Implémentation progressive de stratégies intelligentes pour le jeu Mastermind : du hasard pur à une approche minimax, avec analyse expérimentale et visuelle des performances

Bruyere Nathan

Étudiant athlète de haut-niveau à Grenoble INP - Génie Industriel

Abstract

Mastermind[1] est un jeu combinatoire simple en apparence, mais qui soulève des questions algorithmiques riches. Ce projet propose une exploration progressive de différentes stratégies pour le *codebreaker* et le *codemaker*, allant du hasard pur à des approches informées et optimales au sens du pire cas.

À partir d'une stratégie naïve par force brute, nous introduisons des mécanismes de mémoire et de filtrage logique permettant de réduire l'espace des solutions compatibles avec les évaluations précédentes, ce qui conduit à une amélioration drastique des performances. Nous étudions ensuite un *codemaker* tricheur, autorisé à modifier dynamiquement sa combinaison cible tant que les évaluations fournies restent cohérentes, afin de se placer systématiquement dans un scénario défavorable pour le *codebreaker*.

Dans ce contexte de pire cas, nous implémentons un *codebreaker* fondé sur un critère minimax. Les résultats expérimentaux, appuyés par des visualisations adaptées, mettent en évidence à la fois l'efficacité et le caractère fortement déterministe de cette stratégie face à un codemaker tricheur.

Au-delà du jeu lui-même, ce projet illustre des concepts clés de conception algorithmique.

Table des matières

1. Introduction

Mastermind est un jeu de réflexion combinatoire dans lequel un joueur, appelé *codebreaker*, cherche à deviner une combinaison secrète choisie par un adversaire, le *codemaker*. À chaque tentative, le *codebreaker* reçoit une évaluation partielle indiquant combien d'éléments de sa proposition sont corrects et bien placés, et combien sont corrects mais mal placés. Derrière des règles simples, ce jeu met en jeu des problématiques algorithmiques profondes, modélisables facilement sous divers langages de programmation, nous utiliserons `Python`.

Ce projet propose une exploration progressive de différentes stratégies algorithmiques appliquées à *Mastermind*. Dans un premier temps, nous considérons des stratégies naïves du *codebreaker*, reposant sur le hasard et la force brute, afin d'établir une base de comparaison théorique et expérimentale. Nous montrons rapidement les limites de ces approches.

Nous introduisons ensuite des stratégies plus informées, dans lesquelles le *codebreaker* conserve en mémoire l'ensemble des combinaisons encore compatibles avec les évaluations précédentes. Cette approche, proche de l'intuition adoptée par les joueurs humains, permet une réduction drastique de l'espace des solutions possibles et conduit à des performances nettement supérieures.

Afin d'aller plus loin, nous remettons en question une hypothèse implicite du jeu classique : l'honnêteté du *codemaker*. Nous autorisons celui-ci à modifier dynamiquement

1. Mastermind est un jeu de réflexion combinatoire largement étudié dans la littérature algorithmique et ludique. Vous trouverez plus d'informations ici .

sa combinaison cible, tant que les évaluations fournies restent cohérentes avec l'historique de la partie. Ce cadre adversarial met en évidence les limites des stratégies intuitives et motive la recherche de politiques plus robustes.

Enfin, nous implémentons une stratégie de *codebreaker* fondée sur un critère **minimax**, visant à maximiser l'information obtenue dans le scénario le plus défavorable possible. Cette approche, proche des stratégies optimales connues de la littérature, permet d'analyser finement les compromis entre optimalité, coût de calcul et caractère déterministe des décisions.

2. Architecture et moteur du solveur

Le projet repose sur une architecture modulaire visant à séparer le moteur de jeu des différentes stratégies implémentées. Cette organisation permet de comparer facilement plusieurs approches algorithmiques, sans modifier la logique centrale du jeu.

