cpge-paradise.com

Filières MP/MPI - ENS de Paris-Saclay, Lyon, Rennes et Paris - Session 2024

Page de garde du rapport de TIPE

NOM : LAISSAOUI	Prénoms : YAZID
Classe : MP*	
Lycée : POTHIER	Numéro de candidat : 13328
Ville : ORLÉANS	

Concours auxquels vous êtes admissible, dans la banque MP inter-ENS (les indiquer par une croix) :

ENS Cachan	MP - Option MP		MP - Option MPI	
	Informatique			
ENS Lyon	MP - Option MP		MP - Option MPI	
	Informatique - Option M		Informatique - Option P	
ENS Rennes	MP - Option MP	X	MP - Option MPI	
	Informatique			
ENS Paris	MP - Option MP	X	MP - Option MPI	
	Informatique			

Matière dominante du TIPE (la sélectionner d'une croix inscrite dans la case correspondante) :

Informatique	X	Mathématiques		Physique	

Titre du TIPE : Le principe du CFR et applications

Nombre de pages (à indiquer dans les cases ci-dessous) :

Texte	4	Illustration	1	Bibliographie	1

Attention, les illustrations doivent figurer dans le corps du texte et non en fin du document !

Résumé ou descriptif succinct du TIPE (6 lignes, maximum) :

L'objectif de ce projet est de présenter un principe très utile pour résoudre des jeux d'informations incomplètes : le CFR, et de le mettre en application.

À Orléans

Le 06/06/2024

Signature du (de la) candidat(e)

Signature du professeur responsable de la classe préparatoire dans la discipline

Cachet de l'établissement

LYCÉE POTHIER
2 bis, rue Marcel Proust
45044 ORLÉANS CEDEX 1

Scanné avec CamScanner

cpge-paradise.com

Le principe du CFR et applications

Yazid Laissaoui

June 6, 2024

Contents

cpge-paradise.com

I-Introduction

Dans le domaine de la théorie des jeux, les jeux à informations incomplètes représentent une classe importante de problèmes où les joueurs ne possèdent pas une connaissance parfaite de l'état du jeu ou des intentions des autres joueurs. La résolution efficace de ces jeux nécessite des méthodes sophistiquées pour modéliser et prévoir les comportements potentiels des adversaires. Une des méthodes les plus prometteuses et largement utilisées dans ce contexte est le *Counterfactual Regret Minimization* (CFR).

L'objectif de ce projet est de présenter le principe du CFR et de démontrer son efficacité et son application, notamment dans le cadre d'un jeu extensif (c'est-à-dire, représenté sous forme d'arbre de décision) à informations incomplètes : le Poker (variante Texas Hold'em à 2 joueur avec limite de bet). Nous allons d'abord introduire les concepts théoriques sous-jacents au CFR, puis nous illustrerons l'application pratique du CFR à travers deux exemples concret.

II-Définitions

II-1-Élément de théorie des jeux

Pour un jeu extensif à information incomplète, on définit :

- $N = \{1, \dots, n\}$ un ensemble fini de joueurs, pour la suite on se contentera de $N = \{1, 2\}$
- H l'ensemble des histoires possibles, formellement c'est un ensemble de suites finies d'éléments codant une partie vérifiant :

$$\forall h = (h_1, \dots, h_n) \in H,\ \forall k \in \{1, \dots, n\},\ (h_1, \dots, h_k) \in H$$

- On pose Z l'ensemble des histoires terminales, c'est-à-dire:

$$Z = \{h \in H \mid \forall h' \in H \setminus \{h\}, h \text{ n'est pas préfixe de } h'\}$$

- Pour h dans H, on pose $A(h)$ l'ensemble des actions possibles après h:

$$A(h) = \{a \mid (h, a) \in H\}$$

- Enfin, on pose P : fonction qui prévoit le joueur qui doit jouer:

$$P : H \setminus Z \to N \cup \{c\}$$

$P(h) = c \Leftrightarrow$ c'est la "chance" (le hasard) qui doit jouer (distribution de cartes, par exemple).

