Rapport Projet Mastermind Info CPP S4

Ghadi ABDALLAH
Houssam OUAALITI

2024

NB: Tous les codes sont joints dans les autres fichiers, tous les tests sont dans le fichier tests_personnels.py et tous les histogrammes sont dans histogrammes.py. Les explications de nos diverses fonctions ne seront pas détaillées dans notre compte-rendu, en raison qu'ils ont tous été commentés en détails durant l'implémentation dans leurs fichiers correspondants.

Introduction:

Le jeu Mastermind est un jeu de société, de réflexion et de déduction, créé par le Mordecai Meirowitz dans les années 1970. C'est un jeu entre deux joueurs, dont un des deux doit tenter de retrouver le code établi par l'autre avec un nombre limité de tentatives. Le code est composé de 4 ou 5 couleurs différentes et normalement, le "codebreaker" (la personne qui essaie de casser le code) a le droit à 10 ou à 12 tentatives. Après chaque tentative, le "codemaker" (qui met en place le code) répond en indiquant le nombre de couleurs bien placées et mal placées.

Figure 1 : Exemple de plateau Mastermind

Le but de ce projet est d'implémenter des programmes qui jouent les "codebreaker" et "codemaker" du jeu Mastermind, avec diverses stratégies pour chacun. Nous allons également modéliser ce jeu par une interface graphique correspondante, qui permettra d'effectuer des jeux entre humains, robots, ou même un mélange des deux. Notre "codebreaker" devra donc deviner la combinaison choisie par le "codemaker", en fonction des évaluations données par celui-ci.

Questions :

Question 1 :

L'implémentation commentée de notre fonction d'évaluation se trouve dans le fichier common.py. Cette fonction prend pour arguments la combinaison qu'on veut étudier, et la combinaison avec laquelle on veut comparer la première. La fonction renvoie un tuple contenant le nombre de plots bien placés puis celui des plots mal placés. Afin de tester la correction de ce programme, nous avons effectué nos propres tests, en plus du fichier selftest.py, sur des exemples précis qu'on donne dans la première cellule du fichier tests_personnels.py. Nous remarquons que pour les quatre combinaisons et les quatre références données, notre fonction renvoie des évaluations cohérentes. Nos tests étaient basés sur des cas où nous trouvons des répétitions de certaines couleurs, et des cas où nos couleurs sont toutes distinctes.

```
# Combinaisons choisies
c1 = 'RVVR'
c2 = 'RBOG'
c3 = 'JNMN'
c4 = 'OMRJ'
c5 = 'RRRR'
c = [c1, c2, c3, c4, c5]

# Combinaisons de référence choisies
cr1 = 'RVBJ'
cr2 = 'NOMG'
cr3 = 'NNOM'
cr4 = 'RVBJ'
cr5 = 'BRRJ'
cr = [cr1, cr2, cr3, cr4, cr5]
```

Figure 2 : Créations des tests

```
#Evaluations
for i in range(len(c)):
    tot_eval = common.evaluation(c[i], cr[i])
    print("évaluation ({}, {}) : {}".format(c[i], cr[i], tot_eval))
# affiche : évaluation (RVVR, RVBJ) : (2, 0)
#           évaluation (RBOG, NOMG) : (1, 1)
#           évaluation (JNMN, NNOM) : (1, 2)
#           évaluation (OMRJ, RVBJ) : (1, 1)
#           évaluation (RRRR, BRRJ) : (2, 0)
```

Figure 3 : Résultats obtenus, cohérents

Question 3 :

Chaque essai peut être considéré comme un événement de Bernoulli, avec deux résultats possibles : succès (trouver la bonne combinaison) ou échec (ne pas trouver la bonne combinaison). La probabilité de succès dans chaque essai est déterminée par la formule suivante, vue en cours de probabilités :
$p = (1 \div len(COLORS))^{LENGTH} = (1/8)^4$ avec len(COLORS) le nombre de couleurs qu'on a (ici, 8), et LENGTH la taille des combinaisons (ici, 4).