Le cœur du moteur est implémenté dans le fichier `play.py`. Il définit une fonction générique `play()` chargée d'orchestrer une partie entre un *codebreaker* et un *codemaker*. Cette fonction ne dépend d'aucune stratégie particulière : elle appelle, à chaque tour, les fonctions `codebreaker` et `codemaker` fournies en argument, tout en gérant l'initialisation, le déroulement de la partie et le critère d'arrêt.

Les paramètres globaux du jeu (taille des combinaisons, ensemble des couleurs autorisées) ainsi que les fonctions utilitaires partagées sont regroupés dans le fichier `common.py`.

Les différentes stratégies de *codebreaker* et de *codemaker* sont implémentées dans des fichiers distincts. Chaque module expose la même interface minimale : une fonction `init`, appelée en début de partie, et une fonction principale (`codebreaker` ou `codemaker`) appelée à chaque tour.

Enfin, un fichier dédié (`plot.py`, généré à l'aide de Chat GPT) regroupe les scripts d'analyse expérimentale et de visualisation.

Cette architecture simple mais rigoureuse, met en avant une séparation claire des responsabilités et facilite à la fois l'expérimentation, la comparaison des approches et l'extension à volonté du projet.

Notations et conventions.. Dans tout le rapport, une *combinaison* désigne une chaîne de caractères de longueur `LENGTH`, représentant la proposition du codebreaker ou la solution du codemaker. Chaque caractère correspond à une couleur choisie dans l'ensemble `COLORS`. Une même couleur peut apparaître plusieurs fois dans une combinaison. Les évaluations associées à une combinaison sont notées sous la forme (b, m), où b est le nombre de plots bien placés et m le nombre de plots corrects mais mal placés.

3. Premières stratégies naïves

3.1. Force brut sans mémoire

Nous commençons par établir une base de référence à l'aide d'une stratégie naïve, notée `codebreaker0`. Cette version du *codebreaker* propose à chaque tour une combinaison tirée uniformément au hasard parmi toutes les combinaisons possibles, sans utiliser aucune information issue des évaluations précédentes.

Analyse théorique

Soient :

— L la longueur de la combinaison,

— C le nombre de couleurs disponibles.

Le nombre total de combinaisons possibles est alors :

$$N = C^L$$

À chaque tentative, le *codebreaker* a une probabilité $1/N$ de proposer exactement la bonne combinaison. Le nombre d'essais nécessaires pour trouver la solution suit donc une loi géométrique de paramètre $p = 1/N$.

L'espérance du nombre d'essais avant succès est donnée par :

$$\mathbb{E}[T] = \frac{1}{p} = N = C^L$$

Dans notre configuration expérimentale ($L = 4$ et $C = 8$), on obtient :

$$\mathbb{E}[T] = 8^4 = 4096$$

Résultats expérimentaux

Afin de valider cette analyse théorique, nous avons simulé un grand nombre de parties opposant `codebreaker0` à un *codemaker* honnête. La Figure 1 présente l'histogramme du nombre d'essais nécessaires pour trouver la solution sur 1000 parties.

On observe une distribution fortement asymétrique, caractéristique de la loi géométrique. La majorité des parties se termine relativement tôt, mais une proportion non négligeable nécessite un nombre très élevé d'essais, ce qui explique la présence d'une longue queue à droite.

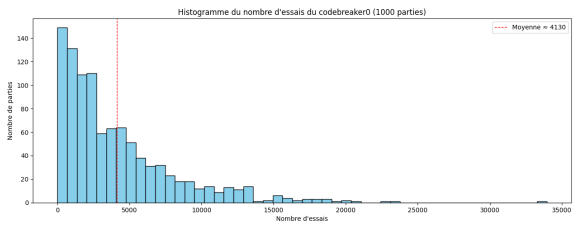

FIGURE 1: Histogramme du nombre d'essais de codebreaker0 sur 1000 parties

La moyenne expérimentale observée est proche de la valeur théorique attendue, confirmant la validité du modèle probabiliste. En revanche, la variance est très élevée :

certaines parties peuvent dépasser largement plusieurs dizaines de milliers d'essais, rendant cette stratégie impraticable en pratique.

