Rapport de mi-projet

19 mars 2023

PREDICTIONS AU POKER TEXAS HOLDEM

Auteurs:
Amaury Curiel
Quentin Gavoille

Encadrants:
Antoine Genitrini
MedhiNaima

Table des matières

1 Introduction **2**

2 Définitions **2**
2.1 Convention 2
2.2 Générateur Congruentiel Linéaire (GCL) 2
2.3 Nombres pseudo aléatoires 2
2.4 Mélange uniforme 3

3 Générateurs de nombres pseudo aléatoires **3**
3.1 Critères pour un bon GNPA 3
3.2 Méthode des carrés médians 3
3.3 Mersenne twister 3
3.4 Générateurs observant des phénomènes purement aléatoires 4

4 Générateurs congruentiels linéaires **4**
4.1 Choix des paramètres d'un GCL 4
4.2 Cas du générateur de nombres pseudo aléatoires de Java 5

5 Définition du problème **6**
5.1 Objectif du projet 6
5.2 Approche du problème 7

6 Mélange naïf **7**
6.1 Présentation du mélange 7
6.2 Étude théorique du mélange 8
6.3 Méthode utilisée pour retrouver la graine 8
6.4 Résultats 9

7 Mélange uniforme **10**
7.1 Présentation du mélange 11
7.2 Théorie 11
7.3 Méthode utilisée pour retrouver la graine 12
7.4 Résultats 12

8 Conclusion **13**

1 Introduction

Tirer un nombre aléatoire est quelque chose de très simple pour un être humain mais pour un ordinateur, la recherche ne semble pas montrer de consensus à ce sujet. Cependant, de nombreux domaines tels que la cryptographie reposent sur cette génération de nombres aléatoires. De plus, les algorithmes randomisés permettent souvent d'apporter une solution plus précise ou plus rapide à certains problèmes. Nous pouvons par exemple citer l'algorithme de Welzl [3], un algorithme probabiliste permettant de calculer une approximation d'un cercle couvrant minimal d'un groupe de point en $O(n)$ ou encore l'heuristique de Monte Carlo utilisée notamment en théorie des jeux [1], qui permet d'arbitrer entre deux possibilités en simulant un grand nombre d'expériences aléatoires. Afin de pallier à ce problème, les chercheurs ont, dès le milieu du $20^{ème}$ siècle, essayé de trouver des suites très sensibles aux conditions de départ, ayant une distribution uniforme et, dans un soucis de performance inhérent aux technologies de l'époque, peu coûteuses en calcul. C'est dans ce cadre que furent développé les **générateurs congrueltiels linéaires**, générateurs que nous avons étudiés dans ce projet, et que nous nous sommes fixés comme objectif d'arriver a prédire la suite d'entiers générés par de tels générateurs à partir d'observations d'expériences (pseudo)aléatoires. Pour fournir de telles expériences, le poker Texas Holdem fut un bon candidat : si nous arrivons a prédire la sortie d'un générateur congruentiel linéaire, nous serons en mesure de gagner à presque tous les coups!

2 Définitions

2.1 Convention

Lorsque nous écrirons des entiers en binaire, nous écrirons toujours le bit de poids faible a gauche. Par exemple, 100 vaut 1 et non 8.

On utilisera également l'abréviation "var" pour variable aléatoire réelle.

2.2 Générateur Congruentiel Linéaire (GCL)

Dans la suite, nous appellerons Générateur Congruentiel Linéaire(plus tard abrégé en GCL), toute suite de la forme :

$$X_n = \begin{cases} X_0 & \text{si } n = 0 \\ (aX_{n-1} + c)\ mod(m) & \text{si } n > 0 \end{cases}$$

avec $a, m > 0$, $c \geq 0$ et $X_0 \geq 0$. Nous appellerons a le multiplieur , c l'incrément, m le module et X_0 la graine du générateur.

2.3 Nombres pseudo aléatoires

Comme mentionné en introduction, il est impossible de générer une suite de nombres parfaitement aléatoires à partir d'un ordinateur. On appelle donc nombres pseudo aléatoires, toute suite "suffisamment imprévisible" pour que l'on ne puisse pas "facilement"déterminer une forme explicite de cette suite. Ces termes, un peu vagues seront définis par la suite.

