NTI Nacka
Teknikprogrammet-Teknikvetenskap
Gymnasiearbete 100p
HT 2023 - VT 2024

Den Kula titeln

o mycket kulare under titlen

Eduards Abisevs
eduards.abisevs@elev.ga.ntig.se
Leo Altebro
leo.altebro@elev.ga.ntig.se
Handledare: Elias

Innehåll

1 Inledning

1.1 Introduktion till ämnet

Spelbranschen är en industri som omsätter miljardbelopp och precis som alla industrier utvecklas den med resten av världen. Industrier över hela världen har som mål att hitta nya sätt att effektivisera och sänka kostnad på det arbete som utförs för att bedriva vinst och spelbranschen har följt denna långvariga trend med till exempel online casinon. Inom spelbranschen är kortspel vanligt förekommande, att försöka automatisera dem är ett logiskt steg att ta, men för en så stor industri vill man vara extra säker på att allt utförs på bästa sätt. Alla sätt att blanda kort är nämligen inte lika bra, vilken sorteringsmetod som används kan ha påverkan på resultatet. Teknologins inflytande inom spelbranschen ökar kraftigt, de största delarna av branschen bedrivs mer och mer av maskiner och algoritmer som får stor potentiellt inflytande på spelresultat, därför är det bäst att börja tänka på möjliga problem så snart som möjligt. Redan nu finns det blandningsmaskiner för kortlekar som vi inte vet någonting om; varken hur de funkar eller hur bra de är. Allt som vi vet är de är certifierade av tredjepartsföretag. Faktumet att de kan kosta upp till 100 000 kr betyder att inte vem som helst kan ha tillgång till en kortblandningsmaskin.

1.2 Syfte

Syftet med denna undersökning är att ta fram den hypotetiskt bästa kortblandaren utifrån två faktorer: 1) hur slumpmässigt den algoritm som den byggs efter kan blanda kort; 2) till vilken grad den potentiella maskinen byggd efter algoritmen skulle fungera i verkligheten. Med resultaten förväntas variationen i slumpmässighet för olika blandningsmetoder kunna visas upp samt kunna använda de resultaten för att komma fram till den hypotetiskt definitiva maskinen, något som är viktigt då spelbranschen handlar mycket om chans. Det är därför viktigt att se till så allt funkar på bästa sätt. I den här vetenskapliga rapporten kommer en jämförelse av hur effektivt framtagna blandningsmetoder kan fungera i en hypotetisk blandningsmaskin att utföras. Samtidigt som man kan utforska hur en dator kan göra slumpmässiga sekvenser på ett effektivt sätt med tanken att den ska tillämpas till en potentiell kortblandare. Detta är viktigt för att spelbranschen är en stor industri som handlar mycket om tur, att se till att de slumpmässiga resultaten är framtagna på bästa möjliga sätt är en essentiell del. Vi söker med hjälp av data svaret på frågan:

Utifrån slumpmässighet och effektivitet, vilken kort blandningsmetod är bäst för en potentiell spelkortsblandare som uppfyller följande:

2 Teori

2.1 Beräkningsteori

Beräkningsteori är en del av matematik vars syfte är grundat i hur och om problem kan bli lösta på olika beräkningssätt. Beräkningsteori har flera olika grenar. En gren handlar om vad som går att bevisa inom matematik angående om nummer och funktioner är beräknelig eller inte. Med beräknelig menas något som kan beräknas, det vill säga värderas, uppfattas eller förutses, när det kommer till matematik syftar det på att bestämma något via matematiska modeller/processer, alltså att kalkylera.

Figur 1: Your caption here

2.2 Slumpmässighet

Slumpmässighet är en egenskap som anger att något sker utan klart mönster, när något är helt slumpmässigt är det i princip omöjligt att förutse. Slumpmässighet inom matematik är byggd beräkningsteori och den delas upp i två beroende på om det gäller slumpmässighet av en bestämd mängd eller en oändlig mängd av objekt (Terwijn, 2016).

3 Metod

Ett av målet med undersökningen var att avgöra hur slumpmässiga algoritmerna var. För att undersöka detta kriterium valdes det att simulera stora mängder av kortblandningar. Metoden kunde indelas i tre huvudområdena 1) framtagande av algoritmerna 2) rådata generering via simulation 3) normalisering av rådata och statistiska tester

3.1 Testmiljö

Datan insamlades från digital simulation på olika kortblandningsmetoder. De algoritmerna som undersöktes var skrivna i programspråk Rust i försök att göra dem så likt verkligheten som möjligt. För insamling av data användes programmeringsspråket Python 3.11 och utnyttjades bibliotek som Numpy till datamängd bearbetning, Scipy till statistisk analys (Chi-square) och Matplotlib till visualisering av resultat. Undersökningen kördes på en linux dator med följande specifikationer:

Typ	Specification
Processor	AMD Ryzen 5 3600 6 cores / 12 threads 3.6 GHz
RAM	15.93 GB
Hårdisk	KINGSTON SA400S3, 447 GB
Operativsystem	`Ubuntu 22.04.3 (LTS) x86_64,` `linux kernel 6.2.0-36-generic`

Tabell 1: Linux Dator

3.2 Algoritmer som undersöktes

Då tanken var att algoritmerna skulle potentiellt användas i ett fysiskt inbäddad system fanns det vissa kriterier som algoritmerna skulle följa: det skulle gå att bygga en maskin, den skulle vara slumpmässig, dvs. simulation skulle vara så lik verkligheten som möjligt. Med det sagt så måste vi bestämma oss för en pseudo-random number generator (PRNG) algoritm bestämmas. En väldigt känd och testad algoritm är Mersenne Twister (MT 19937) som har en väldigt långt period innan siffrorna börjar att upprepa sig, mer exakt 2^{19937}.

3.2.1 Riffle shuffle

THe riffle shuffle

3.3 Simulation för kortblandningar och implemenation

Simulationen utvecklades med Rust (version 1.73.0) och användes bibliotek (Rust Crates) som Rand (version 0.8) för PRNG (rand utvecklare 2023). Rayon för enkel användning av parallell multiprocessing (rayon utvecklare 2023). Simulation utgår från att man genererar matris med bestämd mängd av kortlekar. Vart varje kort, låt kalla dem till x som följer följande mängden $\{x \in \mathbb{N}, 0 \leq x \leq 51\}$ Resulterande datamängden är en matris D med 52 kolumner och m antal rader, vart m värde visar hur många kortlekar det finns i datamängden.

$$D = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & \cdots & x_{0,51} \\ x_{1,0} & x_{1,1} & x_{1,2} & \cdots & x_{1,51} \\ x_{2,0} & x_{2,1} & x_{2,2} & \cdots & x_{2,51} \\ \vdots & \vdots & \vdots & & \vdots \\ x_{m,0} & x_{m,1} & x_{m,2} & \cdots & x_{m,51} \end{bmatrix} \tag{1}$$

För att bestämma m värde följdes ett kriterium från ett av testerna som ska utföras, klassiskt poker test mer i detalj beskriven i underrubriken 3.3 . Till hjälps användes NIST (National Institute of Standards and Technology, 2023) , i den nämns det att det är viktigt för chi-square approximationens trovärdighet att minsta kategorin ska inte vara mindre än fem teoretiska framkommande i den kategorin. Detta ska tillämpas till simulationen på följande sätt. Det är viktigt att hitta den minsta kategorin som kan förekomma. I poker är den mest sällsynta pokerhand kungliga färgstege (Royal Flush) som kan räknas ut på följande sätt antal av alla kortkombinationer med fem kort. $\binom{52}{5} = 2\,598\,960$

Kungliga färgstege finns det fyra kombinationer av i kortleken som kan räknas ut på följande sätt totala antal kombinationer av kungliga färgstege delat med totala antal femkorts kombinationer från standardkortlek. $\binom{4}{1}$!= 4 4 2'598'960=1649'740 dvs en kunglig färgstege per 649'740 händer. För att datamängd ska uppfylla minsta kravet för poker testet beräknades längden av matrisens m så att m= 649'740 5 = 3'248'700

Efter teoretiska beräkningar av datamängden är det viktigt att välja lämplig datatyp att representera datan i. Man inser att högsta talet som behövs är 51 detta betyder att minsta datatypen får användas som är 8 bit långa som har 28= 256 som är lämpligt antal kombinationer, vi behöver bara naturliga talsidan, högsta talet som kan representeras är 255 inkluderat 0. Samtidigt som det är enkelt att spara datan i binärt format för att 8 bits eller 1 byte är en fundamental och uniform mått i minnesadressen. Med det sagt ingen data bearbetning behövs och det är enkelt att ladda in denna i till exempel Python med numpy till statistiska testerna av rådatan. En till argument till varför valdes minsta möjliga datatyp är för det kommer relativt många rådata filerna och varje fil kommer att ta upp: 52 m=52 3'248'700 = 168'932'400 bytes delar man detta värdet med 1 MB = 1024 1024= 1,048,576 bytes så att 168'932'4001,048,576161 MB / datafil Att räkna hur mycket minne datamängden ska ta är viktigt för att det inte bara är en fil som ska undersökas men relativt många. Därför att sista teoretiska delen att bestämma är hur många iterationer per algoritm ska genomföras. Med iterationer menas hur många gånger kortleken blandas följande på varandra i bilaga listing 2 .

eller. Listing 2

Allt kod finns på github, länken finns i biloga appendix A

Referenser

National Institute of Standards and Technology (2023). *Engineering Statistics Handbook - Section 1.3.5.15: Chi-Square Goodness-of-Fit Test*. URL: `https://www.itl.nist.gov/div898/handbook/eda/section3/eda35f.htm` (Hämtad 2023-10-04).