Ces résultats illustrent clairement les limites d'une approche par hasard pur : bien que correcte en moyenne, elle est extrêmement inefficace et peu robuste, ce qui motive l'introduction de mécanismes de mémoire et d'exploitation de l'information.

3.2. Force brut avec mémoire

La stratégie `codebreaker1` constitue une première amélioration de `codebreaker0`. Ici, le *codebreaker* conserve en mémoire les combinaisons déjà testées et s'interdit de les rejouer. Les propositions sont donc tirées aléatoirement *sans remise* parmi l'ensemble des combinaisons possibles.

Gain théorique attendu.

Dans ce cas, le nombre d'essais nécessaires pour trouver la solution correspond à la position de la combinaison correcte dans une permutation aléatoire de l'ensemble des $N = C^L$ combinaisons possibles. La variable aléatoire suit alors une loi uniforme discrète sur $\{1, \ldots, N\}$, et son espérance est donnée par :

$$\mathbb{E}[T] = \frac{N+1}{2}$$

Pour $L = 4$ et $C = 8$, on obtient :

$$\mathbb{E}[T] = \frac{8^4+1}{2} \approx 2048$$

On s'attend donc à un gain théorique d'un facteur deux par rapport au tirage avec remise.

Résultats expérimentaux.

La Figure 2 confirme cette analyse. La moyenne expérimentale observée est très proche de la valeur théorique attendue, autour de 2048 essais.

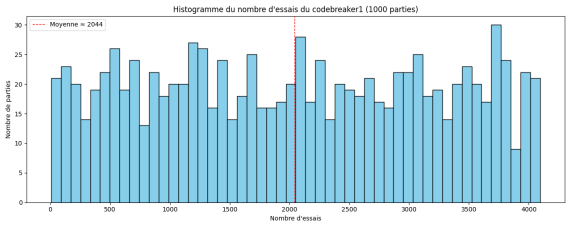

FIGURE 2: Histogramme du nombre d'essais de codebreaker1 sur 1000 parties

Cette stratégie reste fondamentalement inefficace en pratique, car elle n'exploite toujours aucune information fournie par les évaluations. Elle sert toutefois de point de référence important pour mesurer les gains apportés par les stratégies suivantes.

4. Vers un *codebreaker* intelligent

Les stratégies précédentes, bien qu'optimales au sens probabiliste pour un tirage aléatoire, n'exploitent aucune des informations fournies par les évaluations successives. Or, chaque évaluation permet en réalité d'éliminer un grand nombre de combinaisons incompatibles avec l'historique de la partie. La stratégie `codebreaker2` repose précisément sur cette idée centrale.

Le principe consiste à maintenir explicitement l'ensemble des combinaisons encore possibles au vu des évaluations précédentes. À chaque nouveau coup, le *codebreaker* ne propose qu'une combinaison appartenant à cet ensemble, et met à jour celui-ci après réception de l'évaluation. Cette logique s'appuie sur deux fonctions génériques définies dans `common.py` :

- `donner_possibles()`, qui détermine l'ensemble des combinaisons compatibles avec une unique proposition et son évaluation associée ;
- `maj_possibles()`, qui met à jour l'ensemble courant des solutions possibles en éliminant celles incompatibles avec une nouvelle évaluation.

Résultats expérimentaux.

