KTH Computer Science
and Communication

Pseudoslumptalsgenerering

En studie av pseudoslumpsalgoritmer i allmänhet med fokus på C++11-biblioteket *Random* samt implementation av en modifierad *XORShift*

HAMPUS LILJEKVIST

MICHAEL HÅKANSSON

DD143X Examensarbete inom datalogi, grundnivå
Handledare: Per Austrin
Examinator: Örjan Ekeberg

CSC, KTH 2014-04-28

Referat

Slumpmässighet är ett komplext område inom datalogin, där endast pseudoslumptal går att producera algoritmiskt. Denna rapport presenterar etablerade algoritmer för att generera pseudoslumptal genom att utföra en grundläggande litteraturstudie, samt testa fördelning och tidseffektivitet hos algoritmerna i C++11:s Random-bibliotek. Resultaten jämförs med algoritmen XORShift och en modifierad version av denna. Studien visar att den modifierade algoritmen klarar alla ingående test i Small Crush, samt att körtiden är kortare än för algoritmerna inbyggda i Random-biblioteket. Slutsatsen är att det med med enkla medel är möjligt att modifiera en algoritm till att överträffa algoritmerna i C++11:s Random-bibliotek.

Abstract

Pseudo random number generation

Randomness is a complex area of computer science, where only pseudo random numbers are possible to generate algorithmically. This report presents established algorithms for random number generation by conducting a basic study of relevant literature, while testing distribution and speed of the algorithms contained within the C++11 Random library. The results are compared to the algorithm XOR-Shift and a modified version of it. The study shows that the modified algorithm passes all tests in Small Crush, and that the run time is shorter than for those in the Random library. The conclusion is that it is possible to, with simple means, modify an algorithm to outperform the algorithms of the C++11 Random library.

Samarbetsredogörelse

Arbetet med rapporten och studien har som helhet varit jämnt fördelat mellan författarna. De olika kapitlen har skrivits och korrigerats till största del gemensamt, även om vissa stycken har skrivits mer självständigt. Michael Håkansson har lagt mer tid på rapport och formatering medan Hampus Liljekvist har spenderat större tid på programmeringsaspekten av projektet.

Innehåll

1 Introduktion **1**
1.1 Syfte 1
1.2 Frågeställning 1

2 Bakgrund **3**
2.1 Historia 3
2.2 Definition av slumptal 4
2.3 Kategorisering av generatorer 4
2.4 Metoder för att försöka generera äkta slumptal 5
2.4.1 Manuell observation samt radioaktivt sönderfall 5
2.4.2 Webbkameror och mikrofoner 5
2.5 Bedömning av generatorer och slumptal 6
2.5.1 Fördelning och slumpmässighetsgrad 6
2.5.2 Period 6
2.5.3 Tidseffektivitet 6
2.6 Vanliga algoritmiska metoder för att generera pseudoslumptal 7
2.6.1 Linjär kongruens 7
2.6.2 Icke-linjär kongruens 8
2.6.3 Skiftregister 8
2.6.4 Kombinerade generatorer 8
2.7 Inbyggda algoritmer i C++11 9
2.7.1 Minimal standard 9
2.7.2 Mersenne Twister 10
2.7.3 Subtract with carry 11
2.8 XORShift 13
2.9 Testmetoder 13
2.9.1 Chi-två-test 13
2.9.2 Runs test 14
2.9.3 DIEHARD 14
2.9.4 dieharder 15
2.9.5 NIST 15
2.9.6 TestU01 15
2.9.7 Gap test 16

3 Metod **17**
3.1 Litteraturstudie 17
3.2 Algoritmkonstruktion 17
3.3 Testmetod 18
3.3.1 Förutsättningar 18
3.3.2 Utförda test 18

4 Resultat **21**
4.1 Testresultat 21
4.1.1 Utfall från testning av inbyggda funktioner i C++11 21
4.1.2 Utfall från testning av XORShift samt modifierad XORShift 22
4.2 Jämförelse av algoritmer i C++11 22
4.3 Algoritmkonstruktion 23

5 Diskussion och slutsats **25**
5.1 Diskussion 25
5.2 Slutsats 26

Bilagor **26**

A Implementation av XORShift i C++ **27**

B Modifierad XORShift i C++ **29**

Litteraturförteckning **31**

Kapitel 1

Introduktion

Slumpmässighet kan te sig självklar i naturen och vår vardag, men inom datalogin är slumpen ett problematiskt område. Datorer är i sin konstruktion deterministiska, och att få dem att producera slumpmässiga tal med en rimlig fördelning är en fin balans mellan antal processorcykler och kvalitet. Att med hjälp av en dator generera slumpmässiga talsekvenser är därför inte lika lätt som det kan låta. Dessutom ställs allt högre krav på pseudoslumptalsgenerering, framförallt inom kryptografi. Därför är det både intressant och relevant att fördjupa sig i de tekniker som används för att med en dator ta fram vad som ska verka vara en slumpmässig talsekvens.

1.1 Syfte

Denna studie har haft som syfte att undersöka befintliga algoritmer för pseudoslumptalsgenerering, där en fördjupning har gjorts i de algoritmer som finns inbyggda i programmeringsspråket C++11:s Random-bibliotek. En modifierad variant av XORShift har implementerats och jämförts med de inbyggda algoritmerna i Random-biblioteket med avseende på kvalitet och tid.

1.2 Frågeställning

Hur står sig en med enkla medel egenmodifierad variant av algoritmen XORShift fördelnings- och tidsmässigt mot de pseudoslumptalsgenereringsalgoritmer som finns i C++11:s Random-bibliotek?

Kapitel 2

Bakgrund

2.1 Historia

Långt innan datorerna uppfanns var slumpen en naturlig del av till exempel spel och dobbel. Slantsingling och tärningskast har varit, och är fortfarande, vanliga medel för slumpmässiga utfall. När forskare och vetenskapsmän förr i tiden ville ha tillgång till slumptal var de begränsade till att generera slumptal manuellt, exempelvis genom att just singla slant, kasta tärning eller dra kort [6].

År 1927 publicerades en tabell med 40 000 slumptal tagna från olika rapporter, men det dröjde till 1939 innan en mekanisk slumptalsmaskin realiserades. Inte förrän 1951 utvecklades en dator som genererade slumpade bitar, då med hjälp av elektroniskt brus. En klassisk slumptalsgenerator gick under namnet *ERNIE*. Den användes länge för att dra vinnande lottorader i ett brittiskt lotteri [6].

När väl datorna kommit dröjde inte länge efter deras introduktion innan människor började använda dem för pseudoslumptalsgenerering. Även om tabellerna som tidigare har nämnts var brukbara, krävde de mycket minne och tidsåtgång, och dedikerade enheter likt ERNIE var opålitliga. Därför var slumpalgoritmer eftertraktade, och John von Neumann föreslog 1946 följande metod [6]:

1. Ta föregående pseudoslumptal med längd n.
2. Kvadrera det.
3. Tag de n mittersta siffrorna av resultatet som det nya pseudoslumptalet.

Algoritm 1: Middle square

Denna metod har den uppenbara bristen att talen ej är slumpmässiga, men de framstår som sådana. I praktiken spelar det inte någon roll hur talen är relaterade, även om denna metod visat sig jämförelsevis dålig [7, sid.2-3].

