozkładem normalnym korzystając z centralnego twierdzenia granicznego. Możemy zatem ponownie skorzystać z 4.8.

$$P(M_t \geqslant a) = P\left(\frac{M_t}{\sqrt{t}} \geqslant \frac{a}{\sqrt{t}}\right) = 2P\left(\frac{W_t}{\sqrt{t}} \geqslant \frac{a}{\sqrt{t}}\right) = 2\left(1 - \Phi\left(\frac{a}{\sqrt{t}}\right)\right) \tag{4.52}$$

Wartość p

Pierwszym krokiem jest znalezienie największej wartości w losowej ścieżce (M_n). Podstawiając ją do wzoru otrzymamy wartość p:

$$P_{\text{value}} = 1 - \operatorname{erf}\left(\frac{M_n}{\sqrt{2n}}\right) \tag{4.53}$$

Jeśli $P_{\text{value}} > 0.05$, to uznajemy test za zaliczony.

4.4.10 Test rang macierzy

Cel testu

Celem testu jest sprawdzenie złożoności linowej wektora losowego. W tym celu stworzymy losowe macierze binarne wymiaru $m \times m$, a następnie wyznaczymy ich rangi. Porównamy otrzymane wynik z ich teoretycznym rozkładem dla prawdziwie losowej macierzy.

Matematyczne podstawy

Aby przeprowadzić ten test, musimy najpierw wyznaczyć prawdopodobieństwo otrzymania losowej macierzy o wymiarach $m \times m$ z rangą r. Zadanie to zaczniemy od wyznaczenia prawdopodobieństwa, iż wektor losowy długości m będzie liniowo zależny od zbioru wektorów o randze $(m-k)$. Wiemy, że dla przestrzeni liniowej o k wymiarach wyznaczonej przez wektory $\mathcal{V} = (v_1, v_2, \ldots, v_t)$ możemy znaleźć bazę $\mathcal{U} = (u_1, u_2, \ldots, u_k)$, która tworzy identyczną przestrzeń, a wektory są parami rozłączne, co oznacza że jeśli jeden wektor na t-tej pozycji posiada jedynkę, to wszystkie pozostałe na t-tej pozycji mają zero, formalny zapis w następującej formule:

$$\begin{gathered} u_i = (u_i^1, u_i^2, \ldots, u_i^m) \\ (\forall i, j) u_i^j = 1 \rightarrow (\neg \exists t \neq i) u_i^t = 1 \end{gathered} \tag{4.54}$$

Aby nowy wektor q był liniowo zależny od bazy $\mathcal{U}$ muszą zachodzić własności 4.55 oraz 4.56:

$$(\forall i) q \,\&\, u_i = 0 \vee q \,\&\, u_i = u_i \tag{4.55}$$

$$(\forall i) q \,\&\, \overline{\left(\sum_{i=1}^{k} u_i\right)} \neq 0 \tag{4.56}$$

gdzie & oznacza iloczyn bitowy dwóch wektorów, operację sumy dwóch wektorów zdefiniujemy jako ich sumę bitową, a $\overline{a}$ oznacza negację bitową wektora a. Gdyby nie zachodziła własność 4.55 (a więc istnieje wektor u_i, który pokrywa się jedynie w części z wektorem q), to niemożliwym jest stworzenie za pomocą operacji sumy rozłącznej z wektorów bazy $\mathcal{U}$ wektora q. Gdybyśmy dodali do sumy rozłącznej wektor u_i nastąpiłoby dodanie bitów, które nie należą do q sumy, natomiast jego nie dodanie spowodowałoby, że część bitów, które powinny być jedynką pozostaną zerem. Jeśli natomiast nie zachodziłoby 4.56 koniecznym byłoby ustawienie bitu, który nie jest jedynką w żadnym wektorze z bazy $\mathcal{U}$, co jest niemożliwe. Z tych własności możemy wyciągnąć wniosek, iż dla bazy $\mathcal{U}$, istnieje 2^k wektorów zależnych, a prawdopodobieństwo wylosowania

