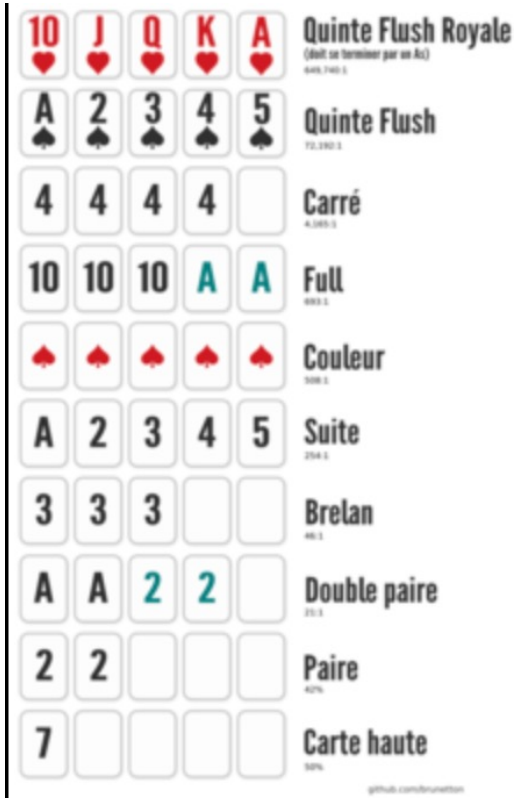
Créer une IA qui gagne au poker

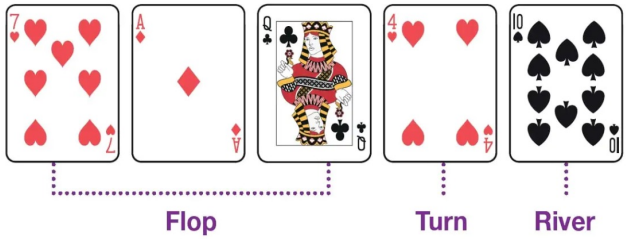
Sommaire

I) Règles du poker

- Texas hold'em :

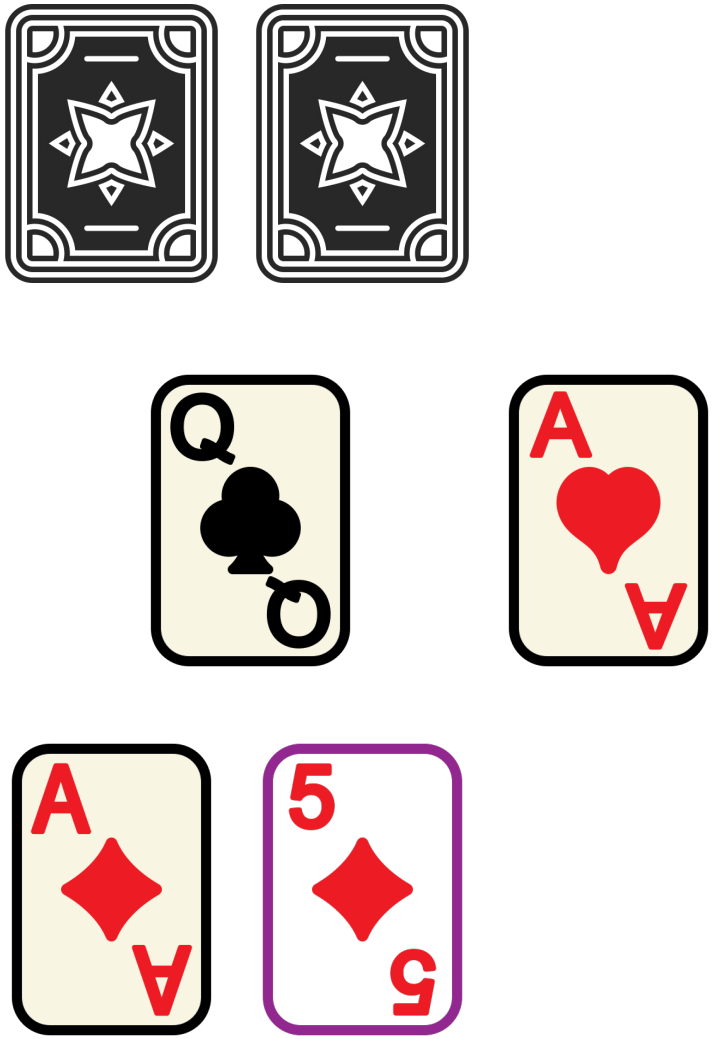
  - 2 cartes par personne
  - Des cartes en commun (5 au total)
  - Des jetons de base (argent)
  - Prise de décision à chaque tour (jusqu'à 4 tours)
  - Si la partie va jusqu'au bout, la meilleure combinaison gagne





I) Règles du poker

Exemple de table :



I) Règles du poker

- VOCABULAIRE :
  - **Fold** : se coucher, c'est-à-dire abandonné la partie car on ne veut pas miser autant que l'adversaire
  - **Call** : miser autant que l'adversaire
  - **Raise** : augmenter la mise, ou simplement miser si on commence

Remarque : pas de small/big blind

II) Principe CFR et 1ère application

- 2 types de jeu :
  - Jeu à information parfaite :

  - Jeu à information imparfaite : notre poker !