Sedan den tiden har slumpalgoritmer varit ett ämne som det har forskats flitigt inom. Idag finns det inbyggda funktioner för pseudoslumptalsgenerering i alla välutvecklade programmeringsspråk. Detta eftersom slumptal är något som används

ofta. De används till exempel i vissa typer av algoritmer, i synnerhet Monte Carlo-algoritmer, som är beroende av slumptal för att vara effektiva [6, sid.2] [7, sid.2]. Slumptal används även inom spel och kryptering [6, sid.2]. En anledning till att det har forskats mycket inom området, förutom att användningsområdet är brett, är att det har visat sig vara svårt att generera högkvalitativa pseudoslumptal.

2.2 Definition av slumptal

National Institute of Standards and Technology (NIST) använder sig av exemplet med slantsinglning för att definiera en slumpässig bitsekvens. Givet ett rättvist mynt med sidorna 1 och 0 kommer sannolikheten för 1 respektive 0 vara $\frac{1}{2}$. Dessutom är upprepade singlingar helt oberoende av varandra [17, sid.1-1].

Inom kryptografin ställs det särskilda krav på vilka talsekvenser som kan betraktas som slumpmässiga. Sannolikheten för ett enskilt genererat tal får inte vara beroende av föregående tal. Det betyder att sekvensen inte ska ge ökad sannolikhet att nästkommande tal ska ha ett visst värde. I kryptografiska sammanhang brukar det krävas att det inte får finnas någon algoritm som kan gissa nästa tal i sekvensen i polynomisk tid. Förutsägbarheten är central för slumpmässighet inom kryprografiska sammanhang [5, sid.3].

Knuth pekar på svårigheten med att resonera kring ett “slumpmässigt tal”. Det går inte att betrakta ett enskilt tal som slumpmässigt, utan det talas istället om slumpmässiga sekvenser av tal. Dessa har någon fördelning, där varje enskilt tal är oberoende av övriga i den genererade sekvensen [7, sid.2].

En äkta slumpmässig sekvens kommer innefatta lokal “icke-slumpmässighet”. Förr eller senare uppstår en sekvens som vid isolerat betraktande inte skulle klassificeras som slumpmässig. Detta är både absolut nödvändigt och potentiellt skadligt beroende på sammanhanget, och ingen sekvens av slumptal passar alla ändamål [7, sid.152].

2.3 Kategorisering av generatorer

Slumptalsgeneratorer kan delas upp i två huvudsakliga kategorier. Det finns dels pseudoslumptalsgeneratorer (*pseudo random number generators* – PRNGs) och äkta slumptalsgeneratorer (*true random number generators* – TRNGs). Pseudoslumptal är de tal som tas fram med deterministiska metoder, det vill säga att resultatet för genereringen är förutbestämt. Med andra ord, om generatorn startas med indata x kommer alltid utdata y att genereras [6, sid.5]. Ett typiskt sådant sätt att generera pseudoslumptal är med någon algoritm som körs på en dator, utan inblandning av ytterligare utrustning. För att generera äkta slumptal går det alltså inte enbart att förlita sig på en algoritm av detta slag, utan det behövs inblandning av yttre element för att kunna generera äkta slump. Mer om detta i nästa avsnitt.

2.4 Metoder för att försöka generera äkta slumptal

För att sätta algoritmisk pseudoslumptalsgenerering i perspektiv redogörs här för metoder att med andra medel generera slumptal.

Att på ett effektivt sätt uppnå äkta slump med hjälp av datorer är något som eftersträvas. Som nämnts tidigare går det inte enbart att använda sig av en algoritm, utan det behövs inblandning av andra element, vilket ofta är fysiskaliska företeelser. Värt att komma ihåg är att TRNGs som baseras på fysikaliska företeelser ändå kan vara förutsägbara. Exempelvis kan eletroniskt brus vara styrt av variationer på reguljära strukturer som vågor. Det medför att en RNG baserad på något tillsynes slumpmässigt naturligt fenomen ändå kan vara förutsägbar. Jämförelsevis framstår PRNGs många gånger i praktiken som mer slumpmässiga än TRNGs. Varje nytt tal kommer vara beroende på transformationer av de föregående, där dessa transformationer verkar introducera mer slumpmässighet. Vissa PRNGs kan vara bättre än vissa TRNGs med avseende på statistiska egenskaper och snabbhet. Vid användning av en PRNG bör ändå startvärdet vara genererat från en "äkta" slumpkälla [17, sid.1-2].

2.4.1 Manuell observation samt radioaktivt sönderfall

Skolexemplen inom sannolikhetsteori när det kommer till slump är slantsingling och tärningskast med mynt respektive tärningar som är helt symmetriska. Slantsingling och tärningskast är inte genomförbart utan påverkan från yttre faktorer, inte heller särkilt effektivt. Dessutom är det opraktiskt att kombinera och automatisera med en dator. Istället brukar fysikaliska processer användas. Tanken är att dessa fysikaliska processer ska kunna mätas utan direkt inblandning av en människa. En vanlig metod har då historiskt sett varit att använda sig utav radioaktivt sönderfall. Från det radioaktiva sönderfallet går det att utläsa slumpmässiga tal, som i sin tur kan användas som indata till någon algoritm. Denna algoritm ska då sedan arbeta fram ett slumptal inom ett önskat intervall [6, sid.3]. I algoritmen gäller det att vara försiktig med hur datan behandlas, för att få en så jämn fördelning som möjligt. Vissa operationer kan nämligen göra att fördelningen blir skev. Exempelvis kan det, om moduloberäkning används för att få talet inom det önskade intervallet, leda till att sannolikheten är större för låga tal är än för höga.

2.4.2 Webbkameror och mikrofoner

På senare tid har forskning bedrivits för att finna sätt att med hjälp av utrustning som ofta finns tillgänglig för privatanvändaren kunna generera äkta slumptal. Förhoppningen är att det inte ska behövas avancerad mätutrustning och radioaktiva material för att relativt effektivt kunna generera dessa. Studier har bland annat gjorts kring användandet av mikrofoner och webbkameror som slumpelement. Tsai, Chen och Tzeng har gjort en studie där de använder sig av en webbkamera för att

ta en bild och med hjälp av denna bild generera indata till sin algoritm. De kom dock fram till att, åtminstone med den algoritm de använde sig av, de webbkameror som de använde sig av inte var bra nog för att kunna användas till att generera äkta slumptal [18]. Samma forskargrupp har även analyserat användandet av mikrofoner för att generera slumptal. I den studien använde de sig av mikrofonen från ett enkelt headset. I sina försök lyckades de kombinera användandet av mikrofon och en filtreringsalgoritm så att 99% av de genererade sekvenserna passerade de test (*NIST SP 800-22rev1a 15 statistical tests* – se avsnitt 2.9 för mer information om olika testmetoder) som utfördes [2].

2.5 Bedömning av generatorer och slumptal

Generatorer och slumptal kan bedömas på olika sätt, beroende på vilka egenskaper som eftersöks hos generatorn och de tal som genereras. Nedan följer en redogörelse av detta.

2.5.1 Fördelning och slumpmässighetsgrad

För att slumptal ska vara användbara är det viktigt att fördelningen är känd, och de enskilda talen måste vara samma fördelning utan beroenden mellan dem. Även om många fysiska processer kan användas som källor för slumptal är det ändå ett problem att avgöra till vilken fördelning talen hör [5, sid.2]. Inom kryptografin finns det särskilda krav på slumptalen som används. Det ska inte vara möjligt att i polynomisk tid gissa nästa tal i sekvensen. Därför utgör förutsägbarhet en central del av definitionen av slumptal i kryptografiska sammanhang [5, sid.3]. NIST anger dessutom kravet att det ej ska vara möjligt att beräkna vilket startvärde som användes för algoritmen [17, sid.1-1].