takiego wektora wynosi $\frac{2^k}{2^n}$ Pozostaje nam zatem wyliczyć prawdopodobieństwo uzyskania macierzy wymiaru $m \times m$ o randze $(m-k)$. Wiemy, iż została zbudowana z k liniowo zależnych oraz $(m-k)$ liniowo niezależnych rzędów. Prawdopodobieństwo otrzymania liniowo niezależnego wektora zależy jedynie od rangi poprzednich wierszy. Niezależnie od ich pozycji stanowią one człon $\prod_{i=0}^{m-k-1}(1-2^{-(m-i)})$. Pozostaje nam jeszcze wyliczenie drugiego członu, a więc prawdopodobieństwo wylosowania k liniowo zależnych wierszy. Niestety, ich pozycja ma wpływ na prawdopodobieństwo, przez co zmuszeni jesteśmy zsumować wszystkie kombinacje. Ostatecznie wzór 4.57 wyznacza prawdopodobieństwo uzyskania losowej macierzy $m \times m$ o randze $(m-k)$. Dokładny dowód można znaleźć w książce *Random graphs* [11].

$$P_m(k) = 2^{-k^2}\left(\prod_{i=0}^{m-k-1}(1-2^{-(m-i)})\right)\left(\sum_{0\leqslant i_1\leqslant \ldots i_k\leqslant m-k} 2^{-(i_1+i_2+\cdots+i_k)}\right) \tag{4.57}$$

Formuła 4.57 jest niestety wymagająca obliczeniowo, w związku z tym wyniki podzielimy na 10 kategorii: $k=0$, $k=1$, ..., $k=8$, $k>8$, a następnie wykonamy test χ^2 Petersona.

$$\chi^2 = N\sum_{k=0}^{9}\frac{(\frac{O_k}{N}-p_k)^2}{p_k} \tag{4.58}$$

gdzie:

N - liczba macierzy $m \times m$,

O_k - liczba przedziałów zaliczonych do k-tej kategorii,

p_k - prawdopodobieństwo przynależności do k-tej kategorii wyliczone na podstawie wzoru 4.57.

Wartość p

Pierwszym krokiem jest wyliczenie teoretycznego prawdopodobieństwa dla każdej z dziesięciu kategorii, ustalenie rangi każdej macierzy oraz dodanie jej do odpowiedniej kategorii O_i. Następnie wyliczamy wartość χ^2, oraz wartość p:

$$P_{\text{value}} = 1 - \Psi_9\left(N\sum_{k=0}^{9}\frac{(\frac{O_k}{N}-p_k)^2}{p_k}\right) \tag{4.59}$$

Przy wartości $P_{\text{value}} > 0.05$ uznajemy test za zaliczony.

4.5 Wyniki testów

Po zapoznaniu się z metodami testowania możemy przeanalizować uzyskane wyniki.

4.5.1 Wyniki naszych testów

Na podstawie przedstawionych powyżej testów stworzyliśmy program sprawdzający losowość w języku Python. Do testów wykorzystaliśmy 12 megabajtowe wektory losowych bitów wygenerowane przez odpowiednie generatory. Test jedynek w bloku oraz test najdłuższego ciągu dla bloków 1024 bitów zostały zmodyfikowane poprzez połączenie mało prawdopodobnych kategorii w jedną większą w celu zminimalizowania błędów numerycznych. Pozostałe testy zostały zaimplementowane bez żadnych zmian. Jak widzimy w tablicy 4.3 najgorzej poradził sobie generator *LCM*, co jest spodziewane z uwagi na jego prymitywną konstrukcję. Wyniki generatora *Tarus* oraz *CRNG* są bardzo zbliżone, co może dziwić z względu na to, iż generator *CRNG* powinien znacząco lepszej klasy. Najlepiej w testach wypadł nasz generator. Z uwagi, iż testy opierają się na testach statystach nie można wyciągnąć jednak jednostajnych wniosków na temat jego wyższości nad *CRNG*.

Nazwa testu	LCM	Tarus	CRNG	Nasz generator
Test częstotliwości	0.12882	0.68573	0.51296	0.11257
Test jedynek w bloku (128)	0.00000	1.00000	1.00000	1.00000
Test jedynek w bloku (1024)	0.00000	0.76419	0.86979	0.68492
Test nieprzerwanych ciągów	0.65722	0.78955	0.04002	0.58470
Test najdłuższego ciągu (64)	0.00000	0.99966	0.99996	1.00000
Test najdłuższego ciągu (1024)	0.00000	0.99826	0.69517	0.99922
Test nieokresowości	0.00000	0.86880	0.75849	0.27542
Test nakładających się wzorców	0.48773	0.97426	0.27960	0.82126
Test złożoności liniowej	0.99864	0.925021	0.99146	0.99740
Test ostatniego przecięcia	0.00000	0.89303	0.98706	0.83454
Test największej sumy	0.06399	0.52433	0.71036	0.47557
Test rang macierzy (32x32)	0.84882	0.92983	0.96152	0.94216
Test rang macierzy (16x16)	0.85061	0.00308	0.99792	0.97909