2.4 Mélange uniforme

On définit un mélange de $[\![1, n]\!]$ comme étant l'ensemble des permutations de $[\![1, n]\!]$. De plus, on diras qu'un mélange est uniforme lorsque chaque permutation apparaît avec une probabilité de $\frac{1}{n!}$

3 Générateurs de nombres pseudo aléatoires

Dans cette section, nous nous attarderons à présenter sans toutefois rentrer dans le détail, des exemples de générateurs de nombres pseudo-aléatoires(GNPA) déjà existants.

3.1 Critères pour un bon GNPA

Un GNPA est considéré comme bon lorsque :

- Il possède une longue période. En effet, un générateur qui génère des suites 3-périodiques n'as rien d'aléatoire.
- La séquence de nombres produit ne permet pas aisément de prédire les sorties futures. En effet, un générateur qui génère la suite 1,2,...., bien que possédant une grande période ne peut être considérée comme satisfaisant.
- Le générateur doit être assez rapide et économe en ressources.

3.2 Méthode des carrés médians

La méthode des carrés médians est un des premiers générateurs de nombres pseudo aléatoires crée. L'algorithme est le suivant:

- Prendre une graine x_0
- L'élever au carré
- Prendre les deux chiffres du milieu et recommencer

Ce générateur bien que satisfaisant le critère de rapidité, n'est pas satisfaisant d'un point de vu aléatoire. En effet, une séquence pseudo-aléatoire finira toujours par déboucher sur une suite infinie de zéro. Par exemple, la graine 12 finit par donner 0 en 8 étapes.

3.3 Mersenne twister

Contrairement à la méthode des carrés médians, l'algorithme du Mersenne twister fait partie des algorithmes dit satisfaisants au regards des critères de la section 3.1 pour générer des nombres pseudo aléatoires. Le Mersenne twister possède une très grande période et est facile à implémenter, ce qui en fais un algorithme de choix pour générer des nombres pseudo-aléatoires. Il est notamment utilisé comme étant le GNPA par défaut dans de nombreux langages tels que Python, PHP ou encore Matlab. Cependant, cet algorithme nécessite de stocker un grand nombre d'entiers, ce qui le rend inutilisable pour des applications avec un espace mémoire limité (jeux vidéos sur cartouche, systèmes embarqués,...).

3.4 Générateurs observant des phénomènes purement aléatoires

Jusqu'ici, nous avons présenté des générateurs de nombres pseudo-aléatoires. Cependant, de tels nombres ne peuvent et ne doivent pas être utilisés pour des applications où l'aléa joue un rôle critique. En effet, si les sites de poker en ligne utilisaient un GCL pour gérer leur parties, il suffirait de lire ce document pour pouvoir gagner à tous les coups. C'est pourquoi ont été développé des générateurs observant des phénomènes purement aléatoires. Le principe de tels générateurs est d'observer un phénomène imprévisible comme par exemple la désintégration d'un atome radioactif ou encore les impacts d'électrons sur une surface donnée et de traduire ces phénomènes en entiers. Ces générateurs, bien que satisfaisant du point de vue de la qualité des entiers générés, sont pour la plupart coûteux ou lents, c'est pourquoi, quand l'aléa n'est pas critique, on préférera utiliser des générateurs de nombres pseudo-aléatoires.

4 Générateurs congruentiels linéaires

Dans cette partie, nous allons étudier les générateurs congruentiels linéaires, générateurs très largement implantés dans les langages de programmations. En effet, des langages comme C , C++ ou encore Java utilisent ce type de GNPA. Dans cette partie, nous étudierons le cas du générateur de Java.

