Projet Exploration/Exploitation

Le dilemme dit de l'exploration vs exploitation est un problème fondamental que l'on retrouve dans plusieurs domaines de l'intelligence artificielle : parmi un certain nombre de choix possibles, vaut-il mieux exploiter la connaissance acquise et choisir l'action estimée la plus rentable ou vaut-il mieux continuer à explorer d'autres actions afin d'acquérir plus d'informations ? L'exploitation consiste à faire la meilleure décision à partir de toute l'information collectée, l'exploration consiste à obtenir plus d'information. Il est parfois préférable, souvent au début d'un processus, de faire des sacrifices et de ne pas choisir l'option a priori la plus rentable afin d'améliorer le gain à long terme. Mais la question reste de savoir quand arrêter d'explorer, quand on estime que l'exploration n'apportera pas de connaissances supplémentaires.
L'objectif de ce projet est d'étudier différents algorithmes du dilemme d'exploration vs exploitation pour des IAs de jeu.

1 Bandits-manchots

Dans cette partie, afin de formaliser le problème de l'exploration/exploitation, on prend l'exemple des bandits manchots (ou machine à sous). Pour une mise, on a le droit d'actionner un levier qui fait tourner des rouleaux, et en fonction de la combinaison obtenue sur les rouleaux, une récompense est attribué au joueur. Supposons une machine à sous à N leviers dénotés par l'ensemble$\{1,2,...,N\}$. Chacun de ses leviers est une action possible parmi lesquelles le joueur doit choisir à chaque pas de temps.

Notons pour un joueur, le gain au bout de T parties, qui correspond a la somme des récompenses qu'il a obtenu pendant les T premières parties. (Le but de chaque joueur est évidemment de maximiser ce gain. Pour cela, le joueur doit identifier le levier au rendement le plus élevé.)

On appelle regret au temps T la différence entre le gain maximal espéré et le gain du joueur.

Nous étudions dans la suite les algorithmes suivants :

• l'algorithme aléatoire : qui choisit une action
uniformément parmi toutes les actions possibles.
(un algorithme référence que tous les autres algorithmes doivent battre.)

• l'algorithme greedy (ou glouton) : un certain nombre d'itérations sont consacrés au début à l'exploration (on joue uniformément chaque levier) puis par la suite on choisit toujours le levier dont le rendement estimé est maximal. Cet algorithme fait purement de l'exploitation.

•l'algorithme ε -greedy: après une première phase d'exploration optionnelle, à chaque itération, avec une probabilité ε on choisit au hasard uniformément parmi les actions possibles, avec une probabilité ε-1
Cet algorithme explore continuellement.

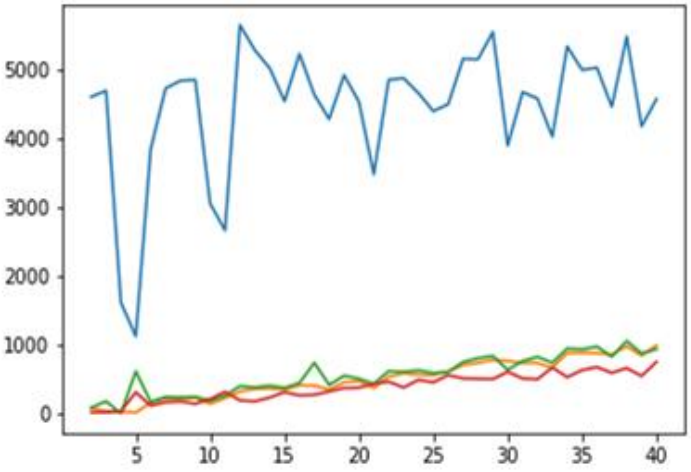
•l’algorithme UCB :
Le premier terme est identique aux autres algorithmes, il garantit l’exploitation ; le deuxième terme lui devient important lorsque le ratio entre le nombre de coups total et le nombre de fois où une action donnée a été choisie devient grand, c’est-à-dire qu’un levier a été
peu joué : il garantie l’exploration.