Tablica 4.3: Wyniki naszych testów dla różnych generatorów wuwzględniające wartości p dla poszczególnych testów. Z wyróżnieniem wyniki gdzie wartość p jest mniejsza od poziomu istotności $\alpha = 0.05$

4.5.2 Wyniki testów narzędziem *Diehard*

Dodatkowo, wykorzystując narzędzie *Diehard* [14], przeprowadziliśmy kolejne testy. Dla każdego z nich zwracającego więcej niż jedną wartość p, w tablicy 4.4 została zaprezentowana ostatnia z nich. Podobnie jak w przypadku poprzednich testów, tu również nasz generator wypadł

Nazwa testu	LCM	Tarus	CRNG	Nasz generator
Birthday Spacings	1.00000	0.50792	0.71394	0.03742
Overlapping Permutations	0.82591	0.02708	0.00131	0.54635
Ranks of 32x32 matrices	0.54445	0.75272	0.65900	0.53647
Ranks of 6x8 Matrices	1.00000	0.42814	0.59030	0.32594
Monkey Tests on 20-bit Words	0.00000	0.20653	0.66948	0.89925
Monkey Tests DNA	0.34920	0.34380	0.86220	0.34810
Count the 1‘s in a Stream of Bytes	1.00000	0.32245	0.98657	0.48885
Count the 1‘s in Specific Bytes	0.74763	0.51252	0.59320	0.52139
Parking Lot Test	0.53309	0.02187	0.71684	0.10980
Minimum Distance Test	0.62647	0.00003	0.58250	0.90770
Random Spheres Test	0.08858	0.94746	0.43352	0.77891
The Sqeeze Test	0.10927	0.07114	0.35482	0.81188
Overlapping Sums Test	0.89813	0.15546	0.86684	0.15546
Runs Test	0.34953	0.35924	0.55256	0.91654
The Craps Test	0.56509	0.69635	0.08896	0.65335

Tablica 4.4: Wyniki testów narzędziem *Diehard* [14] dla różnych generatorów z uwzględnieniem wartości p dla poszczególnych testów i wyróżnieniem wyników, gdzie wartość p znajduje się poza przedziałem ufności $0.025 < P_{\text{value}} < 0.975$ dla tego narzędzia

najlepiej w porównaniu do wszystkich innych generatorów. Jest on wyraźnie lepszy od rozwiązania *LCM*, jak również wydaje się być lepszy od dwuch pozostałych generatorów.

Nazwa testu	LCM	Tarus	CRNG	Nasz generator
Frequency	0.12233	0.91141	0.35049	0.12233
Block Frequency	0.00000	0.53415	0.53415	0.53415
Cumulative Sums	0.53415	0.73992	0.35049	0.35049
Runs	0.53415	0.03517	0.35049	0.03517
Longest Run	0.21331	0.53415	0.12233	0.53415
Rank	0.73992	0.53415	0.12233	0.53415
FFT	0.00000	0.00888	0.53415	0.35049
Non Overlapping Template	0.03517	0.91141	0.21331	0.35049
Overlapping Template	0.00000	0.53415	0.06688	0.53415
Universal	0.00000	0.73992	0.73992	0.73992
Approximate Entropy	0.00000	0.21331	0.35049	0.73992
Serial	0.00000	0.91141	0.12233	0.00430
Linear Complexity	0.73992	0.35049	0.06688	0.21331

Tablica 4.5: Wyniki testów narzędziem *NIST test suite* [2] dla różnych generatorów z uwzględnieniem wartości p dla poszczególnych testów i wyróżnieniem wyników, gdzie wartość p jest mniejsza od poziomu istotności $\alpha = 0.05$