2.5.2 Period

En faktor att ta hänsyn till för PRNGs är deras period. PRNGs har en ändlig period, vilket innebär att talen för eller senare kommer att upprepa sig. Inom vissa användningsområden måste algoritmerna ha en riktigt lång period för att vara användbara [6, sid.4]. Jämför linjärkongruensgeneratorer, som typiskt har en period på runt 2^{32} [6, sid.5], med *Mersenne Twister*, vars period är $2^{19937} - 1$ [13, sid.4].

2.5.3 Tidseffektivitet

När användbarheten hos en algoritm för slumptal bedöms är tiden det tar att producera nya tal relevant. Metoder som används för att producera äkta slumptal är generellt sett långsamma, vilket gör dem mindre lämpade för exempelvis simuleringar som kräver stora mängder slumptal. Då är det mer lämpligt med PRNGs som kan generera talsekvenser på kort tid. I vetenskapliga och statistiska sammanhang utan kritiska krav på äkta slumpmässighet föredras därför ofta PRNGs [6, sid.3-4].

2.6 Vanliga algoritmiska metoder för att generera pseudoslumptal

Det finns en uppsättning av ofta använda metoder för att generera pseudoslumptal. Nedan presenteras några av dessa, där de flesta finns representerade i någon av algoritmerna i C++11 som beskrivs ytterligare i avsnitt 2.7.

Empirisk forskning har visat att det är möjligt att åstadkomma bra PRNGs genom att förse dem med riktigt långa perioder, god struktur på de möjliga talföljderna och inte för okomplicerade upprepningar av tal. Dessa passerar de flesta test, medan sämre algoritmer inte gör det. Dessvärre är det ofta de egenskaper som gör en algoritm enkel att implementera som också gör den dålig. L'Ecuyer och Simard avråder dock från att försök till förbättringar av dåliga algoritmer görs genom att slänga "dåliga sekvenser". Detta kan nämligen göra fördelningen skev [8, sid.2-3].

2.6.1 Linjär kongruens

En vanlig metod för att generera pseudoslumptal är att använda sig av en *linjär kongruensgenerator*. En sådan bygger på en simpel linjär funktion modulo något tal m enligt algoritm 2

$$x_i \equiv (ax_{i-1} + c) \pmod{m}, \quad 0 \leq x_i < m,$$

Algoritm 2: Linjär kongruens

där c ofta är 0 och gör det till en enbart multiplikativ generator. Den här typen av generatorer är dock inte särskilt användbara på egen hand, då de inte kan generera särskilt långa talsekvenser som verkar oberoende. Däremot är de inte sällan en byggsten i mer välfungerande PRNGs [5, sid.11].

Det som är tänkt att göra att talen verkar slumpmässiga i den här typen av generatorer är moduloberäkningen. Knuth jämför moduloberäkningen med att försöka förutse vart en boll kommer att landa på ett snurrande roulettehjul [7, sid.10]. Valet av talen a, c och m är viktig för att få en sekvens som verkar så slumpmässig och har så lång period som möjligt. Enligt Gentle bör en enkel PRNG ha en period på minst 10^9, och för att ha någon möjlighet att ha en sådan lång period måste givetvis $m \geq 10^9$ [5, sid.12-13]. För mer detaljerad information kring valet av parametrar hänvisas till Knuth [7, kap.3.2.1].

Linjära kongruensgeneratorer har vissa brister. Utöver att ha en begränsad period uppvisar talen som genereras en tydlig reguljär struktur, vilket enkelt kan ses vid betraktande av algebran bakom algoritmen [5, sid.14-15].

2.6.2 Icke-linjär kongruens

Som ett alternativ till linjära kongruensgeneratorer förelsog Eichenauer och Lehn rekursionen

$$x_i \equiv (ax_{i-1}^{-1} + c) \pmod{m}, \quad 0 \leq x_i < m,$$

Algoritm 3: En typ av icke-linär kongruensgenerator

där x^{-1} är den multiplikativa inversen till $x \pmod{m}$ om den existerar, annars 0. Tillsammans med andra har de visat att icke-linjära kongruensgeneratorer har god likformig fördelning. Senare presenterade Eichenauer-Herrmann och Ickstadt en annan variant, men idag har dessa metoder för pseudoslumptal inte fått något större genomslag. Detta beror på hög beräkningskostnad och varierande testresultat [5, sid.36-37].

Fler typer av icke-linjära generatorer har tagits fram, som exempelvis algoritm 4 som har utvecklats av Knuth [5, sid.37].

$$x_i \equiv (dx_{i-1}^2 + ax_{i-1} + c) \pmod{m}, \quad 0 \leq x_i < m.$$

Algoritm 4: En variant av icke-linär kongruensgenerator framtagen av Knuth.

2.6.3 Skiftregister

För att generera pseudoslumptal föreslog Tausworthe år 1965 en kongruensgenerator likt algoritm 5,

$$b_i \equiv (a_p b_{i-p} + a_{p-1} b_{i-p+1} + \ldots + a_1 b_{i-1}) \pmod{2},$$

Algoritm 5: Skiftregistergenerator

där alla variabler antingen är 0 eller 1 [5, sid.38]. Denna beräkning kan göras i ett "feedback shift register", där bitarna förskjuts i en vektor åt exempelvis vänster, en bit i taget. Sedan kombineras den utskjutna biten med andra bitar i registret för att skapa biten längst till höger. En generaliserad sådan (*Generalized Feedback Shift Register* – GFSR) algoritm använder bland annat en fördröjning när den placerar ut bitarna [5, sid.40]. En algoritm som använder sig av en typ av GFSR är *Mersenne Twister* som finns inbyggd i bland annat C++11 (se avsnitt 2.7.2).

En annan skiftregisteralgoritm är *XORShift*. Den algoritmen beskrivs i avsnitt 2.8.

2.6.4 Kombinerade generatorer

I syfte att åstadkomma en mer oberäknelig sekvens kan PRNGs kombineras, vilket kan göras på olika sätt. Marsaglia och MacLaren presenterade i en artikel från 1965 hur de hade testat att kombinera två linjära kongruensgeneratorer, och på så sätt

skapa en generator som är bättre än båda de ingående generatorerna. Nackdelen de såg var att kombineringen gjorde genereringen långsammare. Närmare bestämt tog det ungefär dubbelt så lång tid jämfört med en enkel linjär kongruensgenerator. De föreslog att kombinationen skulle göras genom att först generera en sekvens med en linjär kongruensgenerator, och sedan blanda om sekvensen med hjälp av en andra kongruensgenerator. De två generatorerna som används bör ha olika egenskaper för att i sådan stor utsträckning som möjligt undvika regelbundenhet i den genererade sekvensen [10].

Ett annat sätt att kombinera generatorer på föreslogs år 1987 av Collings. Det gick ut på att ha en uppsättning av generatorer, och sedan använda en annan generator för att välja ut vilken av de i uppsättningen som skulle användas [3].

Även om det är troligt att en kombination av två generatorer gör sekvenserna mer oberäkneliga, och därmed också slumpmässiga, är det inte nödvändigtvis alltid fallet. En generator har ett ändligt antal interna tillstånd, vilket kan leda till att en kombination med en annan generator förstärker bådas dåliga egenskaper [5, sid.48].

2.7 Inbyggda algoritmer i C++11

I denna studie har det fokuserats på de tre algoritmer som finns i olika varianter i Random-biblioteket i C++11 samt XORShift. Vilka algoritmer det är, och en beskrivning av dessa, följer i detta avsnitt. XORShift behandlas i avsnitt 2.8.