Expériences

Toutes les expériences que nous avons réalisés ont pour objectif d’observer l’évolution du regret en fonction de différents paramètres afin de pouvoir conclure sur quels paramètres jouer pour minimiser ce regret (maximiser le gain du joueur)

– Premièrement, nous avons observé l’évolution du regret en fonction du nombre de leviers (sur 10000 itérations).

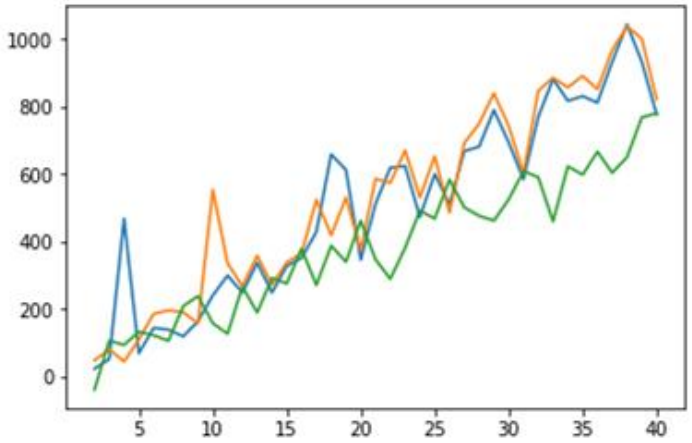
Algorithme aléatoire : bleu
Algorithme greedy : jaune
Algorithme ε -greedy : vert
Algorithme UCB : rouge



Nous pouvons donc observer que, pour l’algorithme aléatoire, le regret ne dépend pas du nombre de leviers, c’est aléatoire.

Par soucis de visibilité, nous avons retracé les courbes, sans celle de l’algorithme aléatoire (car les ordres de grandeur ne sont pas les mêmes).

Algorithme greedy : bleu
Algorithme ε -greedy : jaune
Algorithme UCB : vert
Nous pouvons donc constater que les algorithmes dits intelligents, le regret augmente de manière linéaire, avec un coefficient directeur égal à 20.

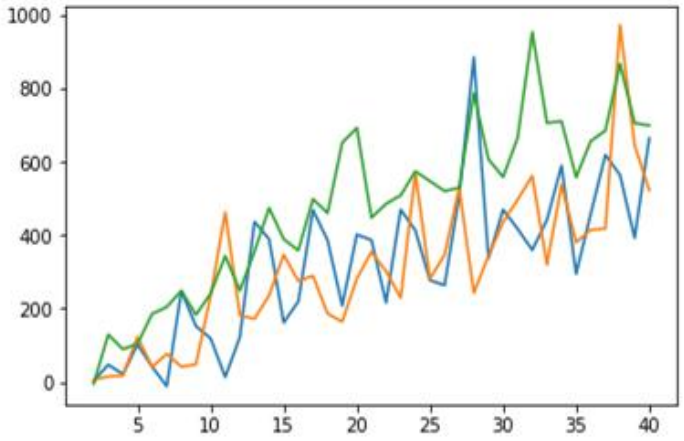


– Nous avons ensuite regardé l'évolution du regret en fonction du nombre de leviers mais en faisant varier les probabilités de victoire (jusqu'ici, les leviers avaient une probabilité de victoire comprise entre 0 et 1).

Pour des probabilités de victoire comprises entre 0 et 0,5 :