4.5.3 Wyniki testów narzędziem *NIST test suite*

Z względu na przejrzysty interfejs narzędzia *NIST test suite* [2] przeprowadziliśmy dodatkowe testy z jego wykorzystaniem. Użyliśmy identycznych danych jak w dwóch poprzednich testach, podzielonych na 10 strumieni o długości 10000000 bitów każdy. Dla spójności zastosowaliśmy poziom istotności $\alpha = 0.05$ (taki jak dla poprzednich testów), pomimo zalecanego $\alpha = 0.01$. W przypadku testów, które narzędzie wykonuje kilkukrotnie w celu poprawienia czytelności w tablicy 4.5 zaprezentowano jedynie ostanie wyniki. Możemy w niej zobaczyć, że nasz generator wypada gorzej od generatora *CRNG*. Szczególnie niepokoić może wynik testu *Serial* dla którego wartość p wynosi 0.00430. Oznacza to, iż w jedynie 0.43% przypadków poprawny generator wygeneruje tak skrajne wyniki. Tak jak w poprzednich testach najgorzej wypada *LCM*. Generator *Tarus* radzi sobie lepiej, jednak również nie przeszedł części testów. *CRNG*, w przeciwieństwie do poprzednich narzędzi, przeszedł wszystkie testy bezbłędnie, jednak w wielu testach wartość p znajdowała się na granicy przedziału ufności.

Podsumowanie

Celem pracy było stworzenie konceptu generatora "prawdziwych" liczb losowych opartego o entropię pochodzącą z czasów dostępu do zasobów sieciowych oraz zaimplementowanie stworzonego narzędzia w celu wykazania jego poprawności. Wektor liczb będący wynikiem pracy takiego generatora musi spełniać warunek jednostajności oraz niezależności, zatem dodatkowym celem było przygotowanie narzędzi umożliwiających potwierdzenie poprawności rozwiązania poprzez przetestowanie wyników generatora pod tym kątem.

Początek pracy omawia czy możliwe jest stworzenie generatora prawdziwych liczb losowych, a który to problem można sprowadzić do pytania na temat determinizmu świata. Kolejna sekcja pochyla się nad analizą dwóch podejść do generowania liczb losowych. Jednym z nich było generowanie liczb na podstawie entropii świata zewnętrznego, natomiast drugim - koncepcja odrzucenia wymogu niezależności na rzecz generowania liczb pseudolosowych. Kolejno zaprezentowany został pomysł na źródło entropii w postaci czasów dostępów do zasobów sieciowych (takich jak czas odpowiedzi na ping) wraz z przesłankami świadczącymi o jego poprawności, po czym przeanalizowano teoretyczne problemy takiego podejścia. Czasy pingów mają specyficzny rozkład, w którym występuje prawo Benforda. Oznacza to, iż prawdopodobieństwo otrzymania jedynki na k-tym bicie nie jest stałe i jest zależne od długości liczby. Innym przeanalizowanym problemem była maksymalna precyzja pomiarów czasu, której możemy się spodziewać na współczesnych komputerach oraz jej wpływ na wydajność. Następny rozdział skupiał się na analizie i rozwiązaniu problemów technicznych, które rodził nasz generator. Został w nim omówiony pomysł na niwelowanie skutków niejednostajnego rozkładu bitów poprzez odrzucenie bitów o nazbyt odchylonym rozkładzie. Przedstawiono również kryteria rozkładu bitów akceptowalnych dla naszego generatora oraz zaprezentowano szczegółowy pseudokod generatora wraz z jego implementacją w języku Python i jej omówieniem. Dokonana została również analiza różnych protokołów sieciowych, które można wykorzystać w celu pozyskania czasów dostępów do zasobów sieciowych. Końcem rozdziału przeprowadzono analizę wydajności już napisanego generatora - zarówno teoretyczną, jak i w środowisku testowym. Ostatni rozdział skupia się na problemie pokazania jednostajności oraz niezależności wektora liczb losowych. Zaczyna się on od wyjaśnienia, dlaczego nie może istnieć matematyczny dowód takich własności, a następnie proponuje testy statystyczne jako alternatywę. Ich idea została dokładanie omówiona uwzględniając problemy jakie rodzą. Następną sekcję poświęcono analizie istniejących już narzędzi testowania losowości oraz wyborze trzech generatorów różnej klasy, z którymi zostanie porównane nasze rozwiązanie. Wyjaśniono następnie, iż gotowe narzędzia nie posiadają satysfakcjonującego uzasadnienia swojej poprawności, przez co przygotowanych 10 testów inspirowanych istniejącymi rozwiązaniami, wzbogaconych o matematyczne uzasadnienie swojej poprawności. Rozdział kończy się zaprezentowaniem wyników przeprowadzonych testów - zarówno tych stworzonych na potrzebę pracy, jak i tych już istniejących.