Différence ?

-échecs : je peux voir toutes les pièces de mon adversaire
-Poker : je ne connais pas les cartes de l'adversaire…

II) Principe CFR et 1ère application

- Equilibre de Nash : un ensemble de stratégies dans lesquelles aucun joueur ne peut s'améliorer en déviant

- Pour l'atteindre, notre stratégie ne doit pas être exploitable même en considérant que notre adversaire la connait

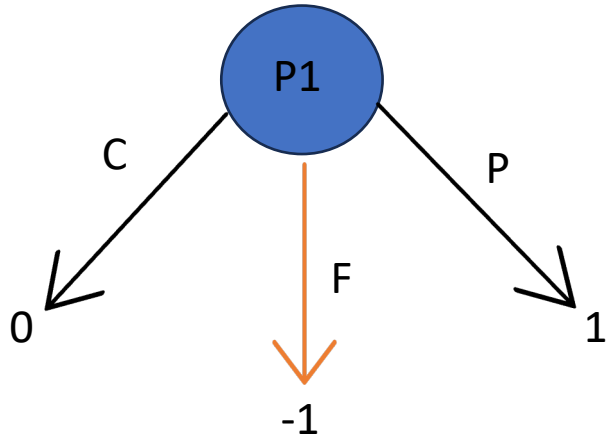
- Le but est donc d'atteindre l'équilibre de Nash (en tout cas de l'approcher)

II) Principe CFR et 1ère application

- CFR : Counterfactual Regret minimization
- Principe : lorsque l'on prend une décision, on regarde le résultat de tous les autres coups et on le compare avec celui que l'on a choisi. Les coups qui auraient été meilleurs que le nôtre auront une plus grande chance d'être choisis à l'avenir.

Application PFC :

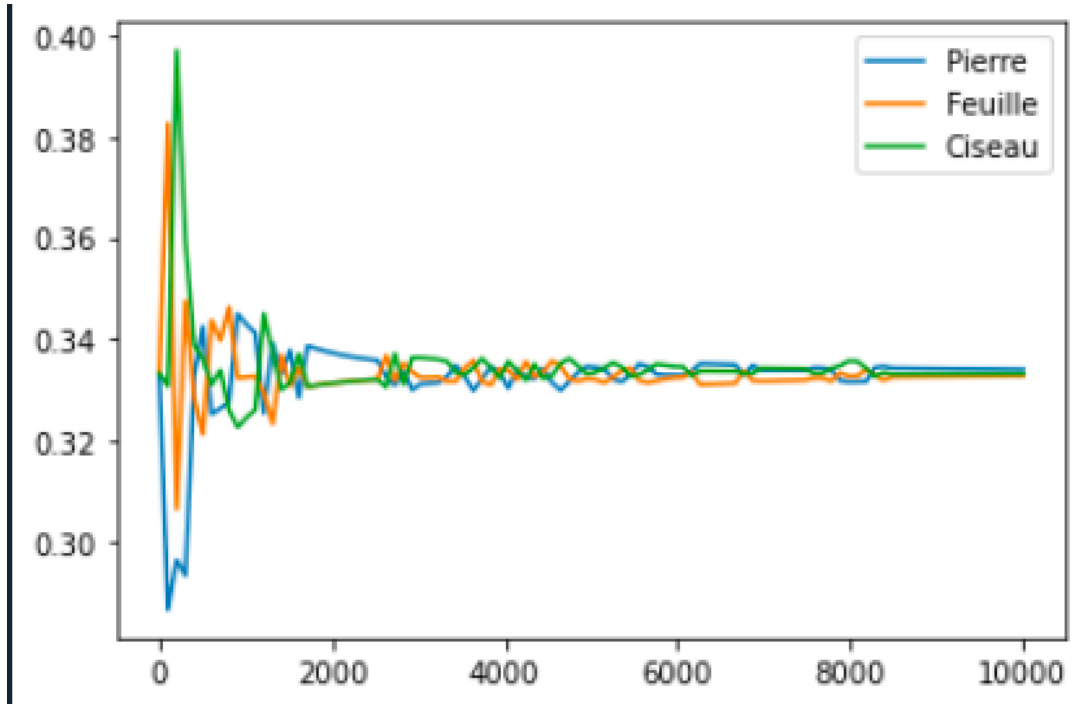
Rgt(P) = 1 – (-1) = 2
Rgt(F) = 0
Rgt(C) = 0 – (-1) = 1