Det finns en stor mängd algoritmer för generering av slumptal, och det skapas hela tiden nya. I programmeringsspråket C++11 finns det ett Random-bibliotek, vilket innehåller en uppsättning av olika varianter av populära algoritmer för pseudoslumptalsgenerering. Utöver dessa algoritmer finns det även funktioner för att generera pseudoslumptal enligt sannolikheten för någon fördelning, exempelvis likformig fördelning och normalfördelning. Det finns i biblioteket också stöd för att med hjälp av hårdvara generera äkta slumptal, något som har beskrivits översiktligt i avsnitt 2.4 [4].

2.7.1 Minimal standard

Namn i C++11	Skapare av algoritm	Alg. publicerad
minstd_rand0	Park, Miller	1988
minstd_rand	Park, Miller, Stockmeyer	1993

Tabell 2.1: Varianter av minimal standard i C++11

År 1988 publicerade Stephen K. Park och Keith W. Miller en artikel [15] där de rekommenderade en PRNG, som enligt dem borde användas som måttstock för andra PRNGs. Denna algoritm fick namnet Minimal standard. Algoritmen uppfanns egentligen 1969, men formaliserades och namnsattes inte förens 1988. Författarna föreslog i artikeln en implementation i Pascal enligt följande:

```
function Random : real;
    (* Integer Version 1 *)
const
  a = 16807;
  m = 2147483647; // = 2^31 - 1
begin
  seed := (a * seed) mod m;
  Random := seed / m
end ;

where

  var seed : integer;
```

Algoritm 6: Minimal standard algoritmen implementerad i Pascal av Park och Miller

Detta är grunden för den algoritm som heter minstd_rand0 i C++11. Fem år senare, 1993, föreslog Stephen K. Park, Keith W och Paul K. Stockmeyer i en korrespondens [16] att $a = 48271$ var ännu bättre, vilket minstd_rand baseras på.

2.7.2 Mersenne Twister

Namn i C++11	Skapare av algoritm	Alg. publicerad
mt19937	Matsumoto, Nishimura	1998
mt19937_64	Matsumoto, Nishimura	1998

Tabell 2.2: Varianter av Mersenne Twister i C++11

En algoritm för pseudoslumptal föreslogs år 1998 av Makoto Matsumoto och Takuji Nishimura, under namnet *Mersenne Twister* [13, sid.4]. Algoritmen som Mersenne Twister baseras på är av typen GSFR (se avsnitt 2.6.3), nämligen Twisted GSFR som publicerades 1992 [13, sid.6]. Den "vrider om" bitmönstret genom att manipulera vektorer av nollor och multiplicera med matriser [5, sid.42].

Mersenne Twister får sitt namn från att dess period är ett Mersenne-primtal. Perioden i fråga är $2^{19937} - 1$, vilket är betydligt mycket högre än hos många andra PRNGs. Algoritmen har dessutom en så kallad "ekvidistributionsgrad", vilket förklaras senare, på 623. Trots allt detta mäter den sig tidsmässigt med algoritmer vars period är under 2^{130}, och klarar *DIEHARD*-testet som behandlas i avsnitt 2.9.3 [13, sid.6].

Ekvidistributionsgraden ger ett mått på graden av slumpmässighet för talen genererade av en algoritm. Följande definition ges av Makoto och Takuji [13, sid.4-5]:

En pseudoslumpmässig sekvens x_i av w-bitars heltal med period P, som uppfyller följande krav, sägs vara k-fördelad med v-bitars noggrannhet: låt $trunc_v(x)$ vara

talet skapat av de v ledande bitarna av x och betrakta P av kv-bitarsvektorerna:

$$(trunc_v(x_i), trunc_v(x_{i+1}), \ldots, trunc_v(x_{i+k-1})(0 \leq i \leq P)).$$

Då förekommer de 2^{kv} möjliga kombinationerna av bitar samma antal gånger i en period, förutom 0-kombinationen som förekommer en gång mindre. För varje $v = 1, 2, \ldots, w$, låt $k(v)$ utgöra det största tal sådant att sekvensen är $k(v)$-fördelad upp till v bitars noggrannhet.

Mersenne Twiser får med $v = 32$ värdet $k = 623$, vilket vid rapporterns publiceringstillfälle verkade vara den bästa någonsin [13, sid.4].

Det som skiljer Mersenne Twister från Twisted GFSR-algoritmen är först och främst en “ofullständig vektor”, då den saknar ett antal bitar i det övre högra hörnet. På denna vektor utförs linjärtransformationer som gör att den antar alla värden i perioden förutom 0-tillståndet [13, sid.12]. Den andra stora förbättringen är en snabb metod för att testa primitiviteten hos det karakteristiska polynomet för en linjär differensekvation [13, sid.4].

I kryptografiska sammanhang krävs det att de genererade talen hashas med någon säker hashfunktion, då det är möjligt att förutspå tal efter en enkel linjärtransformation [13, sid.7]. Alltså är inte Mersenne Twister en algoritm som bör användas inom kryptografi.

Idag finns Mersenne Twister som standard i många programmeringspråk, som exempelvis C++11, Python och Matlab [6, sid.8].

2.7.3 Subtract with carry

Namn i C++11	**Skapare av algoritm**	**Alg. publicerad**
ranlux24_base	Marsaglia & Zaman	1991
ranlux48_base (64 bitar)	Marsaglia & Zaman	1991
ranlux24	Lüscher & James	1994
ranlux24	Lüscher & James	1994

Tabell 2.3: Varianter av subtract with carry i C++11

Subtract-with-carry är en algoritm som baserar sig på George Marsaglia och Arif Zamans arbete som publicerades i The Annals of Applied Probability 1991 [12, sid.462]. Det speciella med denna algoritm var att den kunde generera talsekvenser med lång perioditet. De körde algoritmerna mot DIEHARD-testet (se avsnitt 2.9) och klarade då alla ingående test [12, sid.478].

En “förskjuten Fibonacci-sekvens” fås typiskt genom att ta Fibonacci-serien modulo något tal, eventuellt med konstanter som förskjuter talen i rekursionen (lags). Se algoritm 7 för fyra olika varianter av Fibonacci-generatorer. För att få en längre period går det att lägga till en carry-bit till rekursionen, som sätts till 1 eller 0 beroende på om summan överstiger modulo-talet eller ej [12, sid.464]. Genom att konfigurera parametrarna till rekursionen är det möjligt att åstadkomma perioder

på upp till 2640. Marsaglia och Zaman kallar denna generatortyp för *add-with-carry* [12, sid.465].

1. $x_n = x_{n-s} - x_{n-r} - c \pmod b$
2. $x_n = x_{n-r} - x_{n-s} - c \pmod b$
3. $x_n = x_{n-r} + x_{n-s} - c \pmod b$
4. $x_n = b - 1 - x_{n-r} - x_{n-s} - c \pmod b$

där b är basen, r och s är förskjutningar. [12, sid.478]

Algoritm 7: Fyra varianter av förskjutna Fibonacci-generatorer föreslagna av Marsaglia och Zaman.

En alternativ klass av generatorer använder sig av differenser för sekvensen och subtraherar carry-biten. Om resultatet blir positivt väljs det som nästa element med carry 0, annars adderas modulo-talet och carry-biten sätts till 1. Denna variant av generatorn går under namnet *subtract-with-borrow* [12, sid.466]. Subtract-with-carry i C++11 är av den senare typen [14].