- Pour chaque joueur $i \in N$, une partition $\mathcal{I}_i$ de $\{h \in H : P(h) = i\}$ avec la propriété que $A(h) = A(h')$ chaque fois que h et h' sont dans le même membre de la partition. Plus précisément, un élément I_i de $\mathcal{I}_i$ est un ensemble d'histoires telles que le joueur i ne peut pas les distinguer (cela est dû au caractère incomplet des informations). Pour le Poker, I_i peut être un ensemble d'histoires qui diffèrent dans la main attribuée à l'adversaire. On notera $A(I_i)$ pour désigner $A(h)$ avec $h \in I_i$ quelconque.
- Pour chaque joueur $i \in N$, une fonction d'utilité u_i des états terminaux Z aux réels $\mathbb{R}$. Le Poker étant à deux joueurs et à somme nul, on a $u_1 = -u_2$. Définissons enfin $\Delta_{u,i} = \max_z u_i(z) - \min_z u_i(z)$ comme étant l'intervalle des utilités pour le joueur i.
- Une stratégie σ_i (pour $i \in N \cup \{c\}$) est une fonction qui a tout $I_i \in \mathcal{I}_i$ assigne une distribution de probabilité sur $A(I_i)$. On parle de stratégie mixte, à la différence d'une stratégie pure qui a un état d'information associe une action (c'est le cas dans les jeux à informations parfaites)
- On appelle profil stratégique un élément $\sigma = (\sigma_i)_{i \in N}$ (on notera aussi $\sigma_{-i} = (\sigma_j)_{j \neq i, j \in N}$)
- On note $\pi^\sigma(h)$: la probabilité que l'histoire h arrive si on joue les actions par rapport à σ.
  On a alors : $\pi^\sigma = \prod_{i \in N \cup \{c\}} \pi_i^\sigma$ avec π_i^σ qui représente la contribution du joueur i.
- Pour $I \in \mathcal{P}(H)$, on définit

$$\pi^\sigma(I) = \sum_{h \in I} \pi^\sigma(h)$$

- On définit enfin

$$u_i(\sigma) = \sum_{h \in Z} u_i(h) \pi^\sigma(h)$$

cpge-paradise.com

II-2-Notion d'équilibre de Nash

On appelle un profil stratégique $\sigma = (\sigma_i)_{i \in N}$ un équilibre de Nash (resp. un ε-Eq de Nash) si :

$$\forall i \in N,\ u_i(\sigma) \geq \max_{\sigma'_i \in \Sigma_i} u_i(\sigma'_i, \sigma_{-i}) \text{ (resp. } \geq \max_{\sigma'_i \in \Sigma_i} u_i(\sigma'_i, \sigma_{-i}) - \varepsilon)$$

où Σ_i est l'ensemble des stratégies possibles pour le joueur i.

Qualitativement, on peut voir ici qu'un équilibre de Nash est une stratégie telle que que toute déviation de celle ci ne peut amener à une amélioration. On peut aussi parler d'exploitabilité : c'est à dire que même en supposant que notre adversaire connaisse parfaitement notre stratégie, il lui ai impossible d'en trouver une qui nous batte en espérance.

Enfin, on pourrait se poser la question suivante : existe t'il toujours une stratégie mixte qui est une équilibre de Nash ? La réponse est oui, et vient du **Théorème de Nash** , qui nécessite l'hypothèse d'un jeu avec un nombre d'action fini (ce qui est le cas dans la variante du Poker utilisée dans la suite)

III-Principe de l'algorithme

III-1-Principe général

Le principe général du CFR est le suivant : à chaque décision prise pendant une partie, l'algorithme explore les alternatives possibles à cette décision et calcule le regret associé. Le regret est défini comme la différence entre la récompense réellement obtenue et la récompense qui aurait été obtenue en prenant une autre décision, les décisions qui ont un fort regret seront ensuite plus jouées à l'avenir.

Plus formellement, étant donné $I \in \mathcal{I}_i, h \in I$, on définit la récompense qu'on aurait pu recevoir si on avait joué $a \in A(h)$ comme :

$$v_i^\sigma(h, a) = \sum_{h' \in Z, h \text{ préfixe de } h'} \pi^{\sigma_{-i}}(h) \pi^{\sigma_{I \to a}}(h, h') u_i(h')$$

avec $\sigma_{I \to a}$ la stratégie identique à σ mais dont le joueur i joue forcement a si il est dans une histoire dans I, et $\pi^{\sigma_{I \to a}}(h, h')$ la probabilité de passer de h à h' en jouant selon la stratégie $\sigma_{I \to a}$. On définit aussi notre récompense de base comme :

$$v_i^\sigma(h) = \sum_{h' \in Z, h \text{ préfixe de } h'} \pi^{\sigma_{-i}}(h) \pi^{\sigma}(h, h') u_i(h')$$