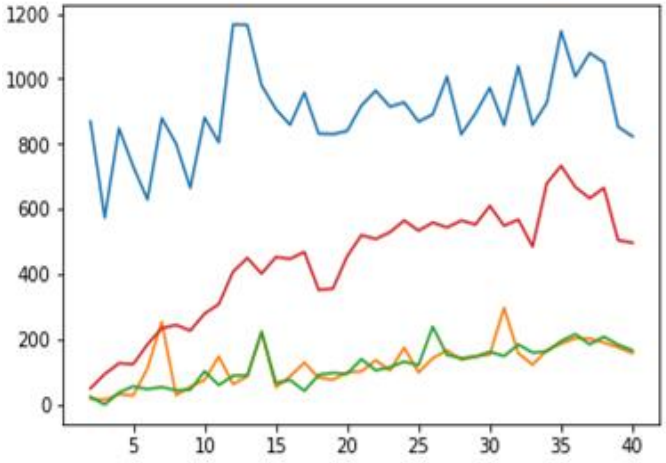
Par soucis de visibilité, nous n'avons pas représenté la courbe de l'algorithme aléatoire, car son regret est proportionnel à la différence entre le gain maximum espéré et le gain moyen.

Algorithme greedy : bleu
Algorithme ε -greedy : jaune
Algorithme UCB : vert



Pour des probabilités de victoire comprises entre 0,8 et 1 :

Algorithme aléatoire : bleu
Algorithme greedy : jaune
Algorithme ε -greedy : vert
Algorithme UCB : rouge



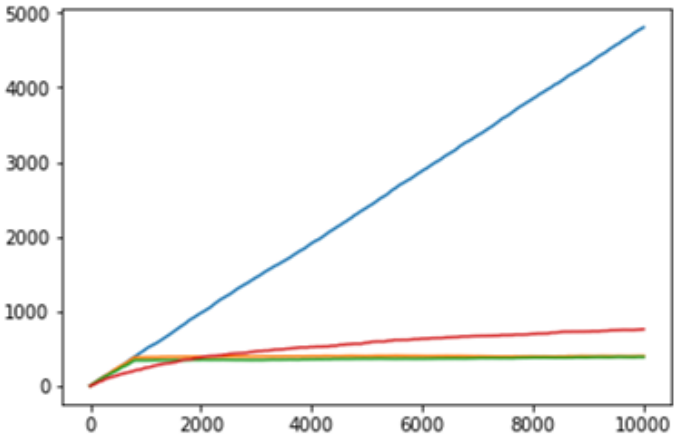
On peut constater que lorsque les probabilités de victoire se concentrent, l'algorithme UCB perd en puissance par rapport aux autres algorithmes déterministes présentés, car il se retrouve rapidement dans l'obligation de jouer d'autres leviers que le meilleur.

On remarque que l'évolution du regret dépend de la répartition des récompenses car les tendances sur l'aléatoire se retrouvent sur les autres courbes.

– Par la suite, nous avons observé l'évolution du regret en fonction du nombre d'itérations.

Nb : Le regret peut être négatif si le joueur est chanceux sur une petite série. En effet, le taux de victoire temporaire peut être supérieur au taux de victoire théorique.

Pour 40 leviers,



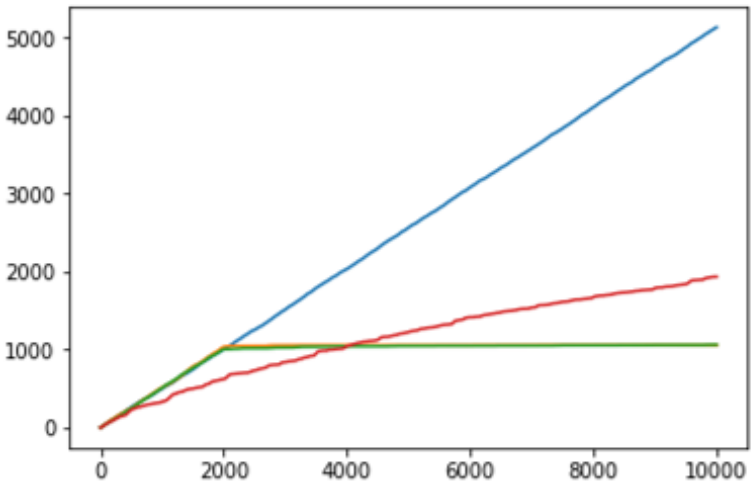
Algorithme aléatoire : bleu
Algorithme greedy : jaune
Algorithme ε -greedy : vert
Algorithme UCB : rouge

Observations ;
-ε -greedy et greedy, après la periode d'apprentissage, où ils sont exactement en phase avec l'aléatoire, déterminent le levier le plus fort et ne jouent plus que sur celui là. Ainsi, le regret est très proche de zéro si on exclut la phase d'apprentissage de ces algorithmes.