Martin Lüscher hävdade att dessa algoritmer misslyckas i många test för slumptal, som exempelvis *gap test*, då det finns många uppenbara samband mellan påföljande tal i serien [9, sid.103]. Lüscher framför ändå i sin rapport att algoritmerna utgör ett deterministiskt kaosartat system, där beroenden mellan tal blir exponentiellt mindre då de observeras över en längre tidsperiod [9, sid.104].

Algoritmen förbättras genom följande algoritm [9, sid.106]:

1. Börja med en serie pseudoslumptal skapade från Marsaglia-Zalmans algoritm med carry-bitar och rimliga startvärden.
2. Läs r tal från serien.
3. Släng de $p - r$ efterföljande talen, där p är en konstant parameter $\geq r$.
4. Gå till (2).

Algoritm 8: Lüschers förbättrade variant av Marsaglia-Zamans förskjutna Fibonacci-generator

Lüscher rekommenderar $p = 223$ som standard, alternativt $p = 389$ för att säkerställa att PRNGs inte ger ett partiskt resultat [9, sid.108]. Den förbättrade algoritmen har med ett p-värde på över 384 utmärkta statistiska egenskaper [9, sid.107].

2.8 XORShift

I denna rapport presenteras i avsnitt 3.2 en modifiering av den befintliga algoritmen *XORShift*, som utvecklades av George Marsaglia och presenterades år 2003. XORShift bygger på utförande av exklusiv disjunktion (XOR) för ett tal med en bitskiftad version av talet självt [11]. Algoritmen är en typ av skiftregisteralgoritm, vilka behandlades kort i avsnitt 2.6.3. Eftersom både XOR och bitskiftning är snabba operationer, blir XORShift en snabb algoritm, eller som Marsaglia valde att uttrycka det: "extremely fast" [11].

En implementation av XORShift som presenterades av Marsaglia, och som det kommer att fokuseras vidare på i denna rapport är den i algoritm 9.

// $a \odot b$ definieras som a bitskiftat b steg åt höger
// $a \otimes b$ definieras som a bitskiftat b steg åt vänster
// $a \oplus b$ definieras som a xor med b

$x \longleftarrow 123456789$
$y \longleftarrow 362436069$
$z \longleftarrow 521288629$
$w \longleftarrow 88675123$

$t \longleftarrow (x \oplus (x \otimes 11))$
$x \longleftarrow y$
$y \longleftarrow z$
$z \longleftarrow w$
$w \longleftarrow (w \oplus (w \odot 19)) \oplus (t \oplus (t \odot 8))$

return w

[11, sid.5]
Algoritm 9: XORShift implementerad av Marsaglia

2.9 Testmetoder

För att testa hur bra en slumptalsgenerator är i förhållande till de faktorer som presenterades i avsnitt 2.5 finns det en mängd olika test och färdiga testpaket. I det här avsnittet presenteras information om några av de vanligaste testen och testpaketen som användes som underlag för val av testmetod.

2.9.1 Chi-två-test

För att testa algoritmer kan en nollhypotes ställas upp, och sedan se om den går att förkasta. I detta sammanhang är ett positivt resultat att nollhypotesen inte förkastas. När slumpgeneratorer testas statistiskt är hypotesen antingen att talen

är likformigt fördelade enligt U(0, 1), eller att de är Bernoullifördelade med $\frac{1}{2}$ som parameter. Alternativt att de inte har den fördelningen [5, sid.71]. Ett test av denna typ är *chi-två-test*, där utfallen av algoritmen jämförs med det enligt hypotesen förväntade utfallet. Skulle hypotesen vara likformig (0, 1) fördelning och det görs 100 observationer kan 10 tal i varje $\frac{1}{10}$-intervall förväntas. Visar det sig att så inte var fallet bör nollhypotesen förkastas [5, sid.74].

$$\chi_c^2 = \sum_{i=1}^{k} \frac{(o_i - e_i)^2}{e_i}$$

k = antal intervall, o och e är observerade och förväntade tal i det i:te intervallet.

Chi-variablen har chi-två-fördelning med $k - 1$ frihetsgrader. Blir variabeln stor bevisar det att de observerade värdena markant skiljer sig från de förväntade, och hypotesen om fördelning är således felaktig [5, sid.74].

2.9.2 Runs test

En effektiv dynamisk testmetod som mäter antal (med eller utan längd) successiva körningar av algoritmen där talen antingen ökar eller minskar heter *runs test*. Ett exempel är serien 1 5 6 9 2 4 5, där 1 5 6 9 ökar, 9 2 minskar och 2 4 5 ökar. Det finns flera sätt att mäta resultatet på, ett alternativ är att räkna längderna av dessa serier, ett annat är att räkna brytpunkterna där en serie slutar öka eller minska. Då är längderna antal nummer mellan dessa brytpunkter. Med samma exempel blir den ökade serien 1 5 6 9 | 2 4 5. För att tolka resultatet skapas en så kallad “kovariansmatris” av antalet körningar för en viss längd, där resultatet är den kvadratiska formen av förekomsterna i matrisen. Dessa har en chi-två-fördelning [5, sid.77-78].

2.9.3 DIEHARD

DIEHARD är en testsvit som består av 18 deltest utförd på heltal, typiskt i intervallet $(0, 2^{31} - 1)$ med tänkt likformig fördelning. Till skillnad för vanliga test av denna typ motsvarar ett litet p-värde från chi-två-testen i DIEHARD att resultatet överensstämmer med hypotesen. Dessa test kan antingen peka på att utfallet markant skiljer sig från det förväntade, eller att det är misstänkt nära de teoretiska siffrorna. Vissa av testen jämför det mätta p-värdet med nollhypotesens teoretiska fördelning [5, sid.80]. Testen består bland annat av binär rang av matriser, bitströmmar, 3D-sfärer, överlappande summor och runs test. För att ange indata till DIEHARD skapas en binärfil av 32-bitars unsigned integers som genererats av algoritmen som ska undersökas [5, sid.81-83].

Några av nackdelarna med DIEHARD är dels formatet på indata, men också det faktum att det är 32-bitarstal när många algoritmer producerar 31-bitarstal. Dessutom är provstorlekarna relativt små, och blir avklarade på några sekunder på en relativt modern dator [8, sid.6].

2.9.4 dieharder

dieharder bygger på testsviten DIEHARD, men har förfinad kod i C och GPL-licens. Utöver detta har den tillägg från NISTs STS-svit samt nyutvecklade test. Sviten är byggd för att pressa dåliga till avsevärt dåliga mätvärden. Till skillnad från DIEHARD måste inte indata lagras på fil, utan biblioteket är byggt för att testa algoritmer direkt via interface. De flesta av de ursprungliga testen är omskrivna för att förbättra dem, till exempel med större testdata och möjlighet att själv ange storlek på datan [1].

2.9.5 NIST

Det finns ingen komplett testsvit som kan garantera statistiskt goda egenskaper hos alla algoritmer, men *NIST* har ställt upp en samling statistiska test som de finner relevanta. I rapporten som beskriver dessa test är nollhypotesen att talsekvensen är slumpmässigt fördelad. Den alternativa hypotesen är den motsatta, att talsekvensen ej är slumpmässig. Utöver detta används matematiska metoder för att fastslå ett kritiskt värde som utfallet jämförs med.

Om antagandet som gjorts gällande slumpmässigheten är korrekt, är det en väldigt liten chans att utfallet från testet överstiger det kritiska värdet. Visar det sig att utfallet ändå överstiger detta värde (det osannolika händer), bör den osannolika händelsen inte ske naturligt. Med andra ord är det läge att förkasta nollhypotesen om slumpmässighet, och godta den motsatta [17, sid.1-3].