On définit alors naturellement :

$$v_i^\sigma(I, a) = \sum_{h \in I} v_i^\sigma(h, a)$$

$$v_i^\sigma(I) = \sum_{h \in I} v_i^\sigma(h)$$

On a alors le regret immédiat contrefactuel qui vaut :

$$R_{i,imm}(I, a) = v_i^\sigma(I, a) - v_i^\sigma(I)$$

Notons maintenant T le nombre de fois qu'on est tombé sur I (c'est à dire, t s'incrémente de 1 à chaque fois qu'on a rencontré I), et σ^t la stratégie mixte au temps t, on a alors :

$$R_{i,imm}^T(I, a) = \sum_{t=1}^{T} R_{i,imm}^t(I, a) = \sum_{t=1}^{T} v_i^{\sigma^t}(I, a) - v_i^{\sigma^t}(I)$$

On pose alors

$$R_{i,imm}^{T,+}(I, a) = max(R_{i,imm}^T(I, a), 0)$$

Notre stratégie au temps T+1 vaut donc :

$$\sigma_i^{T+1}(I, a) = \begin{cases} \frac{R_{i,imm}^{T,+}(I,a)}{\sum_{a \in A(I)} R_{i,imm}^{T,+}(I,a)} & \text{si } \sum_{a \in A(I)} R_{i,imm}^{T,+}(I, a) > 0 \\ \frac{1}{|A(I)|} & \text{sinon.} \end{cases}$$

Enfin pour clore cette partie, on peut donner le pseudo-code de ce principe :

cpge-paradise.com

Algorithme 1 Minimisation du regret contrefactuel (CFR)

Entrée : Arbre de jeu G, Nombre d'itérations T
Initialiser les tableaux de regret et de stratégie : $\text{regretSum}[I, a] \leftarrow 0$, $\text{strategySum}[I, a] \leftarrow 0$ pour tous les ensembles d'information I et actions a
for $t = 1$ à T **do**
    $s \leftarrow$ Échantillonner le profil de stratégie en utilisant getStrategy(regretSum)
    Parcourir l'arbre de jeu G pour calculer les utilités et les regrets contrefactuels
    for chaque ensemble d'information I rencontré **do**
        Mettre à jour les regrets : $\text{regretSum}[I, a] \leftarrow \text{regretSum}[I, a] + \text{regret}[I, a]$ pour toutes les actions a
        Mettre à jour les sommes de stratégie : $\text{strategySum}[I, a] \leftarrow \text{strategySum}[I, a] + \text{strategy}[I, a]$ pour toutes les actions a
    end for
end for
Sortie : Stratégie moyenne : $\pi^*(I, a) \leftarrow \frac{\text{strategySum}[I,a]}{\sum_b \text{strategySum}[I,b]}$ pour tous les ensembles d'information I et actions a

Algorithme 2 getStrategy

function GETSTRATEGY(regretSum)
    Initialiser le tableau de stratégie : $\text{strategy}[a] \leftarrow 0$ pour toutes les actions a
    $R^+ \leftarrow \sum_a \max(\text{regretSum}[a], 0)$
    if $R^+ > 0$ **then**
        for chaque action a **do**
            $\text{strategy}[a] \leftarrow \max(\text{regretSum}[a], 0)/R^+$
        end for
    else
        for chaque action a **do**
            $\text{strategy}[a] \leftarrow 1/|\text{actions}|$
        end for
    end if
    return strategy
end function

cpge-paradise.com

III-2-Du regret à l'équilibre de Nash

Il reste donc maintenant à prouver que la minimisation du regret comme ci dessus amène bien à approcher l'équilibre de Nash. Et pour cela, on va énoncé le lien fort entre équilibre de Nash et le fait de minimiser le regret, et c'est l'objet du :

Théorème 1 *Dans un jeu à somme nulle au temps T, si le* ***regret moyen global*** *des deux joueurs est inférieur à ϵ, alors le profil de stratégie moyenne $\bar{\sigma}^T$ est un 2ϵ-équilibre de Nash.*

avec le regret moyen global au temps T définit formellement comme :

$$\forall i \in N,\ R^T_{i,moy} = \frac{1}{T} \max_{\sigma_i^* \in \Sigma_i} \sum_{t=1}^{T} \left(u_i(\sigma_i^*, \sigma^t_{-i}) - u_i(\sigma^t) \right)$$

et la strategie moyenne définit aussi comme :

$$\bar{\sigma}_i^T(I)(a) = \frac{\sum_{t=1}^T \pi_i^{\sigma^t}(I)\sigma_i^t(I)(a)}{\sum_{t=1}^T \pi_i^{\sigma^t}(I)}$$