Et comme la partie devient donc une variable aléatoire de type géométrique de paramètre p, notre cours de probabilité nous permet aussi de calculer l'espérance, donnée par :
$E(x) = 1 \div p = 8^4 = 4096.$

Pour quantifier l'efficacité de la version 0 du "codebreaker", nous avons mis en place une simulation de parties de Mastermind. Dans cette version, le "codebreaker" génère des combinaisons aléatoires jusqu'à ce qu'il trouve la bonne combinaison. Nous avons simulé un

grand nombre de parties, enregistrant à chaque fois le nombre d'essais nécessaires pour trouver la bonne combinaison. Nous avons répété cette simulation un certain nombre de fois pour obtenir des données significatives.

En analysant les résultats de la simulation, nous avons calculé l'espérance du nombre d'essais nécessaires pour chaque partie. Cette espérance représente le nombre moyen d'essais que le “codebreaker” a effectué avant de trouver la bonne combinaison. Nous avons utilisé cette mesure pour évaluer l'efficacité de la version 0 du “codebreaker”. En effet, un nombre d'essais élevé indique une faible efficacité, car cela signifie que le “codebreaker” a besoin de plus de tentatives pour trouver la bonne combinaison.

Pour mesurer cette espérance, nous avons fait des tests suivant différents nombres de parties. La figure 4 représente les cas obtenus avec 1000 parties, alors que la figure 5 représente les cas obtenus avec 5000 parties. Nous remarquons une forme exponentielle décroissante du nombre des parties de Mastermind qui nécessitent x nombres d’essais avant d’aboutir à la solution correcte. Le premier cas (1000 essais) nous rend une espérance (moyenne, avec np.mean) de 4131 essais, alors que le deuxième cas (5000 essais) nous rend une espérance de 4089 essais.

Nous remarquons donc que plus on simule des parties plus notre espérance s’approche de 4096. Nous pouvons conclure donc que cette première version possède bien une espérance cohérente avec notre calcul analytique. Ces graphiques on étés mis en place dans notre fichier histogrammes.py.

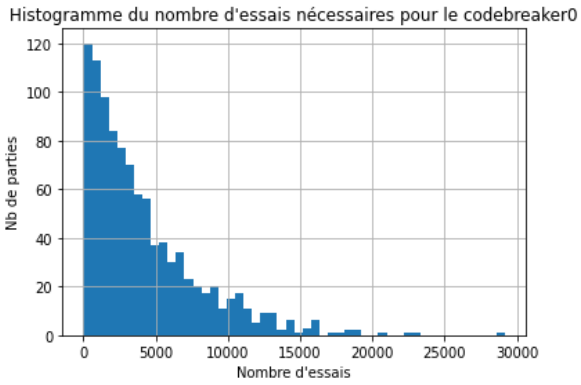

Figure 4 : codebreaker0 pour 1000 essais

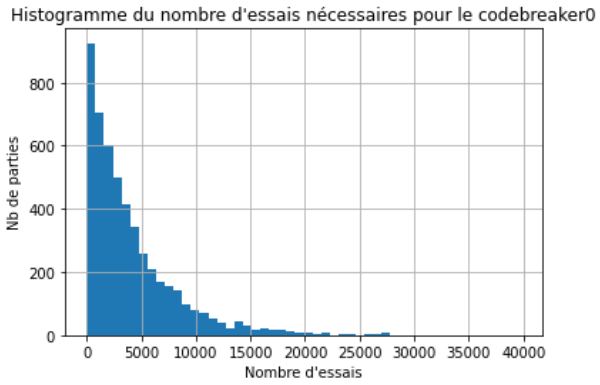

Figure 5 : codebreaker0 pour 5000 essais

Question 4 :

Après avoir implémenté le codebreaker1, une nouvelle version qui prend en compte les essais déjà utilisés, on peut anticiper un avantage significatif. En effet, cette version devrait garantir que le nombre d'essais ne dépasse jamais 4096.