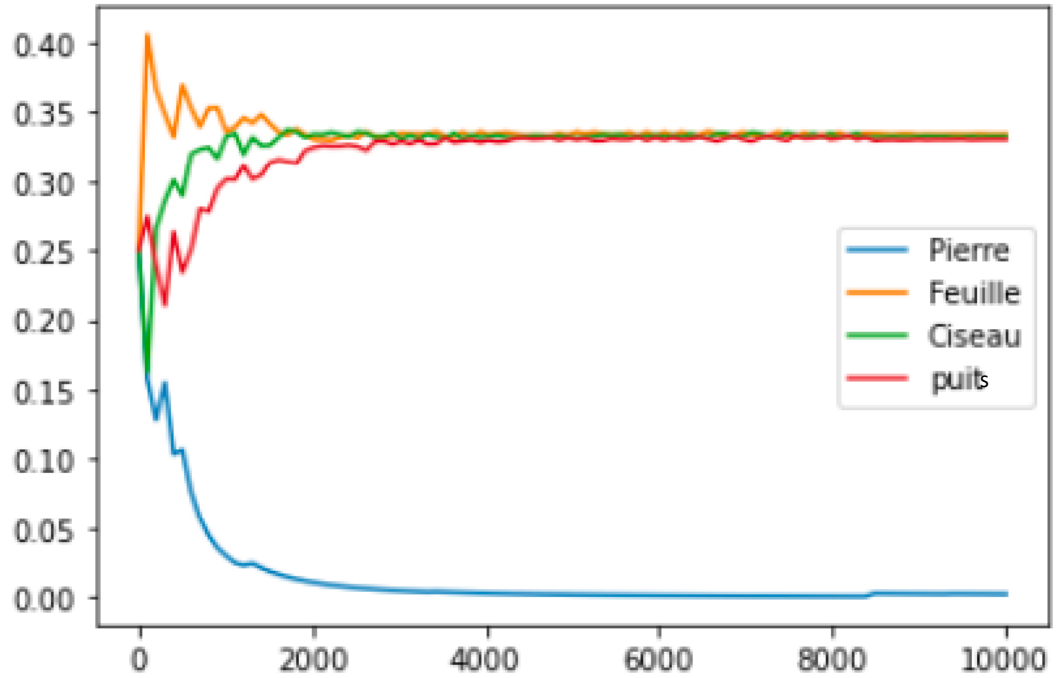


Pour apprendre, notre algorithme va jouer contre lui-même et améliorer sa stratégie en fonction des regrets

II) Principe CFR et 1ère application

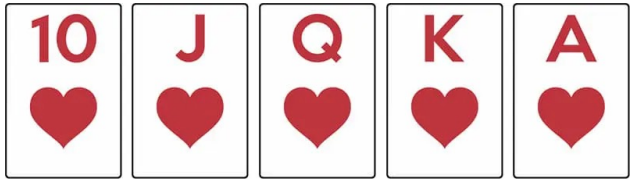
En testant notre algorithme dans la pratique avec python, on obtient des résultats intéressants :