Ainsi, formellement : si

$$R^T_{1,moy} \leq \epsilon \quad \text{et} \quad R^T_{2,moy} \leq \epsilon,$$

alors

$$\bar{\sigma}^T \text{ est un } 2\epsilon\text{-équilibre.}$$

Preuve :
D'une part

$$\begin{aligned} R_1^T &= \frac{1}{T} \max_{\sigma_1^* \in \Sigma_1} \sum_{t=1}^{T} \left(u_1(\sigma_1^*, \sigma_2^t) - u_1(\sigma^t) \right) \\ &= \max_{\sigma_1^* \in \Sigma_1} \left[\frac{1}{T} \sum_{t=1}^{T} u_1(\sigma_1^*, \sigma_2^t) - u_1(\sigma^t) \right] \\ &= \max_{\sigma_1^* \in \Sigma_1} \left[u_1(\sigma_1^*, \bar{\sigma}_2^T) - u_1(\bar{\sigma}^T) \right] \end{aligned}$$

D'autre part,

$$\begin{aligned} R_2^T &= \frac{1}{T} \max_{\sigma_2^* \in \Sigma_2} \sum_{t=1}^{T} \left(u_2(\sigma_1^t, \sigma_2^*) - u_2(\sigma^t) \right) \\ &= \max_{\sigma_2^* \in \Sigma_2} \left[\frac{1}{T} \sum_{t=1}^{T} u_2(\sigma_1^t, \sigma_2^*) - u_2(\sigma^t) \right] \\ &= \max_{\sigma_2^* \in \Sigma_2} \left[u_2(\bar{\sigma}_1^T, \sigma_2^*) - u_2(\bar{\sigma}^T) \right] \end{aligned}$$

Donc,

$$\begin{aligned} R_1^T + R_2^T &= \max_{\sigma_1^* \in \Sigma_1} \left[u_1(\sigma_1^*, \bar{\sigma}_2^T) - u_1(\bar{\sigma}^T) \right] + \max_{\sigma_2^* \in \Sigma_2} \left[u_2(\bar{\sigma}_1^T, \sigma_2^*) - u_2(\bar{\sigma}^T) \right] \\ &\leq 2\epsilon \quad (\text{car } u_1 = -u_2) \end{aligned}$$

Puis,

$$u_1(\bar{\sigma}^T) \leq 2\epsilon + \max_{\sigma_2^* \in \Sigma_2} \left[u_1(\bar{\sigma}_1^T, \sigma_2^*) \right]$$

Symétriquement,

$$u_2(\bar{\sigma}^T) \leq 2\epsilon + \max_{\sigma_1^* \in \Sigma_1} \left[u_2(\sigma_1^*, \bar{\sigma}_2^T) \right] \textbf{ CQFD.}$$

Il suffit enfin de donner le lien entre le regret moyen, et le regret immédiat (c'est à dire celui donner lors du III-1), et cela vient du :

cpge-paradise.com

Théorème 2 *Le regret total R_i^T du joueur i est borné par la somme des regrets contrefactuels immédiats positifs sur tous les ensembles d'information $I \in \mathcal{I}_i$.*

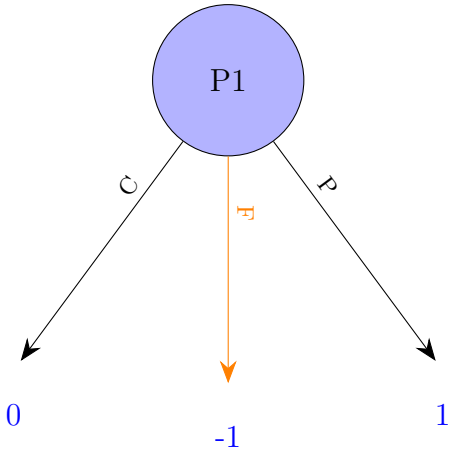
Formellement :

$$R_{i,moy}^T \leq \sum_{I \in \mathcal{I}_i} R_{i,\text{imm}}^{T,+}(I)$$

La preuve (assez technique) sera admise. Ce résultat est centrale dans la justification de l'approche du CFR, car cela indique que réussir à minimiser les regret immédiats permet d'aussi minimiser le regret moyen, et donc par le Théorème 1 d'approcher (**en moyenne**) l'équilibre de Nash.

III-3-Exemple simple d'application : Pierre Feuille Ciseau

Pour mieux comprendre, on va essayer d'appliquer le CFR au jeu du Pierre-Feuille-Ciseaux.