En effet, ici, comme nous évitons les répétitions, tout en faisant des choix aléatoires, nous nous trouvons dans le cas d’une variable aléatoire de type uniforme. Ainsi, on a :
$P(X = x) = 4095 \div 4096 + \ldots + (4096 - k - 1) \div (4096 - k) + \ldots + 1 \div 4096 \approx 1 \div 4096$
Ainsi, nous pouvons calculer l’espérance de ce “codebreaker” de la forme suivante :

$$E(X) = \sum_{k=1}^{4096} (k \div 4096) = 4096 \times 4097 \div (2 \times 4096) = 4097 \div 2 = 2048.5$$

En simulant 1000 parties, nous obtenons une espérance de 2050.9 avec l'histogramme de la figure 6. La figure 6 représente une comparaison des parties des deux "codebreakers". La comparaison nous permet de voir que le codebreaker1 est beaucoup mieux que son prédécesseur, résultat déjà prévu, et par calcul d'espérance, nous retrouvons un gain de 2047.5 essais en moyenne entre les deux "codebreakers". L'histogramme de la figure 7 représentant cette comparaison permet de voir la forte concentration pics bleus (codebreaker1) au début, sans ayant une forme spécifique, et les pics oranges décroissantes exponentiellement du codebreaker0.

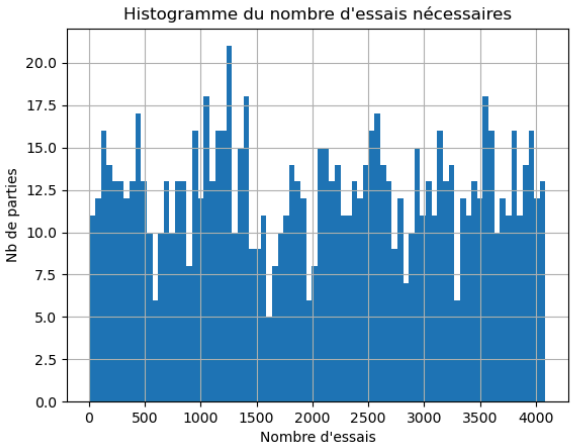

Figure 6 : codebreaker1 pour 1000 parties

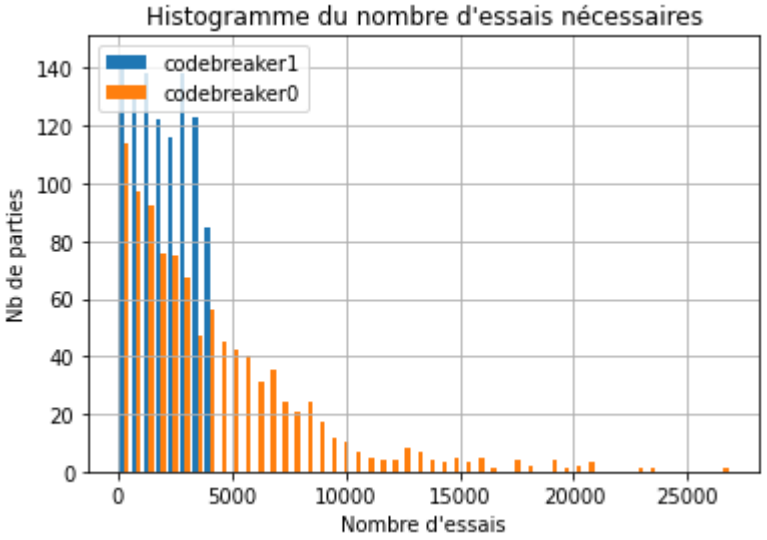

Figure 7 : Comparaison des deux codebreakers

Question 7 :

Dans cette nouvelle version, le "codebreaker" maintient constamment une liste des solutions possibles et la met à jour après chaque évaluation. De cette manière, il réduit

progressivement son ensemble de combinaisons potentielles. À chaque étape, il choisit aléatoirement une combinaison parmi cet ensemble restreint jusqu'à ce qu'il trouve la combinaison gagnante. En effectuant 2000 parties, nous retrouvons une espérance de 5.43 et l'histogramme de la figure 8 suivante :