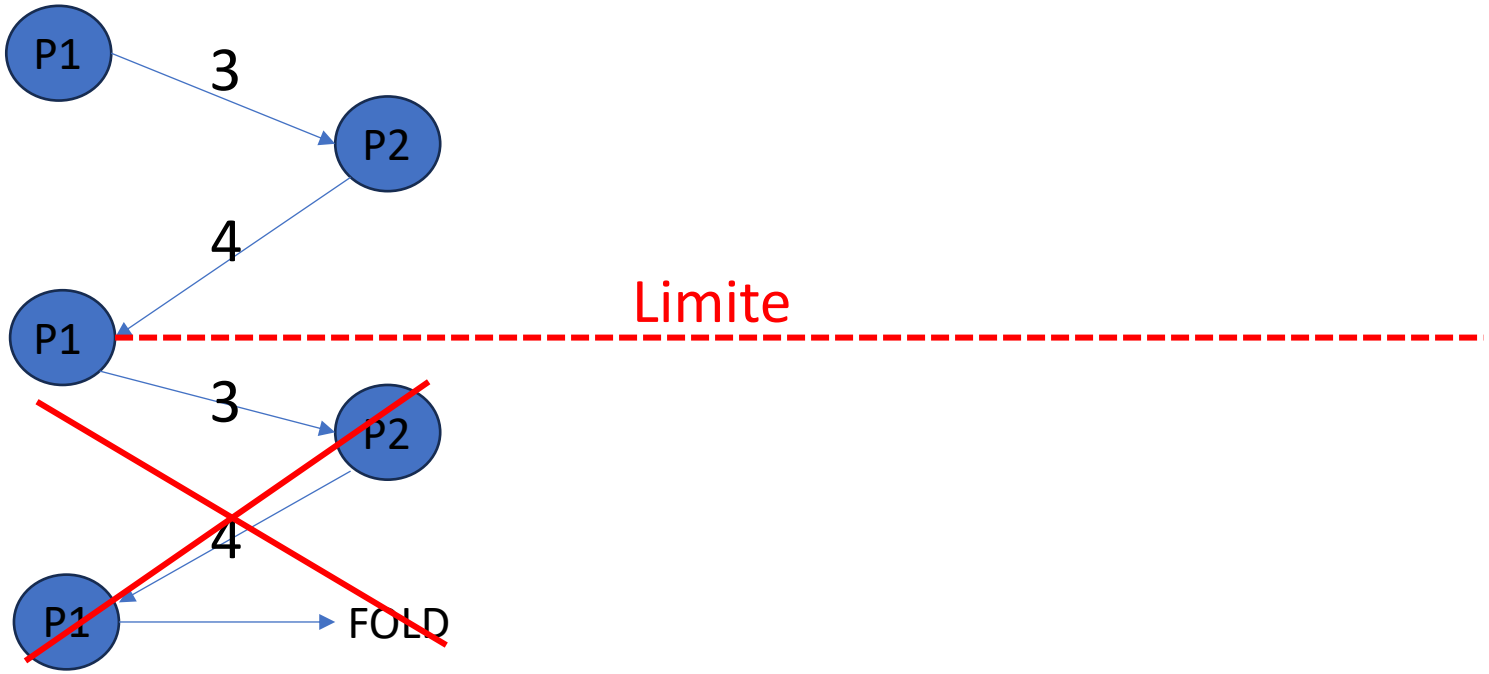
III)Modèle poker simplifié

- On utilisera une variante du poker un peu plus simple :
  - 5 niveaux de cartes au lieu de 13 (10 ,valet, roi , reine ,as)
  - Pas de turn ni de river (1 ou 2 tours)
  - une limite de bet sera fixée
  - Jetons de départ : 10
  - Une limite dans les tours de raise (seul 1 raise est possible)



III)Modèle poker simplifié

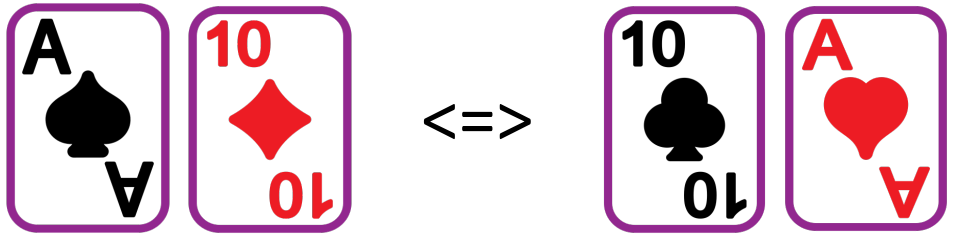
- Pour clarifier le dernier point :



IV)A) 1er Algorithme simple

- Pour cet algorithme, la limite de bet est de 1
- Il n'y a qu'un seul tour de bet
- Cet algo apprend de la manière suivante :
  - Pour chaque « état » il associe un certain % de bet et un certain % de fold
  - Pour cette version, un état c'est juste la main
    - Exemple : ((as de cœur, roi de pic) -> 70% de bet , 30% de fold
  - On utilise le CFR pour déterminer ces %, on fait jouer notre algo contre lui-même !

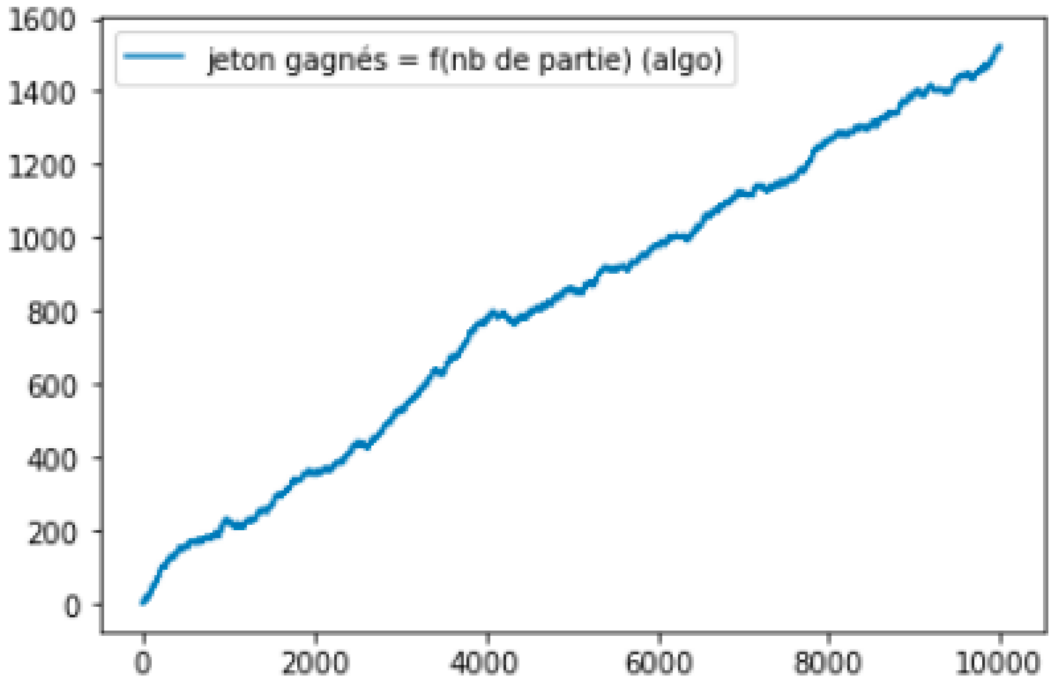
On utilise de l'abstraction, pour nous : les couleurs ne comptent plus