Calcul des Regrets :
Le regret pour chaque action est calculé comme suit :

- Pour l'action **C (Ciseaux)** : Si le joueur avait choisi C, le payoff aurait été 0. Le regret est donc la différence entre le payoff obtenu (par exemple, -1 en choisissant F) et le payoff hypothétique, soit $R(C) = 0 - (-1) = 1$.
- Pour l'action **F (Feuille)** : Le payoff est -1. Le regret est donc $R(F) = -1-(-1) = 0$.
- Pour l'action **P (Pierre)** : Si le joueur avait choisi P, le payoff aurait été 1. Le regret est donc $R(P) = 1 - (-1) = 2$.

On imagine que c'est la première partie que notre algorithme a joué, sa stratégie sera donc au prochain tour de : 2/3 pour pierre, 1/3 pour feuille et 0 pour ciseau. Notre algorithme pour apprendre va donc jouer contre lui même et essayer d'ajuster sa stratégie de manière à minimiser le regret. J'ai donc implémenter cela (code en annexe), et voici le résultat :

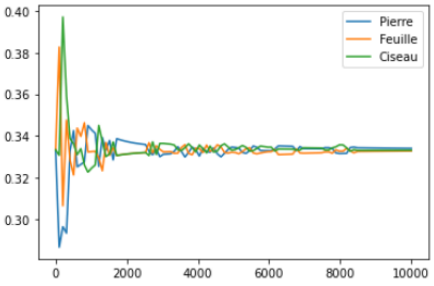


Figure 1: PFC stratégie

avec en abscisse le nombre de parties joués et en ordonné la probabilité de joué un coup, on voit que l'algorithme converge bien vers un équilibre de Nash en stratégie mixte (qui est la stratégie aléatoire)

III-4- 2ème exemple d'application : Poker

On va maintenant voir l'efficacité du CFR dans un jeu (beaucoup) plus compliqué : le Poker, pour ce faire on va considérer une version simplifié du Poker (7 niveau de cartes au lieu de 13, une limite de bet fixé à 4 jetons : on a donc que 5 actions possibles à chaque prise de décision (de 0 à 4 jetons)). Un **état d'information** sera donc composer de nos cartes, des éventuelles cartes sur la table, et de l'ensemble

cpge-paradise.com

des actions passées (les nôtres et celles de l'adversaire). Notre algorithme va donc apprendre en jouant contre lui même, j'ai donc implementé cela (code en annexe), et on peut donc tester notre algorithme s'étant entrainer pendant 10^6 parties, pour ce faire on peut le faire jouer contre des stratégie naïves :

P1 \ P2	**CFR**	**Random**	**Fullbet**
CFR	-	87%	85%
Random	13%	-	49%
Fullbet	15%	51%	-

Table 1: Table de résultat (en % de victoire de P1)

Où **Fullbet** signifie le fait de miser tout le temps 4 (le maximum donc), et **Random** signifie jouer de manière aléatoire. On peut donc noter que le CFR est assez efficace contre ces stratégies naives.

IV-Bibliographie

References

[1] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret Minimization in Games with Incomplete Information. *Advances in Neural Information Processing Systems (NIPS)*, 2007. URL: https://poker.cs.ualberta.ca/publications/NIPS07-cfr.pdf.

[2] Marc Lanctot, Kevin Waugh, Martin Zinkevich, and Michael Bowling. Efficient Monte Carlo Counterfactual Regret Minimization in Games with Many Player Actions. *Advances in Neural Information Processing Systems (NIPS)*, 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/3df1d4b96d8976ff5986393e8767f5b2-Paper.pdf.

[3] Noam Brown and Tuomas Sandholm. Combining Deep Reinforcement Learning and Search for Imperfect-Information Games. *arXiv preprint*, 2020. URL: https://arxiv.org/pdf/2007.13544.pdf.

[4] Renaud Bourles. Cours de théorie des jeux. 2019. URL: http://renaud.bourles.perso.centrale-marseille.fr/Cours/Theorie_des_jeux.pdf.

cpge-paradise.com

V-Annexe

A-Code PFC

```
#Variables
nombreActions = 3
nomAction = ["Pierre", "Feuille", "Ciseau"]
actionPossible =  np.arange(nombreActions)
composanteAction = np.array([[0,-1,1],
                             [1,0,-1],
                             [-1,1,0]])
nombreRegrets = np.zeros(nombreActions)
nombreStrategies = np.zeros(nombreActions)
en_nombreRegrets = np.zeros(nombreActions)
en_strategie_nombre = np.zeros(nombreActions)
```