NIST-sviten innehåller 15 test, bland annat frekvenstest, runs test, linjärkomplexitetstest, serietest, entropitest och random walk-test. Det spelar ingen roll i vilken ordning testen körs, men det rekommenderas att frekvenstestet körs först, då det är det test som lättast avslöjar dåliga algoritmer (att fördelningen inte är likformig). Det test som tar längst tid att genomföra är linjärkomplexitetstestet. Vissa av testen använder sig av standard-normal- eller chi-två-fördelning som referens [17, sid.1-3].

2.9.6 TestU01

TestU01-biblioteket är organiserat i fyra moduler: RNGs, statistiska test, fördefinierade testpaket, samt de vektyg som är kompatibla med hela familjer av generatorer. Alla algoritmer än implementerade i ANSI C [8, sid.3]. Den första modulen innehåller färdiga implementationer av generatorer som ibland kan kombineras, men låter också användaren definiera egna (genom modulen *unif01*). Dock är dessa algoritmer inte tänkt att användas för simuleringar eller liknande syften. Den andra modulen innehåller statistiska test som testar någon av de två nollhypoteserna. Algoritmen som ska testas skickas som en parameter till funktionen [8, sid.4].

Modulen över testpaket har uppsättningar av olika sorters standardtest, specialiserade för att upptäcka en viss typ av svaghet i algoritmerna. Den sista modulen innehåller verktyg baserade på övriga komponenter, för att avgöra vid vilken storlek på indata som algoritmerna slutar klara testen [8, sid.5].

2.9.7 Gap test

Gap test används för att undersöka om en serie tal av längd n är likformigt fördelade. När ett tal x_i faller inom ett visst givet intervall $[a, b]$ undersöks hur många påföljande tal som ej faller inom samma intervall. När ett av dessa senare tal, x_{i+j}, till slut faller inom intervallet, motsvarar det en gap-längd j. Hanteras ändliga serier kan det vara en fördel att sätta en maximal gräns för gap-längden. Efter att ha undersökt de olika längderna jämförs fördelningen med ett chi-två-test [19, sid.216].

Kapitel 3

Metod

3.1 Litteraturstudie

För att få förståelse för området har en grundläggande litteraturstudie utförts. Informationen kommer både från böcker inom området och publicerade rapporter, där originalforskning av författarna bakom några av de berörda algoritmerna har prioriterats. För att få en överblick av ämnet i fråga inleds litteraturstudien med en kort historik, som sedan övergår till definition av slumptal och kategorisering av generatorer.

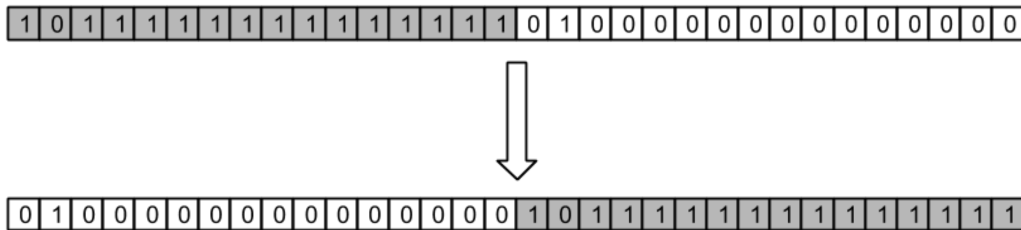
Då frågeställningen berör testning och konstruktion av algoritmer redogör litteraturstudien även för för vanliga metoder för att generera slumptal, samt hur det är möjligt att bedöma deras prestanda. Äkta slumptalsgenerering beskrivs kort för att kontrastera det mot pseudoslumptalsgenering som rapporten fokuserar på.

Litteraturstudien har utförts med särskild fokus på algoritmer i C++11 för att avgränsa ämnet och enkelt kunna utföra test. De utgör ett bra urval av olika vanliga typer av algoritmer, samt passar väl det använda testpaketet som beskrivs i sektion 3.3.2. Utöver detta beskrivs XORShift särskilt ingående då den utgör grunden för den i studien modifierade algoritmen. Det redogörs även för olika testmetoder för att ge förståelse för praktisk bedömning av generatorer.

3.2 Algoritmkonstruktion

Ett av målen initialt var att konstruera en egen PRNG. Under litteraturstudien kom det fram att det är svårt att konstruera PRNGs med bra statistiska egenskaper. Ofta kan det uppstå talsekvenser, som vid observation, ser slumpmässiga ut, men som när de analyseras grundligare uppvisar mönster eller en sned fördelning.

Det som har gjorts är att Marsaglias XORShift-algoritm (algoritm 9 i avsnitt 2.8) har modifierats. Valet att modifiera just XORShift baseras på att denna typ av generator är snabb och enkel att implementera, samtidigt som den presterar bra i test [11, sid.1]. Modifieringen är utförd så att ytterligare komplexitet introduceras till Marsaglias algoritm genom att byta plats på de 16 första och de 16 sista bitarna



Figur 3.1. Byter plats på de första och sista 16 bitarna

innan resultatet returneras. Detta illustreras i figur 3.1. Förhoppningen med denna förändring var att uppnå bättre statistiska egenskaper.

I appendix A finns en implementation av Marsaglias XORShift i C++, och i appendix B finns den modifierade versionen av densamma.

3.3 Testmetod

3.3.1 Förutsättningar

Operativsystem	Ubuntu 13.10 (64 bitar)
Språk	C++11
Kompilator	g++ 4.8.1
Testsvit	TestU01 1.2.3

Tabell 3.1: Mjukvara på testdator

Datormodell	Asus Eee PC 1225B
Processor	AMD E-450 (1.65 GHz)
RAM-minne	4 GB DDR3 (1333 MHz)

Tabell 3.2: Hårdvara i testdator

3.3.2 Utförda test

För att testa och jämföra algoritmer har har testpaketet *bbattery_SmallCrush* från testsviten TestU01 [8] använts. Ett test misslyckas om något p-värde faller utanför intervallet $[0.001, 0.9990]$. Paketet bbattery_SmallCrush innehåller följande test:

BirthdaySpacings En variant på kollisionstestet som sprider ut n punkter över k celler i t dimensioner. Cellerna numreras och t tal genereras, och punkterna betraktas som n födelsedagar under ett år med k dagar. Cellnumren sorteras, och testet

jämför hur många kollisioner som har uppstått när differenserna mellan "födelsedag" j och $j + 1$ beräknas. Fördelningen förväntas vara Poissonfördelad [8, sid.114-115].

Collision Utför ett kollisionstest (antal gånger som en punkt hamnar i en cell som redan innehåller en punkt), istället för att undersöka chi-två-värdet [8, sid.112].

Gap Utför ett "lucktest" där n värden genereras och antal gånger som s successiva värden fallet utanför ett visst intervall räknas. Detta motsvarar antal luckor med längd s mellan besöken av intervallet. Gap-test beskrivs mer detaljerat i avsnitt 2.9.7. Därefter jämförs värdena med de förväntade genom ett chi-två-test vilket beskrivs i avsnitt 2.9.1 [8, sid.111].

SimpPoker En förenklad variant av ett pokertest som genererar n grupper av k heltal i ett intervall. Generatorn anropas $n \cdot k$ gånger, och för varje grupp beräknas mängden distinkta heltal som faller inom deras intervall. Resultatet tolkas med ett chi-två-test [8, sid.111].

CouponCollector Genererar en sekvens med slumpade heltal från 0 till $f - 1$, och undersöker hur många som måste genereras innan de d möjliga värdena förekommer minst en gång. Detta upprepas i sin tur n gånger, och testet jämför detta med de förväntade värdena genom ett chi-två-test [8, sid.111].