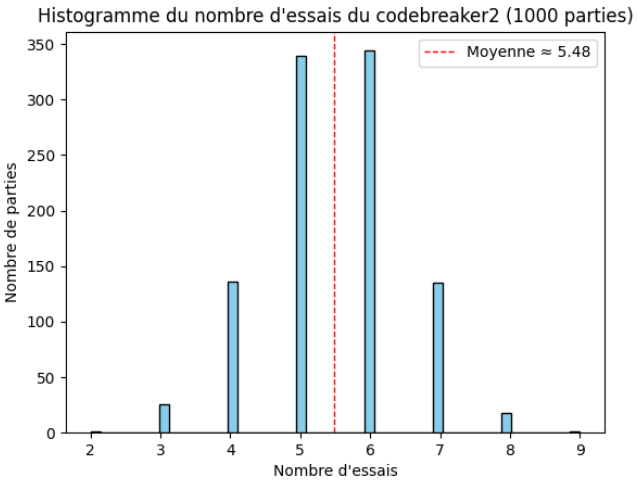

FIGURE 3: Histogramme du nombre d'essais de codebreaker2 sur 1000 parties

Les performances obtenues marquent une rupture nette avec les stratégies naïves. Comme l'illustre la Figure 3, le nombre d'essais chute brutalement : la moyenne observée est d'environ 5,5 essais, contre plusieurs milliers pour les versions aléatoires.

La distribution est fortement concentrée autour de quelques valeurs entières, avec une variance très faible. Cette efficacité remarquable s'explique par la réduction drastique de l'espace de recherche après chaque évaluation.

Ces résultats montrent que l'exploitation systématique des évaluations constitue le facteur déterminant de performance dans *Mastermind* !

5. Un *codemaker* tricheur

Jusqu'ici, le *codemaker* est supposé honnête : il choisit une combinaison secrète au début de la partie et s'y tient jusqu'à la fin. Dans la version physique de *Mastermind* rien n'empêche un *codemaker* de « déplacer »sa solution au cours de la partie, tant qu'il reste cohérent avec les évaluations déjà fournies.

Dans cette section, nous introduisons donc un *codemaker* adversarial (`codemaker2`) dont l'objectif est de faire durer la partie aussi longtemps que possible. Il est autorisé à changer de combinaison cible à chaque tour, à une condition unique : toutes ses réponses doivent rester compatibles avec l'historique de la partie [2].

5.1. Principe et stratégie

L'idée clé est de maintenir, côté *codemaker*, un ensemble de solutions encore possibles au vu des évaluations qu'il a lui-même renvoyées jusque-là. Au départ, cet ensemble correspond à toutes les combinaisons possibles. Lorsqu'une proposition g est faite par le *codebreaker*, le *codemaker* considère toutes les évaluations e qu'il pourrait renvoyer. Chaque évaluation e induit un sous-ensemble :

$$S_{g,e} = \{\, s \in S \mid \texttt{evaluation}(g, s) = e \,\}$$

où S désigne l'ensemble courant des solutions compatibles.

Un *codemaker* cherchant à minimiser l'information transmise adopte alors une stratégie gloutonne simple : choisir l'évaluation e qui maximise la taille de $S_{g,e}$, c'est-à-dire celle qui laisse le plus de solutions encore possibles après ce tour.

Cette stratégie est particulièrement intéressante car elle se place explicitement dans une logique de *pire cas* : elle force le *codebreaker* à progresser même lorsque l'adversaire répond de manière optimale pour ralentir la résolution.

5.2. Impact sur les performances

Nous mesurons l'effet de ce codemaker tricheur en le faisant jouer contre `codebreaker2`, c'est-à-dire la stratégie "humaine" basée sur l'élimination des combinaisons incompatibles.

La Figure 4 compare les distributions du nombre d'essais nécessaires dans deux scénarios :

— `codemaker1` (honnête) contre `codebreaker2` ;
— `codemaker2` (tricheur) contre `codebreaker2`.

On observe un décalage net de la distribution vers la droite lorsque le codemaker devient adversarial. Sur 1000 parties, la moyenne passe d'environ 5.39 essais (codemaker honnête) à 7.34 essais (codemaker tricheur), soit un allongement moyen de +1.95 coups (environ +36%).

2. Il est alors indétectable !

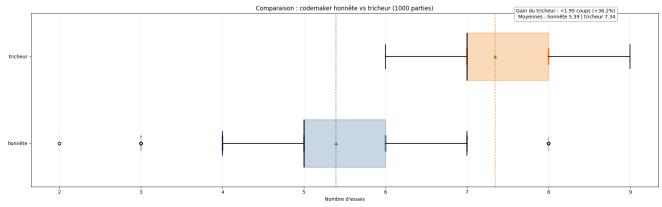

FIGURE 4: Comparaison des performances de `codebreaker2` contre un codemaker honnête (`codemaker1`) et un codemaker tricheur (`codemaker2`), sur 1000 parties.