Terwijn, Sebastiaan A. (2016). "The Mathematical Foundations of Randomness". I: *The Challenge of Chance: A Multidisciplinary Approach from Science and the Humanities*. Utg. av Klaas Landsman och Ellen van Wolde. Cham: Springer International Publishing, s. 49–66. ISBN: 978-3-319-26300-7. DOI: 10.1007/978-3-319-26300-7_3. URL: https://doi.org/10.1007/978-3-319-26300-7_3.

Bilagor

A Github

github link: https://github.com/Abishevs/gymarbete

B Källkod simulation

Kodlistning 1: Python example

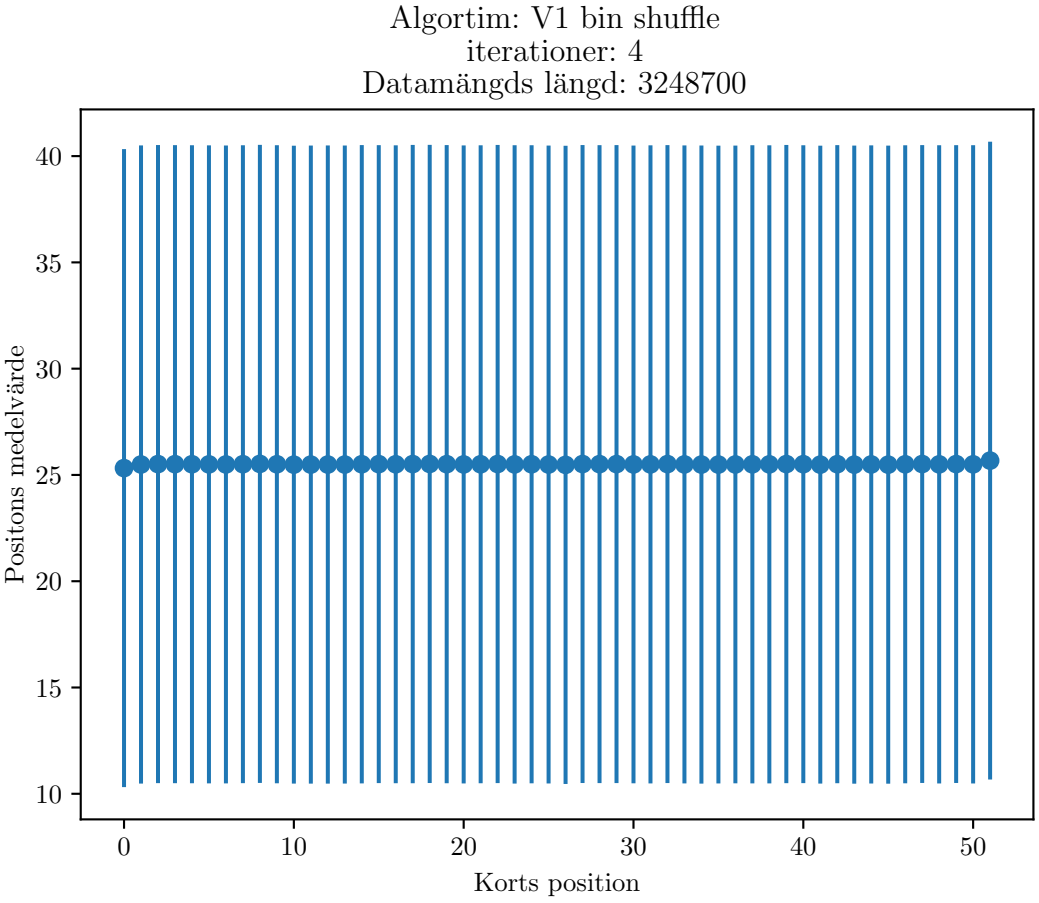
```
# Your source code here
print("Hello, World!")
```

C Källkod algorithm 1 implemention

Kodlistning 2: algorithm 1 implemention

```
impl  ShufflingAlgorithm for Algorithm1{
    fn name(&self) -> &str {
        "v1_bin_shuffle"
    }

    fn shuffle(&self, deck: &mut Deck){
        const BIN_COUNTS:usize = 6;
        let mut bins: Vec<Vec<Card>> = vec![Vec::new(); BIN_COUNTS];

        let mut rng = rand::thread_rng();
        for &card in deck.iter() {
            let random_bin = rng.gen_range(0..BIN_COUNTS);
            // put the cards in random bins
            bins[random_bin].push(card);
        }

        let mut deck_position = 0;
        // Reassemble the deck
        for bin in bins {
            for card in bin {
                deck[deck_position] = card;
                deck_position += 1;
            }
        }
    }

}
```

D Figure example



Figur 2: A PGF figure example från `matplotlib`.