Figure 2: PFC variables

```
#fonction appliquant le CFR
def prendre_strategie (nombreRegrets) :
    #on supprime les valeur de regret négatives
     strategie = np.clip(nombreRegrets, a_min = 0, a_max = None)
     #on somme les differents regrets
     sommeRegret = sum(strategie)
     if sommeRegret != 0 :
             strategie =  strategie / sommeRegret
     else :
         strategie = np.repeat(1/nombreActions, nombreActions)
     return strategie
```

Figure 3: PFC stratégie

cpge-paradise.com

```python
#Fonction pour entrainer notre algo
def entrainer (iteration) :
    for i in range(iteration) :
        #on pose les stratégie en fonction du résultat précédent
        ma_strategie = prendre_strategie(nombreRegrets)
        en_strategie = prendre_strategie(en_nombreRegrets)
        #on les additionne avec les ancienne strategie
        for j in range(nombreActions) :
            nombreStrategies[j] += ma_strategie[j]
            en_strategie_nombre[j] += en_strategie[j]
        #on joue le partie
        mon_action = prendre_decision(ma_strategie)
        en_action = prendre_decision(en_strategie)
        #on prends le résultat
        ma_recompense = prendre_recompense(mon_action, en_action)
        en_recompense = prendre_recompense(en_action, mon_action)
        #déduire le regrets
        for a in range(nombreActions) :
            mon_regret = prendre_recompense(a, mon_action) - ma_recompense
            en_regret = prendre_recompense(a, en_action) - en_recompense
            nombreRegrets[a] += mon_regret
            en_nombreRegrets[a] += en_regret
```

Figure 4: PFC entrainement

B-Code Poker

```python
class pokerBot:
    """
    Bot de poker qui utilise une approche de regret minimisation pour prendre des décisions.
    """
    def __init__(self, strategy_manager, name):
        self.strategy_manager = strategy_manager
        self.main = []
        self.jeton = regle["jeton_base"]
        self.action = [[[], []] for _ in range(nb_Tour)] #contient l'historique des actions, action[tour-1][0] -> renvoie une liste contenant toutes les actions
                                                         #prises dans l'ordre, et action[tour-1][1] celles de l'adversaire elles aussi prises dans l'ordre
                                                         #par exemple, la liste [[[3], [2, 1]], [[4], [4]]] traduit donc le déroulement d'une partie comme suit :
                                                         # pre_flop : l'adversaire a misé 2, j'ai raise 3 puis il a call 1
                                                         # post_flop : il a misé 4 et j'ai call

        self.is_fold = False
        self.is_check = False
        self.is_all_in = False
        self.name = name
    def donner_carte(self, carte):
        self.main = carte
        # Conversion des cartes en string pour clé dans le dictionnaire
        self.main_PP = conv_carte_str(carte)

    def prendre_decision(self, tour, flop, p2):
        key = self.create_key(tour, flop, self.action)
        strategy = self.strategy_manager.get_strategy(key)


        self.action[tour-1][0].append(min(np.random.choice(np.arange(len(strategy)), p=strategy), self.jeton))
        p2.action[tour-1][1].append(self.action[tour-1][0][-1])

    def prendre_decision_bis(self, tour, flop):
        key = self.create_key(tour, flop, self.action)
        strategy = self.strategy_manager.get_strategy(key)


        self.action[tour-1][0].append(min(np.random.choice(np.arange(len(strategy)), p=strategy), self.jeton))
```

Figure 5: class 1

cpge-paradise.com

```python
def create_key(self, tour, flop, etat):
    # Création de la clé pour le dictionnaire de stratégie
    if self.name == "former" :
        if tour == 1:
            if len(etat[tour-1][1]) > 0 :

                return (self.main_PP, etat[tour-1][1][-1])
            else :
                return (self.main_PP, -1)
        elif tour == 2:
            if len(etat[tour-1][1]) > 0 :

                return (self.main_PP, combinaison_jeu_ancien(self.main, flop), etat[tour-1][1][-1])
            else :
                return (self.main_PP, combinaison_jeu_ancien(self.main, flop), -1)

    if tour == 1:
        return (self.main_PP, conv_action_str(etat))
    elif tour == 2:

        return (self.main_PP, combinaison_jeu(self.main, flop), conv_action_str(etat))



def reinitialiser_total(self):
    self.main = []
    self.main_PP = ""
    self.jeton = regle["jeton_base"]
    self.action = [[[], []] for _ in range(nb_Tour)]
    self.is_fold = False
    self.is_check = False
    self.is_all_in = False

def reinitialiser_tour(self):
    self.main = []
    self.main_PP = ""
    self.action = [[[], []] for _ in range(nb_Tour)]
    self.is_fold = False
    self.is_check = False
    self.is_all_in = False
```