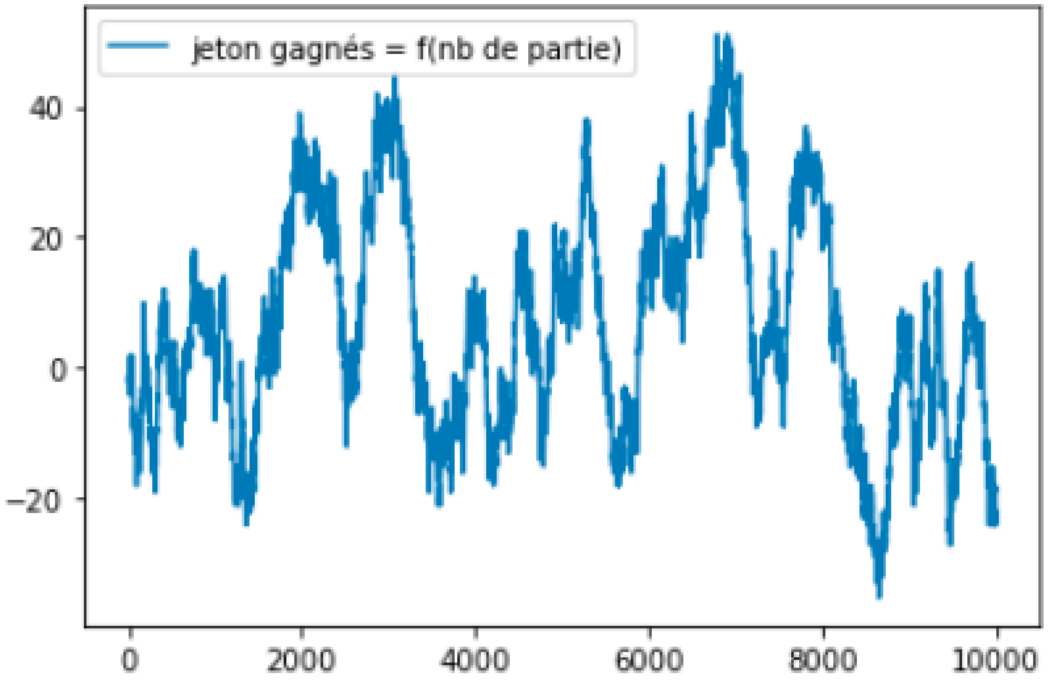
A) 1er Algorithme simple

Après entrainement, l'algorithme sort avec une stratégie : Fullbet (ne fait que jouer 1)

Algo vs Aléatoire



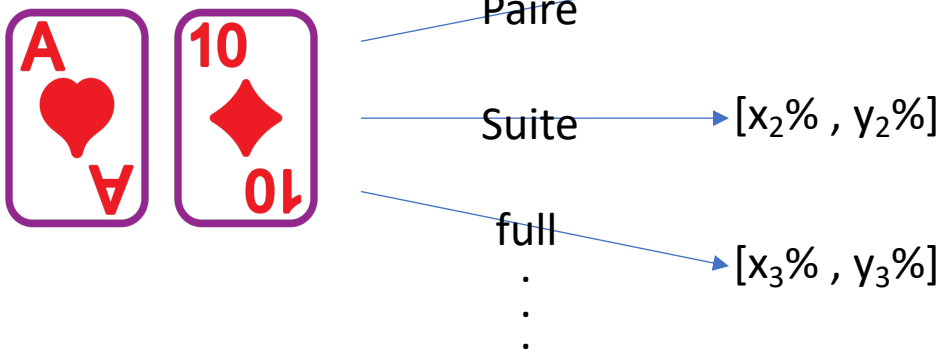
Aléatoire vs Aléatoire



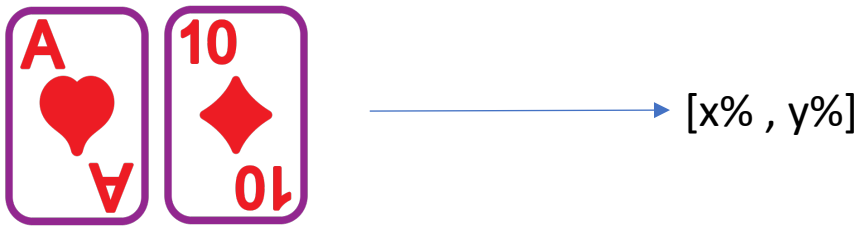
B) Une première amélioration

- Cette fois :
  - Différents niveaux de bet possibles (jusqu'à 4 jetons)
  - 2 tours (pre-flop et flop)
  - L'algorithme prend en compte les combinaisons, et le dernier coup de l'adv !