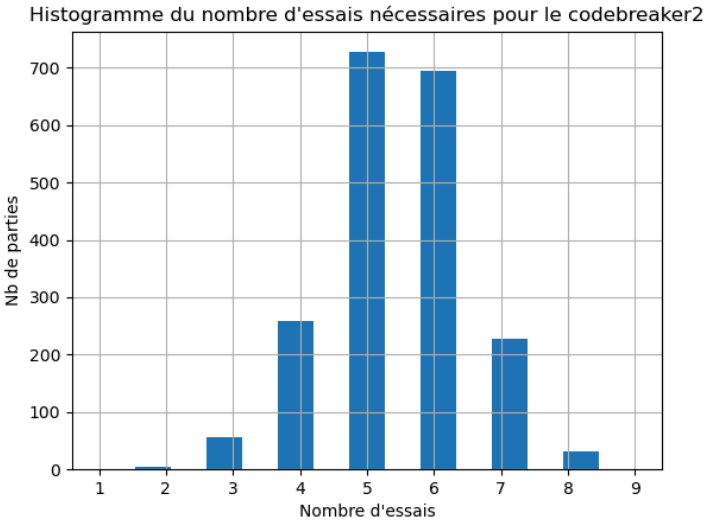

Figure 8 : codebreaker2 pour 2000 parties

Nous arrivons donc à remarquer une très grande différence entre ce "codebreaker" qui ne nécessite pas plus que huit essais pour retrouver une solution, et le précédent, qui nécessitait des milliers. Nous retrouvons un très fort gain d'environ 2043 essais par partie en moyenne. En essayant de modéliser cela dans un histogramme (figure 8), nous observons une très claire différence à un tel point que l'histogramme nous donne un seul pic bleu pour le codebreaker2, et pleins de pics oranges très faibles de façon presque constante (horizontale) pour le codebreaker1.

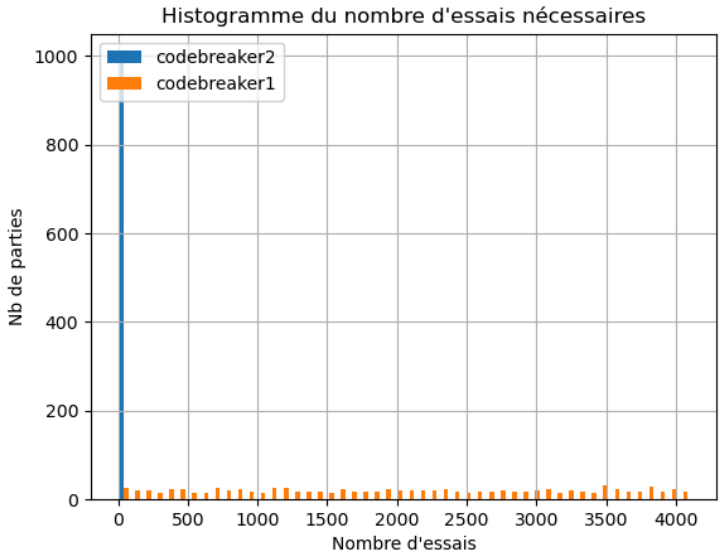

Figure 9 : Comparaison codebreaker1 et codebreaker2

Question 8 :

Dans cette question, on tente de faire durer plus le jeu, même en utilisant une façon malhonnête. Nous voulons changer notre combinaison cible après chaque tour afin de maximiser la complexité du jeu pour le “codebreaker” d’une manière stratégique. Cependant, cela doit être sans créer une incohérence avec les évaluations précédentes. Pour cela, nous avons décidé de passer à chaque fois par la cible qui nous permettra d’avoir le plus de solutions possibles restantes après chaque tentative.

Pour implémenter cela, on initialise d’abord notre liste de possibilités restantes (possibilités_encore) par les 4096 combinaisons possibles à partir de laquelle on choisit une à l’aléatoire. Ensuite, après la proposition du “codebreaker”, notre “codemaker” l’évalue par rapport à chaque combinaison possible restante. Il parcours tous les cas possibles et calcule le nombre de possibilités restantes pour chaque combinaison après avoir pris en compte l’évaluation qu’on aura. Il sélectionne ainsi comme nouvelle combinaison cible celle qui laisse le plus de possibilités restantes et on met à jour enfin la liste des combinaisons possibles en fonction de l’évaluation de la proposition du “codebreaker”, qui sera retournée

En le faisant jouer contre le codebreaker2 pour 10 parties (nous ne pouvons pas mettre trop de parties, sinon le test prendra longtemps pour s’effectuer), nous obtenons une espérance d’environ 7, et nous obtenons l’histogramme suivant du jeu :

.