MaxOft Genererar n grupper av t värden i intervallet $[0, 1)$ och beräknar deras maxvärde X. Testet beräknar därefter den empiriska fördelningsfunktionen för de n värdena, och jämför med den teoretiska fördelningen genom ett chi-två test [8, sid.112].

MaxOft AD En variant av MaxOft.

WeightDistrib Testet genererar k likformigt fördelade värden och beräknar hur många som faller inom ett visst intervall. Under nollhypotesen är detta antal en binomialfördelad variabel som jämförs med den förväntade fördelningen genom ett chi-två-test. Antal värden i intervallet beräknas n gånger [8, sid.118].

MatrixRank Genererar sekvenser av likformigt fördelade värden och tar ett antal bitar från varje sekvens. En matris fylls radvis med en del av dessa värden. Därefter beräknas matrisrangen, alltså antal linjärt oberoende rader. Den emprisiska fördelningen jämförs med den teoretiska. För jämförelsen används ett chi-två-test [8, sid.115].

HammingIndep Testet producerar n block med L bitar och undersöker proportionen av 1:or i varje icke-överlappande L-block. Antalen med 1:or förväntas vara binomialfördelade variabler, en hypotes som testas med ett chi-två-test [8, sid.128].

RandomWalk1 H *RandomWalk1* uför olika test som baseras på “slumpmässig vandrig” över en mängd heltal, med start i 0. Vid varje steg i vandringen är sannolikheten för ett steg till vänster och höger båda $\frac{1}{2}$. Testet undersöker l-stegsvandringar för alla jämna heltal l i ett intervall. Varje efterföljande vandring lägger till två steg, fram till och med intervallets slut. För vandringarna registreras flertalet testvärden, som sedan undersöks med de teoretiskt förväntade värdena genom ett chi-två-test [8, sid.121].

RandomWalk1 M En variant av RandomWalk1.

RandomWalk1 J En variant av RandomWalk1.

RandomWalk1 R En variant av RandomWalk1.

RandomWalk1 C En variant av RandomWalk1.

Kapitel 4

Resultat

4.1 Testresultat

De angivna CPU-tiderna är på formatet

`minuter:sekunder.hundradelar`

och motsvarar den tid det tog att generera 2^{28} st heltal. Resultaten diskuteras i avsnitt 4.2.

4.1.1 Utfall från testning av inbyggda funktioner i C++11

Test/Algoritm	minstd_rand0	minstd_rand	mt19937	ranlux24_base	ranlux24
BirthdaySpacings	✗	✗	✓	✗	✗
Collision	✗	✗	✓	✗	✗
Gap	✓	✓	✓	✓	✓
SimpPoker	✓	✓	✓	✗	✓
CouponCollector	✓	✓	✓	✓	✓
MaxOft	✗	✗	✓	✗	✗
MaxOft AD	✗	✗	✓	✗	✗
WeightDistrib	✓	✓	✓	✗	✓
MatrixRank	✓	✓	✓	✓	✓
HammingIndep	✓	✓	✓	✓	✓
RandomWalk1 H	✗	✗	✓	✗	✗
RandomWalk1 M	✗	✗	✓	✗	✗
RandomWalk1 J	✗	✗	✓	✗	✗
RandomWalk1 R	✗	✗	✓	✗	✗
RandomWalk1 C	✗	✗	✓	✗	✗
CPU-tid	00:20.34	00:19.71	00:27.15	00:20.92	02:25.67

Tabell 4.1: Resultat av Small Crush-test av inbyggda funktioner i C++11

4.1.2 Utfall från testning av XORShift samt modifierad XORShift

Test/Algoritm	XORShift	Modifierad XORShift
BirthdaySpacings	✓	✓
Collision	✓	✓
Gap	✓	✓
SimpPoker	✓	✓
CouponCollector	✓	✓
MaxOft	✗	✓
MaxOft AD	✓	✓
WeightDistrib	✓	✓
MatrixRank	✓	✓
HammingIndep	✓	✓
RandomWalk1 H	✓	✓
RandomWalk1 M	✓	✓
RandomWalk1 J	✓	✓
RandomWalk1 R	✓	✓
RandomWalk1 C	✓	✓
CPU-tid (gen. 2^{28} heltal)	00:00:09.31	00:00:14.98

Tabell 4.2: Resultat från test av XORShift samt modifierad XORShift av inbyggda funktioner i C++11

4.2 Jämförelse av algoritmer i C++11

Testen utförda med TestU01 gav en indikation av algoritmernas effektivitet och kvalitet. Algoritmerna minstd_rand och minstd_rand0 gav identiska testvärden, vilket är förståeligt då de bygger på samma generatorprincip. De misslyckades med RandomWalk, MaxOft, BirthdaySpacings och Collision och hade en körtid på runt 20 sekunder. Utifrån dessa testresultat finns det ingen anledning att välja den nyare minstd_rand0 över minstd_rand.

Algoritmen mt19937 klarade alla utförda test med ungefär 35% längre körtid än varianterna av minstd_rand. Algoritmen är specifikt konstruerad för att förbättra de testpunkter där andra GFSR-algoritmer ofta misslyckas, vilket gör det rimligt att algoritmen presterar bra i testen.

Den algoritm som till testen sett presterade sämst var ranlux24_base. Den lyckades endast med Gap, CouponCollector, MatrixRank och HammingIndep, med en körtid strax över minstd_rand. Generatorn ranlux24 använder sig av ranlux24_base men bygger upp en buffer och förkastar en del av de producerade talen. Detta gör att ranlux24 klarar två test till, nämligen SimpPoker och WeightDistrib, vilket gör den likvärdig med minstd_rand. Förbättringen kommer dock på bekostnad av en ökad körtid på nästan 700%.

Test utfördes inte på 64-bitarsvarianterna av de inbyggda C++-algoritmerna på grund av begränsningar i TestU01.

4.3 Algoritmkonstruktion

XORShift presterade redan i sin omodifierade version väl i testen. Alla testfall förutom MaxOft lyckades. Körtiden på 9.31 sekunder för att generera 2^{31} heltal var kortare än alla testade algoritmer i Random-biblioteket. Den modifierade varianten av XORShift byter plats på bithalvorna för varje genererat tal, vilket får den att även klara MaxOft-testet. Förbättringen i antal klarade testfall innebär en ökad körtid på ungefär 60%, vilket beroende på tidskrav kan berättigas utifrån resultatet. Den modifierade algoritmen är i de utförda testen drygt 31% snabbare än den snabbaste algoritmen (mistd_rand) i Random-biblioteket, och drygt 81% snabbare än den i Random-biblioteket som klarar alla test (mt19937).

Kapitel 5

Diskussion och slutsats

5.1 Diskussion

Att mäta hur bra en pseudoslumptalsgenererande algoritm presterar är ingen enkel uppgift. Varje gång en PRNG körs med olika startvärden, ger den (förhoppningsvis) olika resultat, vilket gör att mätningar av algoritmen kan variera kraftigt från körning till körning.

I denna studie har vi valt att använda oss av uppsättningen av test i bbattery_SmallCrush, vilket är ett relativt snabbt testpaket som ingår i TestU01. Resultatet är därför till stor del bundet till en uppsättning test som vi själva inte har konstruerat, men som ingår i välkänd testsvit som rekommenderas av bland andra Gentle [5, sid.80]. För att ytterligare testa den modifiering av XORShift som vi har gjort, skulle fler test kunna utföras. Andra val av parametrar x, y, z och w skulle också kunna testas för att se vilket utfall det får. Kanske kan alla test i bbattery_SmallCrush klaras bara genom att ändra någon av parametrarna, utan att behöva göra någon efterbehandling.