Figure 6: class 2

```python
def copy(self, name) :
    # Crée une nouvelle instance de pokerBot en utilisant le même gestionnaire de stratégie.
    p = pokerBot(self.strategy_manager, name)
    # Copie les attributs de l'état du jeu courant
    p.main = copy.deepcopy(self.main) # Crée une copie de la liste pour éviter les modifications partagées.
    p.main_PP = self.main_PP
    p.jeton = self.jeton
    p.action = copy.deepcopy(self.action)  # Copie la liste pour assurer l'indépendance des actions.
    p.is_fold = False
    p.is_check = False
    p.is_all_in = False
    return p

def choix_action(self, tour, flop):
    if self.name == 'humain':
        action = int(input(f"Quelle est votre décision ? (entrez un numéro entre 0 et {min(self.jeton, 4)} qui correspond à la valeur de votre bet): "))
        while action < 0 or action > min(self.jeton, 4):
            action = int(input(f"Entrez un numéro valide entre 0 et {min(self.jeton, 4)} qui correspond à la valeur de votre bet: "))
        return action
    strategy = self.strategy_manager.get_strategy(self.create_key(tour, flop, self.action))
    # Filtrer pour ne garder que les mises possibles avec les jetons actuels
    actions_possibles = [i for i in betPossible if i <= self.jeton]

    probabilities = np.array([strategy[i] for i in actions_possibles])
    #les probabilites de misé plus haut sont tous mise dans le all-in si besoin
    for i in range(len(actions_possibles), len(betPossible)) :
        probabilities[-1] += strategy[i]
    probabilities /= probabilities.sum()  # Normaliser les probabilités pour garantir qu'elles somment à 1

    action = np.random.choice(actions_possibles, p=probabilities)
    return action
```

Figure 7: class 3

cpge-paradise.com

```python
class StrategyManager:
    """
    Gestionnaire des stratégies et des regrets pour le pokerBot.
    """
    def __init__(self):
        self.regret_T1 = defaultdict(lambda: np.zeros(nombreActions))
        self.regret_T2 = defaultdict(lambda: np.zeros(nombreActions))
        self.nbr_reg = [self.regret_T1, self.regret_T2]

    def update_regrets(self, key, tour,action, regret):
        # Mettre à jour la matrice de regret pour une action spécifique
        self.nbr_reg[tour-1][key][action] += regret

    def get_strategy(self, key):
        # Récupérer la stratégie actuelle basée sur les regrets cumulés
        regrets = self.regret_T1[key]
        positive_regrets = np.maximum(regrets, 0)
        total_positive = np.sum(positive_regrets)
        if total_positive > 0:
            return positive_regrets / total_positive
        else:
            return np.ones(len(regrets)) / len(regrets)
```

Figure 8: class stratégie

```python
#fonction qui calcule le regret de l'état correspondant
def regret_de_etat(j, tour, p1, p2, flop, etat, jeton_P1, jeton_P2, pot, recomp_initiale) :

    actions_possibles = [i for i in betPossible if i <= jeton_P1 and i != p1.action[tour-1][0][len(etat[tour-1][0])]]
    for action in actions_possibles :
        #on crée des copies de nos bots pour le calcul des regrets
        p1_reg = p1.copy(p1.name + "_reg")
        p2_reg = p2.copy(p2.name + "_reg")
        #on les initialise
        p1_reg.jeton = jeton_P1
        p2_reg.jeton = jeton_P2
        #on simule le noeud (de l'arbre de jeu) duquel on veut calculer le regret
        p1_reg.action = copy.deepcopy(etat)
        p1_reg.action[tour-1][0].append(action)
        p2_reg.action = [[copy.deepcopy(etat[0][1]), copy.deepcopy(etat[0][0])], [copy.deepcopy(etat[1][1]), copy.deepcopy(etat[1][0])]]
        p2_reg.action[tour-1][1].append(action)

        p1_reg.jeton -= sum(p1_reg.action[0][0])
        p2_reg.jeton -= sum(p2_reg.action[0][0])
        if tour == 2 :
           p1_reg.jeton -= sum(p1_reg.action[1][0])
           p2_reg.jeton -= sum(p2_reg.action[1][0])

        if p1_reg.jeton >= 0 :
            if p1_reg.jeton == 0 :
                p1_reg.is_all_in = True
            if len(p2_reg.action[tour-1][0]) > 0 and p1_reg.action[tour-1][0][-1] < p2_reg.action[tour-1][0][-1] : #On s'assure que l'action ne condiit pas
                                                                                                                    #à un fold automatique
                p1_reg.is_fold = True
                recomp_P1 = p1_reg.jeton  - jeton_P1
            else :
                # Simuler les parties avec une action forcée
                pot = jouer_partie(1, tour, p1_reg, p2_reg, flop,  0) #Ici j = 1 car on vient de jouer, donc c'est le tour de l'adversaire
                if not p1_reg.is_fold and not p2_reg.is_fold and tour == 1:
                    # Continuer au tour suivant si personne n'a fold
                    pot = jouer_partie(j, 2, p1_reg, p2_reg, flop, pot)

                recomp_P1, recomp_P2 = conclusion_partie(p1_reg, p2_reg, flop, pot)
            key = p1.create_key(tour, flop, etat)
            p1.strategy_manager.update_regrets(key, tour, action, recomp_P1 - recomp_initiale)
```