4.1 Choix des paramètres d'un GCL

On définit tout d'abord la notion de période d'un GCL. Soit $(U_n)_{n\in\mathbb{N}}$ un générateur congruentiel linéaire de multiplieur a, d'incrément c, module m et de graine X_0. On définit alors la période de U comme étant le plus petit entier n tel que pour tout $m < n$, $U_n \neq U_m$. Nous pouvons alors nous poser la question de savoir quels choix de paramètres sont optimaux pour un GCL. Il est clair que prendre a = 0 est un très mauvais choix car la suite serait constante à c. De même, prendre a = 1 donnerait la suite des multiples de c, ce qui n'est pas satisfaisant d'un point de vue aléatoire. Le théorème suivant donne un critère assurant l'optimalité de la période d'un GCL:

Théorème 1 *(Critère de Knuth) La période d'un GCL associé aux entiers a, c, m , X_0 est maximale et de periode m si et seulement si:*

- $c \wedge {}^1 m = 1$ (1)
- $\forall p \in \mathbb{P}^2$ *tq* $p|m, \exists k \in \mathbb{N}$ *tq* $a - 1 = kp$ (2)
- $\exists k \in \mathbb{N}$ *tq* $m = 4k \Rightarrow \exists k'$ *tq* $a - 1 = 4k'$ (3)

La preuve de ce théorème se trouve dans le livre [2] à la page 17.

[1] On note $a \wedge b$ le PGCD de a et de b

[2] On note $\mathbb{P}$ l'ensemble des nombres premiers

4.2 Cas du générateur de nombres pseudo aléatoires de Java

Dans le cas du GCL de Java, nous avons:

- Le multiplieur : 25214903917
- L'incrément : 11
- Le module : 2^{48}

On a alors d'après le critère de Knuth que l'algorithme java est de periode maximale (2^{48}). En effet:

- Vérifions la condition (1): $c = 11$ et 11 est premier ainsi $c \wedge 2^{48} = 1$
- Vérifions la condition (2): Le seul diviseur premier de 2^{48} est 2 et 25214903917-1 est pair donc la condition (2) est vérifiée
- Vérifions la condition (3): $2^{48} = 4 * 2^{46}$ et 25214903916 = 4∗6303725979 ainsi la condition (3) est vérifiée.

. Cependant, l'algorithme de Java rajoute un "mask" qui va tronquer la sortie du GCL qui est initialement sur 48 bits (X_0) en un entier de 32bits (x_0) en gardant les 32 bits de poids fort de X_0. A cause de ce mask, pour retrouver X_0 à partir d' x_0 et de x_1, nous devons nous assurer du fait que si le GCL de java nous retourne un nombre x_0 puis un nombre x_1, alors il existe un unique X_0 avec comme bits de poids fort x_0 tels que $GCL(X_0) = x_1$ avec $GCL(x)$ l'application qui applique le générateur java à l'entier x. L'existence est assurée par construction du générateur Java. Ceci assure le fait que l'application $\begin{array}{rcl} F : \mathbb{Z}/2^{48}\mathbb{Z} & \rightarrow & \mathbb{Z}/2^{48}\mathbb{Z} \\ n & \mapsto & U_n \end{array}$, bien définie dès lors que X_0 est bien définie, est bijective donc que notre problème possède au moins une solution. Nous devons donc maintenant montrer que le système possède au plus une solution. Pour cela, on sait que $aX_0 + c = d * 2^{48} + x_1 * 2^{16} + f$ avec f un entier inférieur à 2^{16} et d les bits dépassant 2^{48}. Comme on travail dans $\mathbb{Z}/2^{48}\mathbb{Z}$, il est inutile de considérer d. Ainsi, notre multiplication se ramène à : $aX_0 + c = x_1 * 2^{16}$. Lorsque l'on pose la multiplication, nous obtenons un système linéaire. Illustrons cela sur des entiers de 12 bits dans lesquels on ne garde que les 8 bits de poids fort avec le multiplieur égal à 1001:

- 1001 s'écrit : 1001011111
- on écrit : $X_0 = x_0^1, x_0^2, ..., x_0^{12}$ et de même pour x_1
- on pose la multiplication et on obtient le système suivant:

$$\begin{cases} x_0^1 = x_1^1 \ (1) \\ x_0^2 = x_1^2 \ (2) \\ x_0^3 = x_1^3 \ (3) \\ x_0^4 + x_0^1 = x_1^4 \ (4) \\ x_0^5 + x_0^2 = x_1^5 \ (5) \\ x_0^6 + x_0^3 + x_0^1 = x_1^6 \ (6) \\ x_0^7 + x_0^4 + x_0^2 + x_0^1 = x_1^7 \ (7) \\ x_0^8 + x_0^5 + x_0^3 + x_0^2 + x_0^1 = x_1^8 \ (8) \\ x_0^9 + x_0^6 + x_0^4 + x_0^3 + x_0^2 + x_0^1 = x_1^9 \ (9) \\ x_0^{10} + x_0^7 + x_0^5 + x_0^4 + x_0^3 + x_0^2 + x_0^1 = x_1^{10} \ (10) \\ x_0^{11} + x_0^8 + x_0^6 + x_0^5 + x_0^4 + x_0^3 + x_0^2 = x_1^{11} \ (11) \\ x_0^{12} + x_0^9 + x_0^7 + x_0^6 + x_0^5 + x_0^4 + x_0^3 = x_1^{12} \ (12) \end{cases}$$

- Les inconnues de ce systèmes sont $x_0^i | i \in [\![0, 4]\!]$
- Les équations 1 à 4 n'apportent pas d'informations
- les équations 10, 11 et 12 sont indépendantes. Il faut en trouver une quatrième pour avoir que le système possède une unique solution.
- Pour chaque équation e entre la 5 ème et la 9 ème incluse, on calcule le déterminant de la matrice donnée par les équations 10,11,12 et e jusqu'à trouver un déterminant dans $\mathbb{Z}/2\mathbb{Z}$.
- On obtiens avec l'équation 8 un déterminant de 1 ce qui signifie que les équations 8,10,11,12 forment une base de $(\mathbb{Z}/2\mathbb{Z})^4$,
- Nous avons alors que le système possède au plus une solution. On a donc que notre problème possède une unique solution. Il est donc possible de bruteforce la graine sur 48 bits à partir de x_0 et x_1 sur 32 bits.

5 Définition du problème

5.1 Objectif du projet

L'objectif de ce projet est de prédire le mélange d'un jeu de cartes (représentée par des entiers) qui a été mélangé par l'algorithme du "Shuffle de Knuth" dont l'algorithme général est explicité ci-dessous :

Algorithm 1 Shuffle de Knuth

Data: $Liste - triee < int > l$
Result: La liste l mélangée
for $i = n - 1; i > 1; i + +$ **do**
  $index = random(0, i)$
  $tmp = l[index]$
  $l[index] = l[i]$
  $l[i] = tmp$
end
return c

Dans notre cas, la fonction *random* fournit un entier aléatoire généré par le GCL de Java.
Pour ce faire, nous observerons une partie de poker Texas Holdem et en particulier l'état de la table après les mises. Nous fournirons enfin un algorithme qui permet de retrouver la graine du GCL à tous les coups si 2 adversaires montrent leurs cartes après s'être couché.

5.2 Approche du problème

Pour résoudre le problème défini à la section précédente, nous avons tout d'abord codé un générateur de bits aléatoires dont l'algorithme est explicité ci-dessous:

Algorithm 2 nextNbits

Data: Java.util.Random r, nombres de bits désirés i
Result: La liste l mélangée
(*état de la mémoire:
k le nombre de bits que l'on a déjà consommé
t tableau de 32 bits*)
$buf = [i]$ **for** $d = 0$ *to* i **do**
  $k = k + 1$
  if $k \geq 32$ **then**
    $k = 0$
    $t = toBitsArray(r.nextInt())$
  end
  $buf.add(t[k])$
end
return buf

Le principe de cet algorithme est de demander un entier pseudo aléatoire à Java, puis de l'égrainer bit à bit jusqu'à qu'on ait besoin d'un nouvel entier.
Suite à cela, nous avons approché le problème par difficulté croissante. En effet, nous avons tout d'abord essayé de retrouver la graine du générateur Java avec un mélange sans rejet (non uniforme), puis nous avons tenté avec un mélange uniforme avec rejet et enfin en se passant de certaines informations.

6 Mélange naïf

Dans cette section, nous allons présenter le premier mélange que nous avons utilisé pour déterminer la graine du GCL, qui est le premier degré de difficulté de notre projet. En effet, comme nous le verrons plus tard, un rejet implique une perte d'information de six bits.

