

Informatique (S4) - Projet Mastermind

Équipe pédagogique Info CPP

Contact : antoine.frenoy@univ-grenoble-alpes.fr

2021

Ce projet est à réaliser en équipes de 2 étudiants¹. Les questions sont à traiter en parallèle de l'implémentation. Le rendu final consiste en une archive `Nom1_Nom2_numérogroupeTP.zip` contenant vos programmes en langage Python, votre rapport au format pdf, et un README si nécessaire (détaillant le contenu de chaque fichier, expliquant comment utiliser vos programmes, etc). Cette archive est à rendre pour le 25 avril 2021 à 23h59, sur [Teide](#) impérativement.

Prêtez une attention particulière à la rédaction du rapport et à la mise en forme des graphiques. On attend une analyse critique (et non une simple description) des résultats de vos programmes. Tout le code écrit pour répondre aux questions doit être rendu et documenté. Vos fonctions et programmes doivent être testés systématiquement.

Ce sujet est disponible sur <https://perso.crans.org/frenoy/infoCPPS4/projet.pdf>, et sera si nécessaire mis à jour pour répondre à d'éventuels signalements d'erreurs ou demandes de clarification.

La [charte contre la fraude](#) proposée par l'Ensimag s'applique. Notamment la copie ou le partage de programmes ou de réponses avec un autre binôme sont strictement interdits et seront sévèrement sanctionnés. De même, les règles habituelles concernant le plagiat s'appliquent, et sont notamment résumées dans la [charte anti-plagiat](#) adoptée par l'UGA et Grenoble INP.

Il n'est pas nécessaire de traiter toutes les questions pour obtenir une bonne note. En particulier, la section indiquée comme *avancée* est destinée aux étudiants les plus intéressés.

Introduction

Vous connaissez probablement le jeu du [mastermind](#). Si ce n'est pas le cas, familiarisez-vous avec ce jeu en lisant [ce tutoriel](#) et en jouant contre un ordinateur (par exemple [ici](#)).

Le but de ce projet est d'implémenter en Python des programmes qui jouent le *codebreaker* et le *codemaker*, avec une stratégie plus ou moins intelligente pour le premier et plus ou moins honnête pour le second.

Détails techniques, structure des programmes, exemple minimal

Une combinaison est une chaîne de `LENGTH` caractères, les caractères autorisés étant listés dans `COLORS`. Une même couleur peut être présente plusieurs fois.

Le *codemaker* choisit la combinaison à faire deviner. Pour chaque combinaison proposée par le *codebreaker*, il donne l'évaluation de cette combinaison. L'évaluation est un couple d'entiers, indiquant respectivement le nombre de plots de la combinaison proposée présents dans la solution à la bonne position, et le nombre de plots présents mais à la mauvaise position.

Le *codebreaker* propose des combinaisons, choisies en fonction des évaluations précédemment données par le *codemaker*.

1. Éventuellement quelques équipes de 3 après validation par l'enseignant en cas de groupes de TD impairs.

On fournit l'architecture globale du jeu, avec un embryon de codebreaker et de codemaker, à télécharger [ici](#).

- `common.py` contient la définition des variables `LENGTH` et `COLORS`. On y ajoutera (en suivant rigoureusement l'énoncé) des fonctions qui peuvent être utilisées à la fois par le codemaker et le codebreaker.
- `codemaker0.py` est une implémentation *partielle* du codemaker, qui ne compte que les plots présents à la bonne position.
- `codebreaker0.py` est une implémentation rudimentaire du codebreaker qui essaie des combinaisons au hasard jusqu'à trouver la bonne.
- `play.py` lance ces versions 0 du codemaker et du codebreaker et les fait jouer ensemble, en utilisant une fonction `play` qui peut faire jouer n'importe quelle version du codemaker contre n'importe quelle version du codebreaker. Ce programme peut facilement être modifié pour faire jouer ensemble d'autres versions du codebreaker et du codemaker. Pour vous aider à tester vos programmes, il contient aussi une fonction `play_human_against_codemaker` qui fait jouer un joueur humain dans le rôle du codebreaker (dont les propositions sont lues au clavier) contre le codemaker donné en argument, ainsi qu'une fonction `play_human_against_codebreaker` (idem mais l'humain joue le codemaker).

Il est nécessaire de bien comprendre ce fonctionnement avant d'attaquer la suite du projet. Si ce n'est pas le cas, demandez de l'aide à vos enseignants.