Algo complexe :

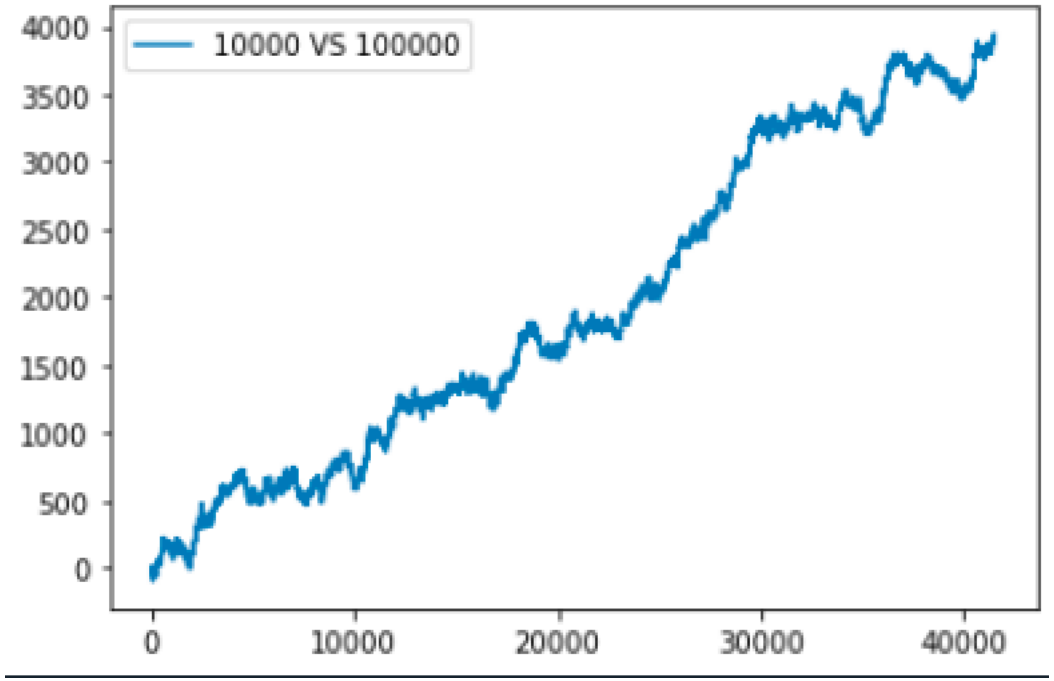


Algo simple :



B) Une première amélioration

- Problème : on ne sait pas quand l'algorithme atteint l'équilibre de Nash !



Algo(10k) vs Algo(100k)

B) Une première amélioration

- Les résultats après 100K de parties :
  - Pour mesurer « l'efficacité » de notre algorithme, on va le faire jouer contre plusieurs stratégies naïves.

```
In [3]: runfile('/Users/nanoisyourskill/Desktop/TIPE/poker_V4/
Poker_test_algo.py', wdir='/Users/nanoisyourskill/Desktop/TIPE/
poker_V4')
Reloaded modules: poker_variables, Poker_cmplx_fct_jeu,
Poker_fonction_entrainement
8751 VS 1249
```

Algo(100K) vs Aléatoire

```
In [4]: runfile('/Users/nanoisyourskill/Desktop/TIPE/poker_V4/
Poker_test_algo.py', wdir='/Users/nanoisyourskill/Desktop/TIPE/
poker_V4')
Reloaded modules: poker_variables, Poker_cmplx_fct_jeu,
Poker_fonction_entrainement
7411 VS 2589
```

Algo(100K) vs Fullbet

B) Une première amélioration

- Les limites de l'algo :

L'algorithme a aussi été testé contre une dizaine personnes (étudiants + profs), le taux de victoire étant < 50% (mais > 0%)

On peut donc conclure que l'Algo
n'est pas surhumain

C) Une deuxième amélioration

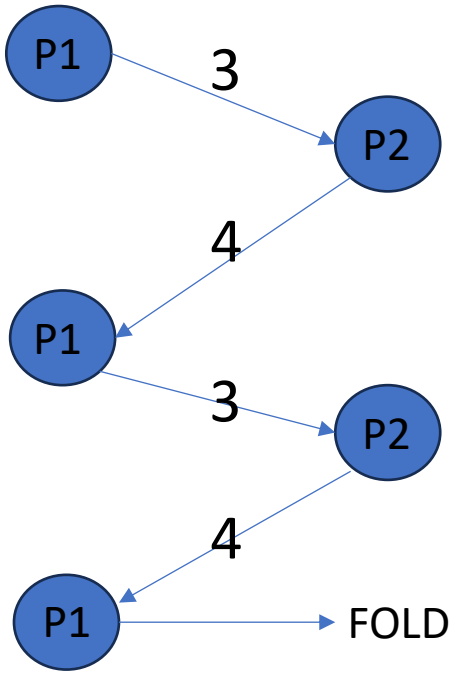
- Maintenant :
  - On monte à 7 cartes (8 9 10 J Q K A) (28 au total)
  - On va autoriser d'avoir **plusieurs tours de raise** !
  - Surtout, maintenant l'« état » correspond à :

État = {Main + combinaisons + l'ensemble de nos action (T1 + T2) + de l'adv (T1+T2)}

Problème : on a considérablement augmenté le nombre d'états à visiter, ce qui implique qu'un nombre très élevé de parties sera nécessaire.