6.1 Présentation du mélange

Ce premier mélange est un mélange sans rejet. Pour mélanger avec le Shuffle de Knuth un jeu de carte, nous avons besoin à chaque étape de : $\lceil \log_2(51-i) \rceil$ bits où i représente l'index de la carte que l'on mélange. Cela fait apparaître le problème suivant: pour mélanger la $0^{ème}$ carte de notre jeu, nous avons besoin de 6 bits aléatoires, qui encodent un nombre entre 0 et 63 alors que notre jeu comporte 52 cartes. Pour palier à cela, nous avons décidé de prendre la valeur du groupe de bits aléatoires obtenus modulo le nombre de cartes restantes, ce qui "dé-uniformise" le mélange mais nous permet d'être certains de tous les bits que l'on à lu.

6.2 Étude théorique du mélange

Comme mentionné précédemment, ce mélange n'est pas uniforme. En effet, pour la 1^{ere} carte mélangée,
$P(X = 0) = P(groupede6bits = 0) + p(groupede6bits = 52) = \frac{2}{2^6}$, ce qui contredit l'uniformité du mélange. Informellement, si deux cartes nécessitent n bits pour être mélangées, la carte d'index la plus proche par valeur inférieure de 2^n aura plus de chances de se retrouver au début du paquet mélangeable. Pour i un index et n le nombre de bits nécessaires pour mélanger la carte courante, on a:
$P(X = k) = P(X = 52 + k - i) + P(X = k) = \frac{1}{2^n}\ si\ 2^n < 52 + k - i, \frac{2}{2^n}$ sinon.
Malgré le fait que le mélange ne soit pas réaliste, il s'agit quand même d'un bon mélange pour appréhender notre problème.

6.3 Méthode utilisée pour retrouver la graine

Après avoir mélangé le jeu de cartes, nous devons maintenant nous servir des observations données par la partie de poker pour retrouver la graine du générateur aléatoire.En effet, on souhaite récupérer x_0 et x_1 pour découvrir X_0.Le mélange expliqué ci-dessus nous permet de nous assurer qu'il est possible de retrouver un groupe de 6 bits à partir d'un couple ($valeurCarte$,$IndexCarte$). Enfin, l'hypothèse selon laquelle deux de nos adversaires se sont couchés et ont montré leurs cartes nous permet de nous assurer d'avoir les 64 bits nécessaires pour retrouver la graine. Nous avons donc mis en place l'algorithme suivant pour retrouver la graine sur 32 bits: On garde en mémoire un paquet de carte initialement trié que l'on mélange au fur et a mesure. On s'appuie sur une fonction *isAmbiguous* qui vérifie si un groupe de 6 bits lu correspond bien à un seul groupe de 6 bits de la graine. Si c'est le cas alors on ajoute aux graines possibles le groupe de 6 bits lu, sinon on ajoute les 2 possibilités à toutes les graines possibles. Suite à cela, on mélange le paquet en mémoire. Une fois cela, nous pouvons par bruteforce calculer la graine possible sur 48 bits, cela est rendu possible grâce à la section précédente.

Algorithm 3 $TrouveX_0$

Data: ListeDobservations (entier) : lo
Result: Liste des graines possibles sur 32 bits (string)
tmp = jeu de carte trié

```
for k in range 6 do
    if isAmbigous(deck.index(lo[k])-k),k then
        result = [ ]
         for x in x_0 do
            result += [x + tmp.index(lo[k])]
             result += [x + complement(tmp.index(lo[k]))]
        end
        else
            for x in x_0 do
                result += [x + tmp.index(lo[k])]

            end
        end
    end
    x_0 = result
end
return x_0
```

6.4 Résultats

Grâce à la méthode expliquée ci-dessus, nous avons pu retrouver la graine dans 100% des cas. Nous avons également pu mettre en lumière le fait que ce mélange n'était pas uniforme. Les figures ci-dessous mettent en lumière la non uniformité du mélange: En effet, sur la figure 1, qui correspond au mélange de l'as de pique (la première carte du jeu), nous pouvons observer que la distribution est presque uniforme(la probabilité du rejet est de 13), contrairement à la figure 2 où nous avons un grand pic qui traduit la non uniformité du mélange utilisé. Cela est encore pire sur la figure 3 ou un pic apparaît à chaque fois que l'on tire un nouvel entier. En effet, les probabilités de rejets sont de loin supérieures pour le 8 de carreau que pour l'as de pique.