Rigorösa test tar tid att utföra. Av praktiska har vi valt att endast använda det kortare av testbatterierna i TestU01. Vid mer utförliga studier av en algoritms prestanda krävs fler och kraftfullare test. Det är sannolikt att för- och nackdelar med de testade generatorerna skulle kunna observeras genom att köra fler och mer omfattande testbatterier. Denna balansgång mellan vad som är praktiskt utförbart och teoretiskt nödvändigt måste hanteras från fall till fall. Vi anser att det för denna studies omfattning, där tiden är begränsad och kliniska resultat inte är huvudfokus, räcker med ett kortare test.

Något mer att tänka på när en algoritm modifieras för att klara ett visst statistiskt test är att det kan vara så att algoritmen faktiskt skräddarsys för att klara just det testet. Beroende på hur testet ser ut är det möjligt att algoritmen inte blir bättre i alla avseenden, men att den passar bättre för just det statistiska testet. Det framgår tydligt i avsnitt 3.3.2 att olika test kan kontrollera helt olika egenskaper hos resultatet.

Litteraturstudien visade att det inte går att “slumpmässigt” konstruera en algo-

ritm för pseudoslumptal, utan det krävs stor förståelse för teorin bakom implementationen. Därför verkar det tämligen omöjligt att inom studiens omfång och med författarnas förkunskaper utveckla en helt ny välpresterande PRNGs från grunden. Med denna uppbarelse valde vi istället att försöka förbättra en existerande algoritm. Valet av XORShift för den egna implementationen beror på dess prestanda i förhållande till simplicitet, och att en enkel förändring gav ett mätbart bättre resultat. En alternativ testmetod hade varit att jämföra de statistiska testvärdena, men då dessa varierade stort mellan körningar lämpade det sig bättre att endast jämföra algoritmer på en klarar/misslyckas-skala.

5.2 Slutsats

Den modifierade varianten av XORShift som har testats presterar enligt testet bbattery_SmallCrush bättre både fördelnings- och tidsmässigt än algoritmerna i C++11:s Random-bibliotek. Den modifierade XORShift-algoritmen klarade samtliga test, vilket kan bero på testbatteriets begränsade omfattning. I C++11:s Random-bibliotek var det endast mt19937 som klarade samtliga test, men den algoritmen var 81% långsammare än den modifierade XORShift-algoritmen. Baserat på dessa test kan en med enkla medel egenmodifierad algoritm prestera bättre än algoritmerna i C++11:s Random-bibliotek.

Bilaga A

Implementation av XORShift i C++

Detta är en implementation av Marsaglias XORShift-algoritm som presenterades som algoritm 9 i avsnitt 2.8.

```
uint32_t x, y, z, w;

void start(uint32_t seed)
{
    x = 123456789;
    y = 362436069;
    z = 521288629;
    w = seed; // Startvärde
}

void advance_state() // Upprepas för varje genererat värde
{
    uint32_t t;
    t = x ^ (x << 11);
    x = y; y = z; z = w;
    w = w ^ (w >> 19) ^ (t ^ (t >> 8)); // Resultat
}

uint32_t get_rand_bits() // Anropas för att generera nytt pseudoslumptal
{
    advance_state();
    return w;
}
```

Bilaga B

Modifierad XORShift i C++

```
uint32_t x, y, z, w;

void start(uint32_t seed)
{
    x = 123456789;
    y = 362436069;
    z = 521288629;
    w = seed; // Startvärde
}

void advance_state() // Upprepas för varje genererat värde
{
    uint32_t t, u, v;
    t = x ^ (x << 11);
    x = y; y = z; z = w;
    w = w ^ (w >> 19) ^ (t ^ (t >> 8));

    u = w & 0x0000FFFF;
    u = u << 16;
    v = w & 0xFFFF0000;
    v = v >> 16;
    w = u | v // Byt plats på bithalvorna för resultat
}

uint32_t get_rand_bits() // Anropas för att generera nytt pseudoslumptal
{
    advance_state();
    return w;
}
```

Litteraturförteckning

[1] Robert G. Brown. Dieharder: A random number test suite. http://www.phy.duke.edu/~rgb/General/dieharder.php Hämtad 2014-02-19.

[2] I-Te Chen, Jer-Min Tsai, and Jengnan Tzeng. Audio random number generator and its application. *2011 International Conference on Machine Learning and Cybernetics vol.4*, pages 1678 – 1683, Juli 2011.

[3] Bruce Jay Collings. Compound random number generators. *Journal of the American Statistical Association Volume 82, Issue 398*, pages 525 – 527, Juni 1987.

[4] cppreference.com. Pseudo-random number generation. http://en.cppreference.com/w/cpp/numeric/random Hämtad 2014-02-18.

[5] James Gentle. *Random Number Generation and Monte Carlo Methods*. Springer New York, 2003.

[6] Helmut G. Katzgraber. Random numbers in scientific computing: An introduction. *Lecture given at the International Summer School Modern Computational Science (August 9-20, 2010, Oldenburg, Germany)*, 2010. http://arxiv.org/abs/1005.4117 Hämtad 2014-02-18.

[7] Donald Knuth. *The art of computer programming Vol.2 Seminumerical Algorithms*. Addison Wesley Longman, 3rd edition, 1997.

[8] Pierre L'Ecuyer and Richard Simard. *TestU01 - A Software Library in ANSI C for Empirical Testing of Random Number Generators*. Departement d'Informatique et de Recherche Operationnelle Universite de Montreal, Maj 2013. http://www.iro.umontreal.ca/~simardr/testu01/guideshorttestu01.pdf Hämtad 2014-02-19.

[9] Martin Lüscher. A portable high-quality random number generator for lattice field theory simulations. *Computer Physics Communications Volume 79, Issue 1*, pages 100 – 110, Februari 1994.

[10] M. Donald MacLaren and George Marsaglia. Uniform random number generators. *Journal of the ACM Volume 12, Issue 1*, pages 83 – 89, Januari 1965.

[11] George Marsaglia. Xorshift rngs. *Journal of Statistical Software Volume 8, Issue 14*, pages 1 – 6, Juli 2003.

[12] George Marsaglia and Arif Zaman. A new class of random number generators. *The Annals of Applied Probability Vol.1, Issue 3*, pages 462 – 480, Augusti 1991.

[13] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation Vol.8(1)*, pages 3 – 30, Januari 1998.

[14] Microsoft. subtract_with_carry class. `http://msdn.microsoft.com/en-us/library/bb982648.aspx` Hämtad 2014-02-14.

[15] Stephen K. Park and Keith W. Miller. Random number generators: good ones are hard to find. *Communications of the ACM Volume 31 Issue 10*, pages 1192 – 1201, Oktober 1988.

[16] Stephen K. Park, Keith W. Miller, and Paul K. Stockmeyer. Technical correspondence. *Communications of the ACM Volume 36 Issue 7*, pages 105 – 110, Juli 1993.

[17] Andrew Ruhkin et al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. National Institute of Standards and Technology, April 2010. `http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf` Hämtad 2014-02-18.

[18] Jer-Min Tsai, I-Te Chen, and Jengnan Tzeng. Random number generated from white noise of webcam. *2009 Fifth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pages 214 – 217, September 2009.

[19] Ilpo Vattulainen et al. A comparative study of some pseudorandom number generators. *Computer Physics Communications Volume 86, Issue 3*, page 209 – 226, Maj 1995.