Figure 9: fonction calculant le regret d'un état

```python
#fonction qui renvoie les etats qu'on devra rejouer afin de calculer le regret
#elle renvoie l'ensemble des états dont on doit calculer le regret, un état étant une partie d'un historique (une sorte de slicing de p.action)
def prendre_ensembles_etats_regrets(j_relatif, tour, p1):
    #j_relatif = 0 si c'est à celui dont on calcul le regret de commencer (indépendament de si c'est p1 ou p2)
    ensemble_regret = []
    for k in range(len(p1.action[tour-1][0])) :
        if j_relatif%2 == 0 :

            etat = [copy.deepcopy(p1.action[tour-1][0][:k]), copy.deepcopy(p1.action[tour-1][1][:k])]
        if j_relatif%2 == 1 :
            etat = [copy.deepcopy(p1.action[tour-1][0][:k]), copy.deepcopy(p1.action[tour-1][1][:k+1])]
        if tour == 1 :
            etat_complet = [etat, [[], []]]
        else :
            etat_complet = [copy.deepcopy(p1.action[0]), etat]
        ensemble_regret.append(etat_complet)
        # if tour == 2 :
        #     ensemble_regret.append([p1.action[0], etat])
    return ensemble_regret
```

Figure 10: fonction renvoyant l'ensemble des états dont on doit calculer le regret

```python
def chercher_regret(j, j_relatif, tour, p1, p2,flop, jeton_P1, jeton_P2, recomp_initiale , pot) :
    for etat in prendre_ensembles_etats_regrets(j_relatif, tour, p1) :
        regret_de_etat(j, tour, p1, p2, flop, etat, jeton_P1, jeton_P2, pot, recomp_initiale)
```

Figure 11: fonction de synthèse des deux précédentes

cpge-paradise.com

```python
def entrainer(i):
    for k in range(i):
        if (k+1) % 10000 == 0:
            print(f"{k+1} parties, déjà !")
        j = 0
        p1.reinitialiser_total()
        p2.reinitialiser_total()
        while p1.jeton > 0 and p2.jeton > 0:
            j += 1

            jeu = init_jeu(p1, p2)
            flop = jeu[2]

            jeton_prePartie_P1 = p1.jeton
            jeton_prePartie_P2 = p2.jeton
            pot = 0

            pot = jouer_partie(j, 1, p1, p2, flop, pot)
            pot_T1 = pot
            partie_continue = not p1.is_fold and not p2.is_fold and not p1.is_all_in and not p2.is_all_in

            if partie_continue:
                pot = jouer_partie(j, 2, p1, p2, flop, pot)

            recomp_P1, recomp_P2 = conclusion_partie(p1, p2, flop, pot)
            p1_combi, p2_combi = combinaison_jeu(p1.main, flop), combinaison_jeu(p2.main, flop)

            #place au calcul des regrets :

            #Pour P2
            chercher_regret(j, (j+1)%2 , 1, p2, p1, flop, jeton_prePartie_P2, jeton_prePartie_P1, recomp_P2, 0)
            if len(p2.action[1][0]) > 0  : #si on a joué quelque chose au T2
                chercher_regret(j, (j+1)%2 , 2, p2, p1, flop, jeton_prePartie_P2, jeton_prePartie_P1, recomp_P2, pot_T1)
            #pour P1
            chercher_regret(j, j%2, 1, p1, p2, flop, jeton_prePartie_P1, jeton_prePartie_P2, recomp_P1, 0)
            if len(p1.action[1][0]) > 0 : #si on a joué quelque chose au T2
                chercher_regret(j, j%2, 2, p1, p2, flop, jeton_prePartie_P1, jeton_prePartie_P2, recomp_P1, pot_T1)
```

Figure 12: fonction d'entrainement