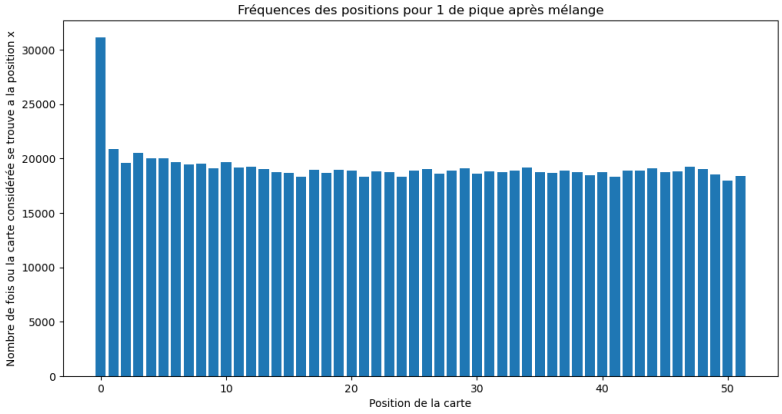

Figure 1: Distribution pour l'as de pique

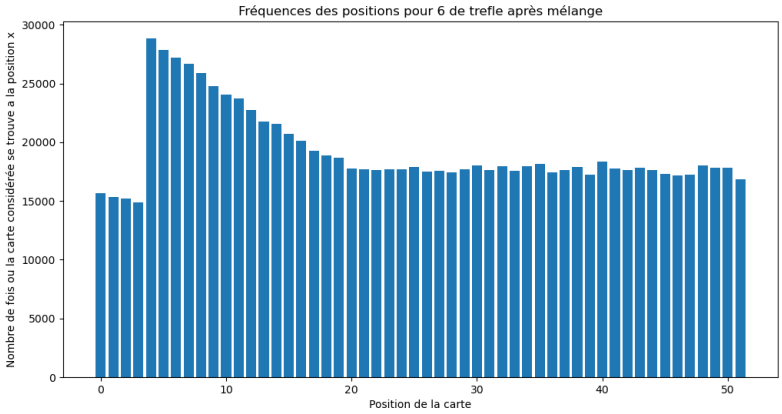

Figure 2: Distribution pour le 6 de trèfle

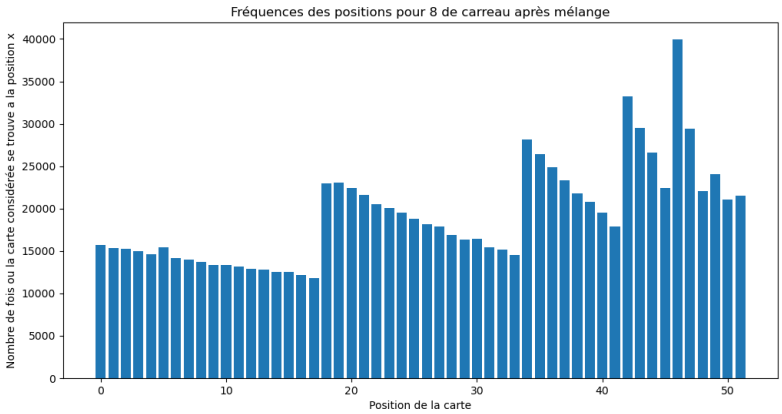

Figure 3: Distribution pour le 8 de carreau

7 Mélange uniforme

Dans cette section, nous allons présenter le second mélange que nous avons utilisé pour déterminer la graine du GCL. Il constitue le second degré de difficulté de notre projet.

7.1 Présentation du mélange

Le second mélange que nous avons utilisé pour notre projet est un mélange avec rejet. En effet, lorsque l'on tire un groupe de n bits, quand la valeur dépasse 52 - la valeur de la carte, alors que dans le mélange précédent nous aurions pris la valeur donnée par le groupe de n bits modulo 52 - la valeur de la carte, ici nous rejetons entièrement ce groupe de n bits et nous en tirons un nouveau. Cette méthode nous assure l'uniformité du mélange (preuve dans la section ci-dessous) et nous permet d'avoir un mélange réaliste. Ce nouveau mélange pose plusieurs difficultés. En effet, notre objectif étant de trouver la graine, Comment pouvons nous nous assurer que les bits que nous avons lus sont bien les bits de la graine? Par exemple comment retrouver la graine (sur 32 bits) : 100000 111111 110000 101000 111000 00? En effet, les bits 111111 vont être rejetés par notre mélange, ce qui rend impossible leur lecture. C'est le problème que nous avons essayé de résoudre dans cette partie.