L'essentiel du projet consistera en l'implémentation de versions plus avancées du codemaker et codebreaker. **Il est essentiel de respecter parfaitement les spécifications** (noms des programmes, noms des fonctions et leurs arguments, etc) imposés par l'énoncé. Vos nouvelles versions doivent ainsi pouvoir s'interfacer avec les versions 0 données du codemaker et codebreaker (vous pouvez utiliser et adapter le code proposé dans ces versions 0). Notamment, vous ne pouvez pas changer les arguments des fonctions `codebreaker` / `codemaker` et `init`, qui sont les seules fonctions à travers lesquelles la boucle principale de jeu dans `play.py` communique avec le codemaker et le codebreaker. La fonction `play` ne doit donc pas être modifiée pour faire fonctionner les codebreaker et codemaker que vous écrirez dans ce projet.

Première version fonctionnelle

La version 0 du codemaker n'est pas entièrement fonctionnelle puisqu'elle ne compte pas les plots existants mais mal placés.

Question 1 — Écrivez dans `common.py` une fonction `evaluation` qui évalue une combinaison donnée en premier argument en la comparant à la combinaison de référence donnée en deuxième argument². L'évaluation est renvoyée comme un couple de deux entiers (nombre de plots présents bien et mal placés respectivement).

Question 2 — Implémentez (dans `codemaker1.py`) une version 1 du codemaker qui évalue correctement les combinaisons proposées.

Un peu de mémoire

La version 0 du codebreaker n'est pas très efficace puisqu'elle essaie des combinaisons aléatoires. Avant de commencer à l'améliorer, commençons par quantifier son efficacité.

Question 3 — Quelle est l'espérance du nombre d'essais faits par cette version avant de trouver

2. N'oubliez pas de tester en profondeur cette fonction. Une erreur dans cette fonction de base sera probablement la cause de programmes faux pour tout le reste du sujet.

la bonne solution (en fonction de `LENGTH` et de la longueur de `COLORS`) ? Lancez un grand nombre de parties et produisez un histogramme du nombre d'essais nécessaires³ (pour les valeurs de `LENGTH` et `COLORS` définies dans `common.py`), comparez avec votre résultat analytique.

En tirant des combinaisons aléatoires, il est probable que le codebreaker essaie plusieurs fois la même combinaison.

Question 4 — Quel gain attendez-vous (en terme d'espérance du nombre d'essais faits pour trouver la bonne solution) si le codebreaker choisit aléatoirement une combinaison *pas encore essayée* ? Implémentez (dans `codebreaker1.py`) cette version 1 du codebreaker et testez votre prédiction : produisez un graphique de votre choix permettant de visualiser le gain en nombre d'essais⁴.

Pour avoir une mémoire entre différents appels à la fonction `codebreaker`, vous pouvez utiliser des variables globales, de la même façon que le codemaker utilise la variable globale `solution` pour se souvenir de la solution choisie. Il est important de bien comprendre le fonctionnement des variables globales (et ce qui se passerait si ces variables n'étaient pas déclarées globales). Si nécessaire, on propose un exemple simple [ici](#).

Un embryon de logique

Même en gardant en mémoire les combinaisons déjà essayées, l'essai par force brute⁵ est très inefficace. Avec 4 plots et 8 couleurs, un joueur humain non expert est généralement capable de trouver la combinaison en une dizaine d'essais. Pour cela une stratégie très simple consiste à n'essayer que des combinaisons possibles au vu des évaluations des combinaisons précédemment proposées.

Question 5 — Écrivez dans `common.py` une fonction `donner_possibles` qui détermine l'ensemble des combinaisons possibles (renvoyé comme une variable de type `set`) après un unique essai (les arguments sont la combinaison testée et l'évaluation associée, dans cet ordre).

Chaque étape (proposition d'une solution et obtention d'une évaluation) supprime un certain nombre de combinaisons précédemment possibles mais n'ouvre jamais de nouvelles possibilités. Le codebreaker pourrait donc maintenir une variable de type `set` indiquant l'ensemble des combinaisons encore possibles.

Question 6 — Écrivez dans `common.py` une fonction `maj_possibles` qui met à jour cette variable, prenant en arguments, dans cette ordre, l'ensemble des combinaisons encore possibles avant ce nouvel essai, la combinaison testée, et l'évaluation associée. Plutôt que de renvoyer une nouvelle variable de type `set`, la variable donnée en premier argument sera directement modifiée⁶.

Question 7 — Implémentez (dans `codebreaker2.py`) une version 2 du codebreaker qui n'essaie que des combinaisons possibles au vu des évaluations précédentes, en choisissant aléatoirement parmi toutes les possibilités. Cette version est-elle plus efficace (comparée à la version 1) ? Justifiez votre réponse avec un graphique.