Avec cela, l'algorithme est (en théorie) capable de devenir surhumain

Par exemple :



C) Une deuxième amélioration

- Place aux tests : (Algo 1M)
  - Algorithmes naifs :
    - Random :

```
In [38]: test_entre_2_strat(10000, M1_bot, random_bot)
8672 VS 1328
```

    - Fullbet :

```
In [37]: test_entre_2_strat(10000, M1_bot, fullbet_bot)
8618 VS 1382
```

  - Old VS New :
    - 20K (new) vs 300K (old) :

```
In [2]: test_entre_2_strat_1_ancienne(10000, K20_bot,
ancien_300K)
5498 VS 4502
```

D)Conclusion

- Limites : Mais au vu des nouvelles règles, malgré l'amélioration notable, l'humain reprends son règne (< 10% de WR, testé contre trentaine de personnes) : Il faut plus de parties.

V) Bibliographie

- Icones utilisées dans le diapo : **rizal2109** (cartes poker)
- Noam Brown | AI for Imperfect-Information Games like Poker and Beyond : www.youtube.com/watch?v=cn8Sld4xQjg
- Noam Brown | Superhuman AI for heads-up no-limit poker: Libratus beats top professionals : www.youtube.com/watch?v=2dX0lwaQRX0
- SDS 569: A.I. For Crushing Humans at Poker and Board Games — with Noam Brown : www.youtube.com/watch?v=5syG8Zkekx8
- (Regret Minimization in Games with Incomplete Information, Martin Zinkevich) https://poker.cs.ualberta.ca/publications/NIPS07-cfr.pdf

```python
import numpy as np

# Variables
regle = {"bet-fold" : 1, "fold-bet" : -1, "fold-fold" : 0, "jeton_base" : 20}
betPossible =  [i for i in range(5)]
nombreActions = len(betPossible)
nb_Tour = 2
carte_possible = ["H","N","T", "J", "Q", "K", "A"]

# Création des combi de toutes les cartes possibles dans une main
combi_main = [[carte_possible[i], carte_possible[j]] for i in range(len(carte_possible)) for j in range(i, len(carte_possible))]

Deck = [[f"{j}{i}" for i in range(1,5)] for j in carte_possible]

# Optimisation de la comparaison entre carte, on crée un dico associant à chaque carte son symbole
Deck_symb = {f"{i}{j}": i for i in carte_possible for j in range(1,len(Deck[0]) + 1)}
Deck_symb.update({i: i for i in carte_possible})

# On considère le poids suivant : H -> 1, N -> 2, T -> 3, J -> 4, Q -> 5, K -> 6, A -> 7.
poids_carte = {carte_possible[i]: i + 1 for i in range(len(carte_possible))}

# Dico des combinaisons possibles
combinaison= {"hauteur" : 0, "pair" : 1, "double_pair" : 2, "brelan" : 3,
              "suite" : 4, "flush" : 5,"full" : 6, "carre" : 7,  "quinte_flush" : 8}

# Dico des combinaisons + carte
combi_main_plus = [(i, j) for i in combi_main for j in combinaison if i[0] != i[1] or j != "hauteur"]
```

VI) Annexe

Implémentation des règles

```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))


    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))
```

VI) Annexe

Implementation du bot

```
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

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
```

VI) Annexe

VI) Annexe

- Fonctions de regret :

```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
```

```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) :
    # print(etat)
    # print(etat[0])
    # print(p1.action, len(etat[0]))
    actions_possibles = [i for i in betPossible if i <= jeton_P1 and i != p1.action[tour-1][0][len(etat[tour-1][0])]]
    # print("")
    # print(f"REGRET DE L'etat {etat} au tour {tour}")
    # print("")
    for action in actions_possibles :
        # print("")
        #print(f"on passe à l'action {action}")
        p1_reg = p1.copy(p1.name + "_reg")
        p2_reg = p2.copy(p2.name + "_reg")

        p1_reg.jeton = jeton_P1
        p2_reg.jeton = jeton_P2
        #print("l'etat est " + str(etat))
        p1_reg.action = copy.deepcopy(etat)
        #print(p1_reg.action)
        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)
        #print(p1_reg.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)
            #print(p1.strategy_manager.nbr_reg[tour-1][key])
            #print(f"le regret est de {recomp_P1 - recomp_initiale}")
```