6. Vers l'optimalité

Jusqu'à présent, `codebreaker2` se contente de choisir aléatoirement une combinaison parmi celles encore possibles au vu de l'historique. Cette stratégie est déjà très efficace contre un *codemaker* honnête, mais elle ne cherche pas explicitement à maximiser l'information obtenue à chaque coup. Or, comme le suggère la section précédente, l'objectif n'est pas seulement de rester cohérent : il s'agit aussi de progresser *même dans le pire des cas*, lorsque l'adversaire répond de manière optimale pour ralentir la résolution.

6.1. Faut-il toujours jouer une combinaison encore possible ?

Intuitivement, jouer une combinaison déjà connue comme impossible peut sembler contre-productif, puisqu'elle ne peut pas être la solution. Pourtant, ce type de coup peut être *informatif* : une proposition peut agir comme une « sonde »permettant de mieux discriminer les solutions restantes. Autrement dit, il peut être préférable de jouer une combinaison hors de l'ensemble des solutions possibles si celle-ci partitionne plus finement l'ensemble courant.

L'idée n'est pas de gagner immédiatement, mais de choisir la proposition qui minimise, dans le pire des cas, le nombre de solutions qui resteront après l'évaluation renvoyée. Cette quantité correspond à la taille du plus grand sous-ensemble induit par une même évaluation.

6.2. Codebreaker minimax : maximiser l'information dans le pire cas

Pour formaliser cette notion, supposons que l'ensemble S des solutions encore possibles soit connu du *codebreaker*. Pour une proposition g, les différentes évaluations possibles e partitionnent S en sous-ensembles :

$$S_{g,e} = \{\, s \in S \mid \texttt{evaluation}(g, s) = e \,\}$$

Un *codemaker* adversarial choisira l'évaluation qui laisse le plus grand sous-ensemble (donc le moins d'information) :

$$\text{pire\_cas}(g) = \max_e |S_{g,e}|$$

L'approche minimax consiste alors à choisir la proposition qui minimise ce pire cas :

$$g^* = \arg\min_g \max_e |S_{g,e}|$$

Ainsi, même si l'adversaire répond de manière optimale pour ralentir la partie, le *codebreaker* garantit une réduction maximale de l'espace des solutions au sens du pire cas.

C'est le principe mis en œuvre dans `codebreaker3`. Concrètement, à chaque tour, le *codebreaker* évalue l'effet de chaque proposition candidate en regardant comment elle partitionne l'ensemble S en « buckets »d'évaluations, puis sélectionne celle dont le plus gros bucket est minimal. Un critère de départage simple est utilisé : en cas d'égalité, on privilégie une proposition appartenant à S afin de conserver une chance de conclure immédiatement.

6.3. Résultats expérimentaux

La Figure 5 compare les performances de `codebreaker3` face à un *codemaker* honnête (`codemaker1`) et face à un *codemaker* tricheur (`codemaker2`).

On observe deux effets importants :

— **Amélioration de la robustesse.** Contrairement à `codebreaker2`, la stratégie minimax réduit nettement l'impact du codemaker adversarial. Dans nos essais, le tricheur n'augmente la durée moyenne que d'environ +0.8 coup (soit $\approx 15\%$), contre près de +36% avec `codebreaker2`.

— **Caractère fortement déterministe.** La dispersion est quasi-nulle et la majorité des parties se terminent avec un nombre d'essais quasi constant (ici 6). Ce comportement est cohérent avec la logique minimax : à état identique (même ensemble S), la stratégie choisit systématiquement le même coup, et un *codemaker* tricheur tend à reproduire un scénario de pire cas très stable.