3. Le code utilisé pour réaliser ce grand nombre de parties et produire cet histogramme doit bien sûr être documenté et être inclus dans le rendu.

4. Choisissez le nombre de parties à lancer selon la rapidité de votre programme.

5. C'est comme ça qu'on appelle l'essai exhaustif de toutes les solutions.

6. Si vous vous demandez comment une fonction peut modifier une variable donnée en argument, rappelez-vous que la situation est différente pour un entier ou pour des objets plus complexes comme `list` ou ici `set`.

On a maintenant une stratégie qui fonctionne relativement bien, similaire à la stratégie intuitivement adoptée par beaucoup de joueurs humains.

Un soupçon de rouerie

Dans la version analogique du jeu, le codemaker réalise physiquement sa combinaison sur le plateau de jeu⁷. Cela permet d'éviter que le codemaker ne change de combinaison en cours de jeu pour rallonger la partie.

Autorisons le codemaker à changer sa combinaison cible quand il le souhaite, à condition de respecter les évaluations déjà données⁸. L'objectif du codemaker est de faire durer la partie le plus longtemps possible.

Question 8 — Réalisez (dans `codemaker2.py`) une version 2 du codemaker aussi performante que possible, en laissant libre cours à votre imagination. Décrivez votre raisonnement et votre démarche⁹. Mesurez sa performance en la faisant jouer contre la version 2 du codebreaker, et produisez un graphique illustrant les performances de ce nouveau codemaker comparé à la version 1 (qui ne changeait pas de combinaison en cours de partie).

Une dose d'optimal (avancé)

Pour l'instant, notre codebreaker choisit une combinaison à tester aléatoirement parmi toutes les combinaisons possibles (au vu des évaluations précédentes).

Question 9 — Pensez-vous qu'il est toujours préférable pour le codebreaker de tester seulement des combinaisons encore possibles au vu des évaluations précédentes ? Autrement dit, pouvez-vous trouver une situation où essayer une combinaison qu'on sait déjà impossible donne plus d'information qu'essayer une des combinaisons encore possibles ? Le cas échéant, **donnez un exemple précis**.

L'idée d'un codebreaker optimal serait d'essayer la combinaison dont l'évaluation lui rapportera le plus d'informations *dans le pire des cas*. Ce pire des cas s'obtient notamment en considérant la rouerie maximale du codemaker qui de son côté cherche à donner le moins d'information possible.

Question 10 — Implémentez (dans `codebreaker3.py`) un tel codebreaker optimal. Faites le jouer contre les versions 1 et 2 du codemaker, commentez vos résultats (sans oublier de vous appuyer sur des graphiques pertinents)¹⁰.

Analyse de parties

Au lieu (ou en plus) de l'affichage à l'écran, on aimerait pouvoir obtenir un *log* d'une partie lancée par `play.py` dans un fichier texte. Ce log sera une succession de lignes qui sont soit une proposition de combinaison soit une évaluation, en suivant strictement la syntaxe donnée dans l'exemple `log0.txt`.

Question 11 — Écrivez dans `play.py` une nouvelle version de la fonction `play` (que vous

7. En la gardant cachée pour le codebreaker jusqu'à la fin de la partie.

8. Autrement dit il peut tricher tant que le codebreaker ne peut pas s'en rendre compte.

9. N'hésitez pas à donner des exemples de parties pour illustrer votre propos.

10. Si votre *codebreaker* est trop lent, vous pouvez diminuer dans `common.py` le nombre de couleurs possibles ou la longueur de la combinaison.

nommerez `play_log`) qui crée ce log dans un fichier texte dont le nom est fourni dans un troisième argument.

Question 12 — Écrivez un programme `check_codemaker.py` qui à partir du log d’une partie terminée¹¹ (donné en argument) vérifie que le codemaker n’a pas triché de façon visible¹².

Et pour finir

Cette question n’a pas d’incidence sur votre note, mais nous permettra d’adapter la difficulté et l’organisation du projet l’an prochain.

Question 13 — Combien de temps estimez-vous avoir passé sur ce projet (séances en présentiel incluses) ? Quelles sont les principales difficultés que vous avez rencontrées ? Comment avez-vous réparti et organisé le travail au sein du binôme (répartition des tâches, communication, solutions pour le partage de code et le suivi de version...) ? Avez-vous l’impression que le travail en équipe a été un gain de temps ou une complication ?

11. Qui finit donc par l’évaluation `LENGTH,0`.

12. C’est à dire n’a pas changé de combinaison d’une façon qui invaliderait des évaluations précédemment données.