VI) Annexe

VI) Annexe

```
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)
```

VI) Annexe

- Fonctions utiles pour la mise en place du jeu :

```python
def init_jeu(p1, p2):
    """
    Initialise le jeu pour deux joueurs, en distribuant les cartes et en réinitialisant l'état du jeu.
    """
    p1.reinitialiser_tour()
    p2.reinitialiser_tour()

    jeu = cree_jeu()  # Fonction à définir pour créer et mélanger un jeu de cartes
    p1.donner_carte(jeu[0])
    p2.donner_carte(jeu[1])

    # Réinitialisation des états des joueurs


    return jeu
```

```python
def equilibrage_main(j, tour, p1, p2, flop) :
    def equilibrage_aux(tour, p1, p2, flop) :
        # print(f"tour de {p1.name} :")
        # print(f"au total, {p1.name} a misé {sum(p1.action[tour - 1][0])}")
        # print(f"au total, {p2.name} a misé {sum(p2.action[tour - 1][0])}")
        decision_p1 = p1.choix_action(tour, flop)
        nouvelle_action = min(decision_p1, p1.jeton)  # Ne pas dépasser le montant de jetons disponible
        p1.action[tour - 1][0].append(nouvelle_action)
        p2.action[tour-1][1].append(nouvelle_action)
        p1.jeton -= nouvelle_action  # Réduire les jetons de p1 par le montant misé
        #print(f"{p1.name}  décide d'ajouter {decision_p1}")
        # print(f"jetons : {p1.name} -> {p1.jeton}, {p2.name}  -> {p2.jeton}")
        if sum(p1.action[tour - 1][0]) < sum(p2.action[tour - 1][0]):
            if p1.jeton == 0 :
                #print(f"{p1.name}  a donc all in")
                p1.is_all_in = True
                p1.is_check = False
                p2.is_check = False
                p2.jeton += sum(p2.action[tour-1][0]) -sum(p1.action[tour-1][0])
                p2.action[tour-1][0][-1] -= (sum(p2.action[tour-1][0]) -sum(p1.action[tour-1][0])) # je lui abaisse sa dernière action
                p1.action[tour-1][1][-1] -= (sum(p2.action[tour-1][0]) -sum(p1.action[tour-1][0]))
                 # et je lui rends les jetons
                return
            #print(f"{p1.name}  a donc fold")
            p1.is_fold = True
            # p1 se couche si son action est inférieure à celle de p2
            p1.is_check = False
            p2.is_check = False
            return
        elif sum(p1.action[tour - 1][0]) == sum(p2.action[tour - 1][0]):
            #on doit prendre en compte la possibilité d'un check eventuelle
            if p1.action[tour-1][0][-1] == 0 and not p2.is_check:
                p1.is_check = True
                equilibrage_aux(tour, p2, p1, flop) #on laisse donc l'opportunité à l'adversaire de réagir

            return  # Les deux joueurs sont alignés, aucune nouvelle relance
        else:
            #print(f"{p1.name}  a donc raise")
            # Continuer si p1 relance
            p1.is_check = False
            p2.is_check = False
            return equilibrage_aux(tour, p2, p1, flop)
    if j%2 == 0 :
        equilibrage_aux(tour, p1, p2, flop)
    if j%2 == 1 :
        equilibrage_aux(tour, p2, p1, flop)
```

VI) Annexe

VI) Annexe

```python
def prendre_decision_jeu(j, tour, p1, p2, flop):
    def prendre_decision_jeu_aux(tour, p1, p2, flop) :
        # Joueur 1 prend une décision basée sur sa stratégie
        p1.prendre_decision(tour, flop, p2)
        p1.action[tour-1][0][-1] = max(1, p1.action[tour-1][0][-1])#car il commence donc mise obligatoire
        p2.action[tour-1][1][-1] = max(1, p1.action[tour-1][0][-1])

        # Joueur 2 prend une décision en réponse
        p2.prendre_decision(tour, flop, p1)

        #on enleve les jetons
        p1.jeton -= sum(p1.action[tour -1][0])
        p2.jeton -= sum(p2.action[tour-1][0])
    if j%2 == 0 :
        prendre_decision_jeu_aux(tour, p1, p2, flop)
    if j%2 == 1 :
        prendre_decision_jeu_aux(tour, p2, p1, flop)



def jouer_partie(j, tour, p1, p2, flop, pot):
    #print(f"début du tour {tour}")
    # on joue le partie
    equilibrage_main(j, tour, p1, p2, flop)
    pot += sum(p2.action[tour-1][0]) + sum(p1.action[tour-1][0])
    #print(f"p1 action finale est {p1.action[tour-1]}, p2 action finale est {p2.action[tour-1]}")
    return pot
```

VI) Annexe

```python
def conclusion_partie(p1, p2, flop, pot):
    # Calcul des actions totales pour chaque joueur pour référence
    mon_action = sum([sum(p1.action[i][0]) for i in range(nb_Tour)])
    en_action = sum([sum(p2.action[i][0]) for i in range(nb_Tour)])

    # Déterminer le résultat basé sur les plis et les mains des joueurs
    if not p1.is_fold and not p2.is_fold:
        ma_recomp = prendre_recompense(pot, p1.main, p2.main, flop)
        if ma_recomp == False :  # Supposé que prendre_recompense retourne False en cas d'égalité
            ma_recomp = pot // 2
            en_recomp = pot // 2
        else:
            en_recomp = -ma_recomp
    elif p1.is_fold:
        ma_recomp = -mon_action
        en_recomp = pot
    elif p2.is_fold:
        ma_recomp = pot
        en_recomp = -en_action

    # Mettre à jour les jetons après calcul des récompenses
    #print(f"jetons avant pot : p1 -> {p1.jeton}, p2 -> {p2.jeton}")
    p1.jeton += max(ma_recomp, 0)
    p2.jeton += max(0, en_recomp)
    #print(f"jetons après pot : p1 -> {p1.jeton}, p2 -> {p2.jeton}")
    return (ma_recomp, en_recomp)
```

```python
def entrainer(i):
    W, L = 0, 0

    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)
        # Comptage des victoires et défaites
        if p1.jeton <= 0:
            L += 1
        if p2.jeton <= 0:
            W += 1

    print(W, L)
```

VI) Annexe

Fonction d'entrainement