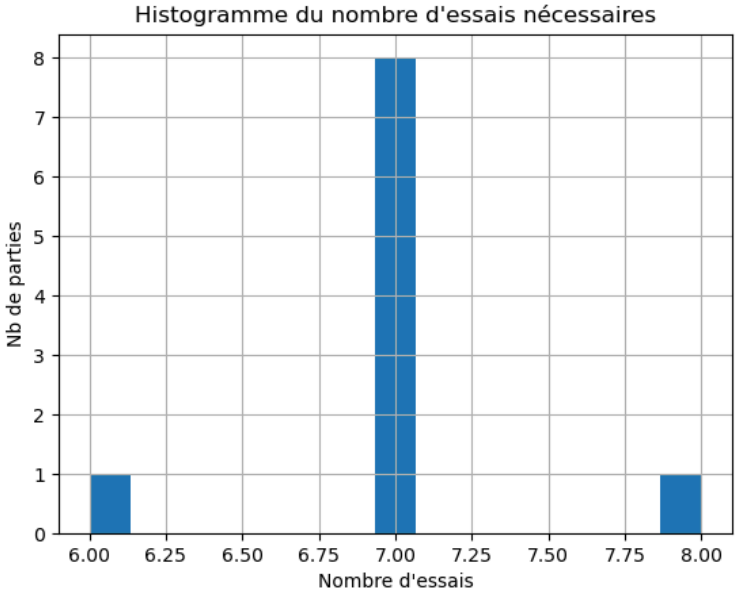

Figure 10 : codemaker2 vs codebreaker2 pour 10 parties

Nous remarquons un faible déplacement vers la droite par rapport au codemaker1, et donc une petite difficulté en plus pour notre “codebreaker”. Ainsi, nos obstacles rajoutés étaient successifs en compliquant le travail du “codebreaker”. Pour mieux représenter cela, nous avons effectué un histogramme de comparaison entre les deux “codemakers” (figure 11), pour 10 parties, et contre le même codebreaker2.

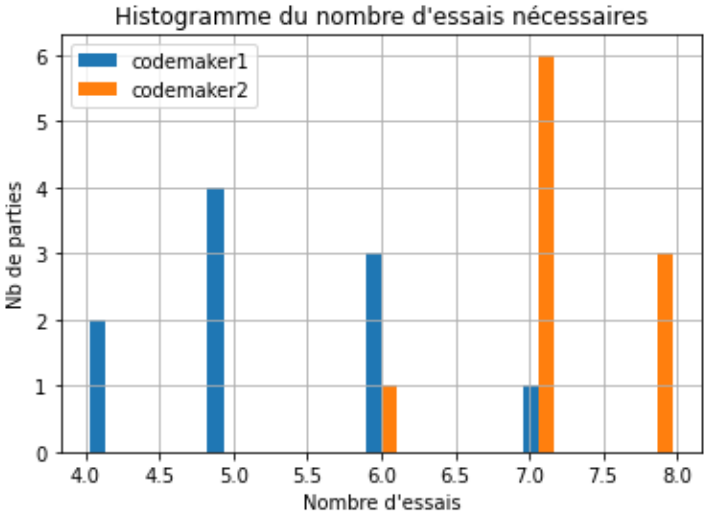

Figure 11 : Comparaison entre les codemakers 1 et 2

Question 11 :

Ce “codebreaker”, à l’opposé du 2, ne choisit pas par l’aléatoire, mais recherche sur une zone plus grande. En effet, il est capable d’étudier même les cas qui ne sont pas restants, pour chercher plus d’informations, bien que ces combinaisons soient fausses. Par exemple, si on a une solution de la forme « RBGM » et qu’on sait déjà qu’il y a un seul du rouge, un seul du vert et un seul du bleu dans le bon ordre, il serait plus intelligent d’utiliser des combinaisons avec d’autres couleurs à part celle qu’on connaisse déjà puisqu’ils ne donneront aucune information supplémentaire.