7.2 Théorie

Commençons tout d'abord par calculer le nombre de rejets pour 64 bits de x_0 et x_1. On note g_n la var qui vaut la valeur du n^{eme} groupe de 6 bits ainsi que R la var qui compte le nombre de rejets dans un groupe de 6 bits.

$$
\begin{aligned}
P(R=0) &= \prod_{i=1}^{11} P(g_i \leq 52-i) \\
&= \frac{46*47*48*49*50*51*...*41}{64^{11}} \\
&= \frac{1901078194188096000}{64^{11}} \\
&\approx 0.03 \\
P(R=1) &= \sum_{k=0}^{11}(\prod_{i\neq k} P(gi \leq 52-i))P(gk > 51-k) \\
&= \frac{8321568899375854080}{64^{11}} \\
&\approx 0.11 \\
P(R=2) &= \sum_{k=0}^{6}(\sum_{\substack{i=0 \\ i\neq k}}^{6}(\prod_{\substack{j\neq k \\ j\neq i}} P(gj \leq 52-j))P(gi > 52-i))P(gk > 52-k) \\
&= \frac{16459587220647292416}{64^{11}} \\
&\approx 0.22 \\
P(R=3) &= \frac{19417432855547788800}{64^{11}} \\
&\approx 0.26 \\
P(R=4) &= \sum_{k=0}^{6}(\sum_{\substack{i=0 \\ i\neq k}}^{6}(\prod_{\substack{j\neq k \\ j\neq i}} P(gj > 52-j))P(gi \leq 52-i))P(gk \leq 52-k) \\
&= \frac{15179553529018184704}{64^{11}} \\
&\approx 0.21 \\
P(R=5) &= \sum_{k=0}^{6}(\prod_{i\neq k} P(gi > 52-i))P(gk \leq 51-k) \\
&= \frac{8256357583903618048}{64^{11}}
\end{aligned}
$$

≈ 0.11

$P(R = 6) = \frac{3188115224410414080}{64^{11}}$
≈ 0.04

$P(R = 7) \approx 0.01$

$P(R = 8) \approx 0.002$

$P(R = 10) \approx P(R = 11) \approx 0$
On a donc un rejet moyen égal à $\mathbb{E}[R] \approx 3,046$.

7.3 Méthode utilisée pour retrouver la graine

Pour retrouver la graine, la méthode utilisée était similaire à la méthode explicitée dans la section précédente. Cependant, lorsque l'on rejette un groupe de 6 bits, nous perdons l'information des 6 bits rejetés et le reste des bits est décalé d'un cran vers la droite. Par exemple, pour la graine
001001 110110 101101 100111 011100 00 qui présente un rejet au 4^{eme} groupe de bits, la graine lue sera : 001001 110110 101101 011100 001100 00. Il s'agit de la même graine avec 011100 inséré en 4^{eme} position. Pour retrouver la graine nous avons utilisé une méthode incrémentale en fonction du nombre de rejets. Pour chaque nombre de rejets possible (ici 11), nous testons tous les cas possibles. Cette méthode fonctionne cependant, elle est très lente. En effet, pour n rejets, il y'a : $\binom{11}{n}$ cas à traiter sois un maximum de 462 cas atteints pour 5 rejets. Cependant, dans chaque cas, pour chaque rejet, nous devons tester 64 valeurs pour le groupe de 6 bits considéré, qui déboucheront sur chacun 216 tests. On a donc pour 5 rejets 2^{16}*462*13^3*64*11^3 = 4 257 277 280 256 000 tests, ce qui avec une machine de l'ordre du GHz représente plus de 1 heure en considérant le test comme opération élémentaire. Cela nous empêche de retrouver la graine dès lors que nous avons au moins 4 rejets, ce qui arrive environ 36% du temps.