7. Analyse critique, limites et conclusion

Les résultats montrent que l'exploitation systématique de l'information fournie par les évaluations est le facteur clé de performance. Le passage de `codebreaker0` à `codebreaker2` permet de réduire le nombre moyen d'essais d'un facteur 100.

La stratégie minimax implémentée dans `codebreaker3` répond précisément à la problématique d'un *codemaker* fourbe. En maximisant l'information obtenue dans le scénario le plus défavorable, elle réduit fortement l'avantage du *codemaker* adversarial et conduit à un comportement quasi déterministe, avec une variance très faible du nombre d'essais. Ce gain en robustesse se fait cependant au prix d'un coût algorithmique élevé : les *run-times* sont multipliés par un facteur 100.

Cette observation illustre un compromis classique en algorithmique : l'optimalité théorique, en particulier dans un cadre adversarial, s'accompagne presque toujours d'un surcoût computationnel significatif. Dans notre cas, ce coût reste acceptable grâce à des paramètres volontairement modestes, mais il limiterait directement la scalabilité de l'approche pour des variantes plus lourdes du jeu : avec plus de couleurs et/ou des codes plus longs.

Plusieurs pistes d'amélioration peuvent être envisagées. Il serait notamment possible de restreindre l'ensemble des propositions candidates lors du calcul minimax, d'introduire des heuristiques approximatives, ou d'exploiter des symétries du problème afin de réduire le nombre de cas à considérer. Ces approches permettraient de conserver une partie des garanties offertes par le minimax tout en améliorant sensiblement les performances pratiques.

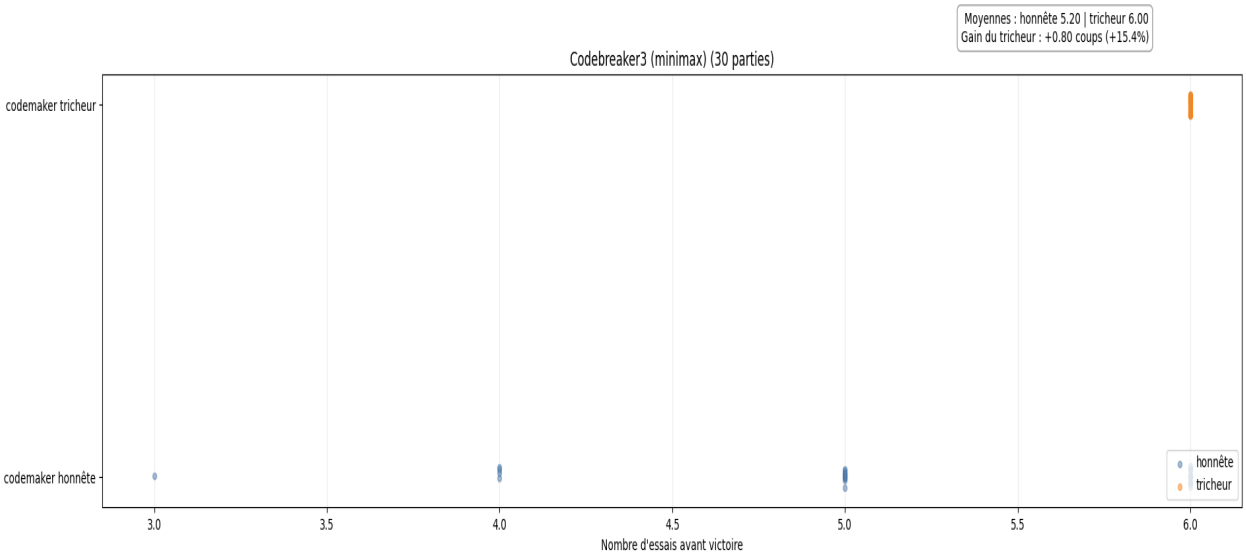

FIGURE 5: Performances de `codebreaker3` face à un *codemaker* honnête et à un *codemaker* tricheur (30 parties). La stratégie minimax réduit fortement l'avantage du *codemaker* adversarial, avec une variance très faible du nombre d'essais.