Notre code réduit l’ensemble des possibilités au fur et à mesure des étapes, mais en utilisant nos fonctions supplémentaires commentées dans le fichier common.py, nous arrivons à retrouver la combinaison pour laquelle on élimine le moins de cas. Ensuite, à l’intérieur de ce cas, nous recherchons le meilleur cas, et donc celui qui élimine le plus de combinaison (à l’intérieur du pire cas), et c’est cette combinaison qu’on étudiera.

Question 12 :

Nous avons simulé ce code pour plusieurs parties de taille de COLORS différentes, et voici les résultats que nous avons réussi à obtenir (malgré que ca prend un bon temps) :

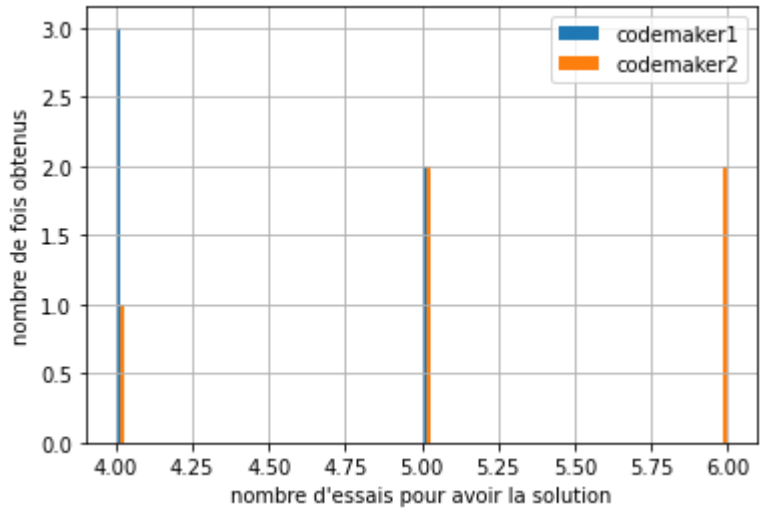

Figure 12 : codebreaker3 vs codemaker1 et codemaker2

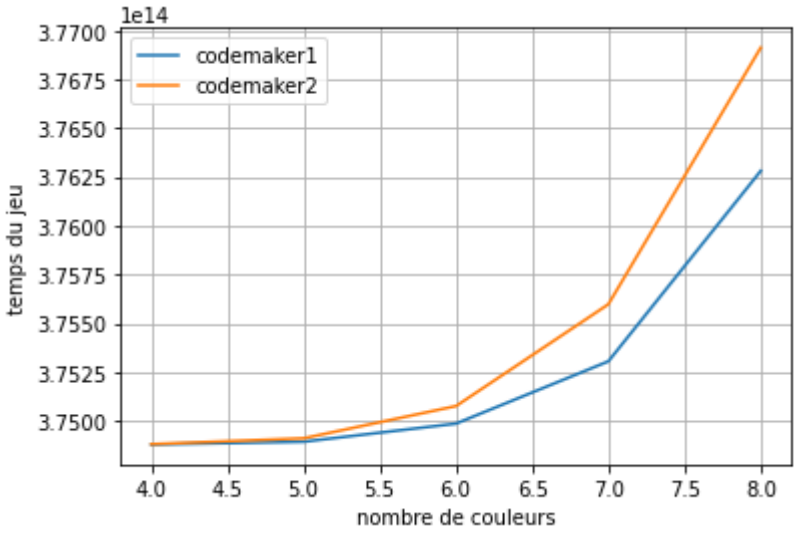

Figure 13 : codebreaker3 vs codemaker1 et codemaker2

La figure 12 nous permet de voir le nombre d'essais nécessaire et la figure 13 forme l'étude en fonction du temps. Malheureusement, nous avons fait une erreur dans la figure 12 en paniquant, et comme ça prend longtemps nous n'avons plus le temps de réessayer de la faire pour la bonne réponse. Mais pour avoir la bonne idée nous avons appliqué la comparaison dans notre fichier histogrammes.py.