7.4 Résultats

Dans un premier temps, intéressons nous au mélange. Sur dix millions de mélanges, nous obtenons des graphes ressemblants à celui-ci pour chaque carte: On reconnait une distribution uniforme, ce qui nous confirme que notre mélange est proche d'un mélange réaliste. Pour ce qui est de la correction de l'algorithme: Si nous avons plusieurs heures devant nous, nous pouvons retrouver à tous les coups la graine utilisée pour mélanger le paquet de carte en ayant seulement 11 cartes de visibles et ce, quelque soit le nombre de rejets. Cependant, dans approximativement la moitié des cas, nous arrivons à trouver la graine en temps raisonnable.

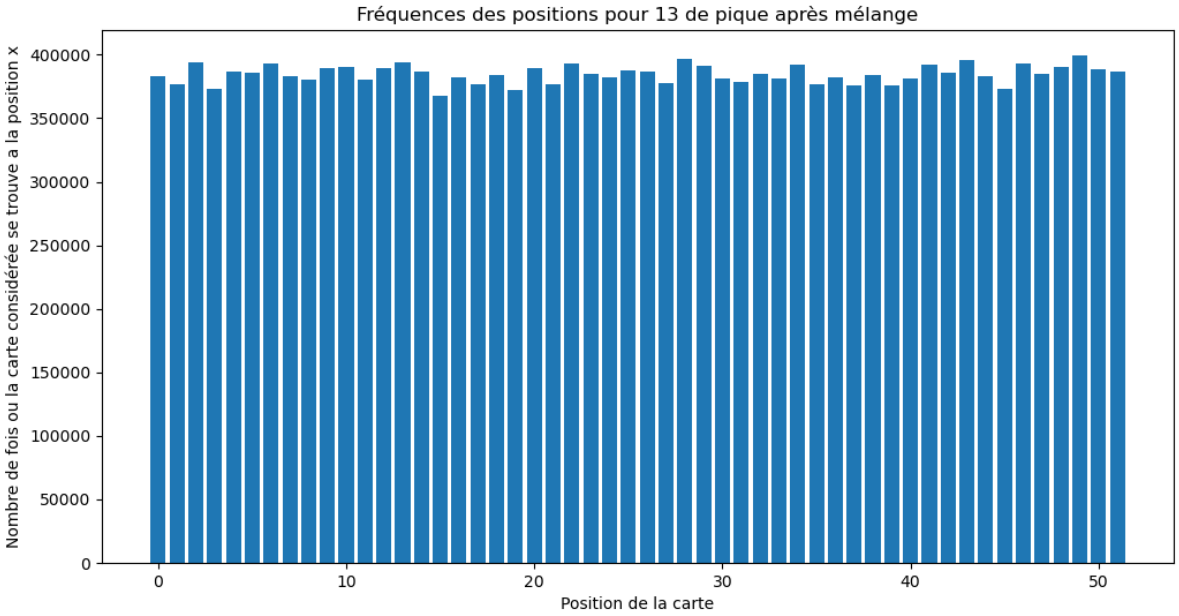

Figure 4: Distribution du roi de pique (carte n°13)

8 Conclusion

Pour conclure, grâce à notre algorithme, nous sommes capables de retrouver une graine en un temps acceptable pour une partie de poker (± 2) dès lors qu'il nous manque moins de 3 groupes de 6 bits d'information. Ce cas se produit environ 62% du temps et dès qu'il s'est produit, nous sommes capables de prédire toutes les mains de tous les joueurs présents dans notre partie de poker à toutes les manches. Cependant, nous avons fait l'hypothèse que nous voyons les mains de 3 joueurs. Cela n'est pas nécessaire car il suffit d'interpréter l'absence d'information comme étant un rejet. Cependant, nous avons tout de même besoin de connaître le mélange utilisé pour la distribution des cartes ainsi que l'ordre dans lequel sont distribuées les cartes.

References

[1] Coulom, P. R., and Jongwane, S. G. Le jeu de go et la révolution de monte carlo.

[2] Knuth, D. E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, third ed. Addison-Wesley, Boston, 1997.

[3] Welzl, E. Smallest enclosing disks (balls and ellipsoids). In *New Results and New Trends in Computer Science* (Berlin, Heidelberg, 1991), H. Maurer, Ed., Springer Berlin Heidelberg, pp. 359–370.