Niniejsza praca wykazała możliwość stworzenia generatora "prawdziwych" liczb losowych opartego o czasy dostępów do zasobów sieciowych. Generowany przez niego ciąg liczb spełnia wymogi jednostajności i niezależności nie gorzej niż generatory rekomendowane do zastosowań kryptograficznych. Oznacza to, że teoretycznie nasz generator również mógłby być użyty w miejscach wymagających generowania losowych kluczy bez uszczerbku dla bezpieczeństwa tych systemów. Problemem, którego nie udało się w niniejszej pracy rozwiązać (i który najprawdopodobniej niemożliwy jest do rozwiązania) pozostaje wydajność tego rozwiązania. Spowodowany

jest on wykorzystaniem mocno ograniczonego zasobu, jakim jest dostęp do sieci. Z tego powodu, generator ten najpewniej nie znajdzie zastosowania w przemyśle i pozostanie konceptem czystko akademickim. Może on jednak posłużyć jako generator wartości początkowych (seedu) dla generatorów liczb pseudolosowych lub jako dodatkowe źródło entropii dla innych rozwiązań.

Dalszych badań wymaga problem znaczącego spadku wydajności w przypadku równoległego pingowana w środowisku testowym. Koniecznym okazuje się także odpowiedź na pytanie, czy problem dotyczy tego jednego środowiska, czy jest szerzej reprodukowany oraz jakie są przyczyny gubienia pakietów, mechanizm wyboru bitów o akceptowanym rozkładzie można by również udoskonalić o użycie bitów o nieakceptowanych rozkładach w celu przekształcenia ich pewnej skończonej ilości w bit o rozkładzie akceptowanym dla naszego generatora, co pozwoliłoby w nieznaczny sposób podnieść wydajność systemu. Innym potencjalnym sposobem na jej podniesienie jest analiza, czy zawartość otrzymywanych pakietów może służyć jako dodatkowe źródło entropii. Z punktu widzenia zastosowań kryptograficznych, najważniejsze jest jednak sprawdzenie, jakie są możliwości manipulowania otrzymanymi czasami dostępów do zasobów sieciowych. Nurtującymi pozostaje pytanie. Czy przeprowadzanie ataku z wykorzystaniem fałszywej bramki domyślnej pozwoliłoby by na manipulowanie wynikami generatora? Czy jednak ze względu na odrzucenie najbardziej znaczących bitów operacja ta jest z praktycznego punktu widzenia niemożliwa do przeprowadzenia?

Bibliografia

[1] Ncss analysis of runs. *https://ncss-wpengine.netdna-ssl.com/wp-content/themes/ncss/pdf/Procedures/NCSS/Analysis˙of˙Runs.pdf*. Accessed: 2021-11-07.

[2] Random bit generation. *https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software*, 2016. Accessed: 2021-11-04.

[3] Pep 564 – add new time functions with nanosecond resolution. *https://www.python.org/dev/peps/pep-0564/*, 2017. Accessed: 2021-11-07.

[4] L. Bassham, A. Rukhin, J. Soto, J. Nechvatal, M. Smid, S. Leigh, M. Levenson, M. Vangel, N. Heckert, D. Banks. A statistical test suite for random and pseudorandom number generators for cryptographic applications, 2010-09-16 2010.

[5] F. Benford. The law of anomalous numbers. *Proceedings of the American Philosophical Society*, 78(4):551–572, 1938.

[6] R. G. Brown. Dieharder: A random number test suite. *https://webhome.phy.duke.edu/˜rgb/General/dieharder.php*, 2021. Accessed: 2021-11-04.

[7] R. Fainlight, R. J. Littman. *The Theban plays : Oedipus the king, Oedipus at Colonus, Antigone*. Johns Hopkins University Press, Baltimore, 2009.

[8] D. J. Griffiths. *Introduction to Quantum Mechanics (2nd Edition)*. Pearson Prentice Hall, wydanie 2nd, Kwi. 2004.

[9] T. P. Hill. A statistical derivation of the significant-digit law. *Statistical Science*, 10(4):354–363, 1995.

[10] P. H. T. Holbach. *The system of nature, or, The laws of the moral and physical world [microform] / done from the original French of M. de Mirabaud.* [S.l, wydanie 3rd ed. with additions., 1817.

[11] V. F. Kolchin. *Random Graphs*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1998.

[12] P. L'Ecuyer. Empirical testing of random number generators. *http://simul.iro.umontreal.ca/testu01/tu01.html*, 2002. Accessed: 2021-11-04.

[13] L. J. Liu Qiang. Arcsine laws and its simulation and application. unpublished.

[14] G. Marsaglia. The marsaglia random number cdrom including the diehard battery of tests of randomness. *https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/*, 1995. Accessed: 2021-11-04.

[15] J. Massey. Shift-register synthesis and bch decoding. *IEEE Transactions on Information Theory*, 15(1):122–127, 1969.

[16] S. Newcomb. Note on the frequency of use of the different digits in natural numbers. *American Journal of Mathematics*, 4(1):39–40, 1881.

[17] H. Okada, K. Umeno. Randomness evaluation with the discrete fourier transform test based on exact analysis of the reference distribution. *IEEE Transactions on Information Forensics and Security*, PP, 01 2017.

[18] D. Panchenko. Chi-squared goodness-of-fit test. *https://ocw.mit.edu/courses/mathematics/18-443-statistics-for-applications-fall-2006/lecture-notes/lecture11.pdf*, 2006. Accessed: 2021-11-07.

[19] N. Privault. Maximum of brownian motion. *https://personal.ntu.edu.sg/nprivault/MA5182/maximum-brownian-motion.pdf*, 2021. Accessed: 2021-11-07.

[20] R. A. Rueppel. Linear complexity and random sequences. F. Pichler, redaktor, *Advances in Cryptology — EUROCRYPT' 85*, strony 167–188, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[21] M. F. Schilling. The longest run of heads. *The College Mathematics Journal*, 21(3):196–207, 1990.

[22] F. Sulak. New statistical randomness tests: 4-bit template matching tests. *Turkish Journal of Mathematics*, 41:80–95, 2017.

[23] J. von Neumann. Various techniques used in connection with random digits. A. Householder, G. Forsythe, H. Germond, redaktorzy, *Monte Carlo Method*, strony 36–38. National Bureau of Standards Applied Mathematics Series, 12, Washington, D.C.: U.S. Government Printing Office, 1951.

Zawartość płyty CD

Dodatek ten krótko omówiona zawartość dołączonej do pracy płyty CD. Zawiera ona implementację generatora jak i testów w języku Python. Umieszczono na niej również wszystkie inne pliki wykorzystane w niniejszej pracy.

- /
  - prepared_binary
    - lcm.bin ..... binarna postać wyniku pracy generatora LCM
    - tarus.bin ..... binarna postać wyniku pracy generatora Tarus
    - crng.bin ..... binarna postać wyniku pracy generatora CRNG
    - my.bin ..... binarna postać wyniku pracy naszego generatora
  - prepared_num
    - lcm.txt ..... wynik pracy generatora LCM w postaci 32-bitowych liczb
    - tarus.txt ..... wynik pracy generatora Tarus w postaci 32-bitowych liczb
    - crng.txt ..... wynik pracy generatora CRNG w postaci 32-bitowych liczb
    - my.txt ..... wynik pracy naszego generatora w postaci 32-bitowych liczb
  - tests
    - frequency_test.py ..... kod testu częstotliwości
    - block_frequency_test.py ..... kod testu jedynek w bloku
    - run_test.py ..... kod testu nieprzerwanych ciągów
    - longest_run_in_block_test.py ..... kod testu najdłuższego ciągu
    - fourier_test.py ..... kod testu nieokresowości
    - overlapping_template_test.py ..... kod testu nakładających się wzorców
    - linear_complexity_test.py ..... kod testu złożoności liniowej
    - last_cross_test.py ..... kod testu ostatniego przecięcia
    - max_cumsum_test.py ..... kod testu największej sumy
    - matrix_rank_test.py ..... kod testu rang macierzy
    - utils.py ..... kod metod pomocniczych
    - __init__.py ..... kod modułu tests
  - addresses.txt ..... lista najpopularniejszych stron www
  - preprocesor.py ..... funkcje pomocnicze do zapisu lub odczytu danych z pliku
  - generator.py ..... kod zaprezentowanego generatora
  - test_generators.py ..... kod generatorów LCM, Tarus i CRNG
  - testing.py ..... kod programu testującego