-l'algorithme aléatoire, l'évolution de son regret est linéaire, grâce au fait que son regret est égal à (gain de la meilleure machine – moyenne des gains des machines)

-l'algorithme UCB, lui, est directement efficace car sa phase d'apprentissage est plus courte. Mais à long terme, son regret est plus élevé.

Pour 200 leviers,

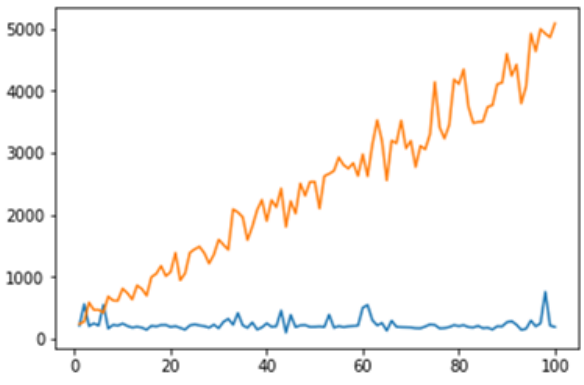


On peut constater que la courbe représentant l'évolution du regret de l'algorithme UCB(rouge) augmente. En effet, plus le nombre de leviers augmente, plus son regret est élevé car il est forcé de jouer sur des machines sur lesquels il n'a pas joué depuis longtemps. Ainsi, le regret de l'UCB se rapproche du regret de l'aléatoire si le nombre de leviers se rapproche du nombre d'itérations.
La taille de la phase d'apprentissage imposée aux algorithmes de type glouton est proportionnelle aux nombres de leviers dans notre code, ce qui explique la différence sur leur courbe par rapport à la précédente.

– Ici, nous avons tracé l'évolution du regret de l'algorithme ε -greedy en fonction de epsilon en %(la probabilité de choisir l'algorithme aléatoire à la place de l'algorithme greedy). Pour pouvoir analyser ces résultats, nous avons choisi de les comparer à l'évolution du regret de l'algorithme greedy.

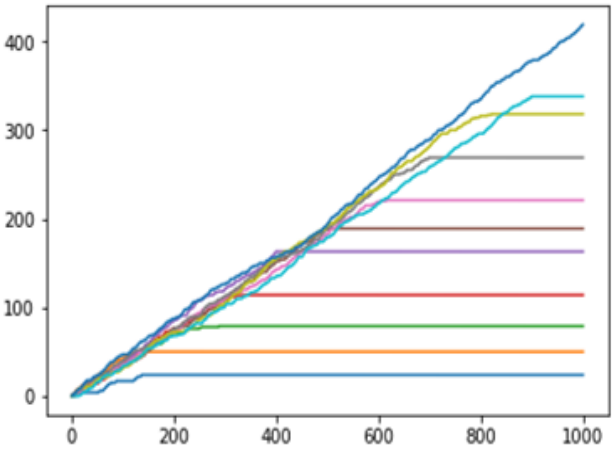
Algorithme greedy : bleu
Algorithme ε -greedy : orange

On remarque que lorsque ε augmente, soit la probabilité de choisir l'aléatoire par rapport au greedy, plus le regret augmente et se rapproche du regret de l'algorithme aléatoire (qui n'est pas tracé ici, mais a un regret autour de 5000)



– Enfin, nous avons tracé l'évolution du regret de l'algorithme greedy en fonction du temps, en augmentant la phase d'apprentissage de 10 % du nombre d'itérations. (entre 0 et 100%)

Plus la phase d'apprentissage est grande, plus le regret augmente car le nombre d'itérations sur lequel il apprend augmente donc le nombre de fois où il choisit l'algorithme aléatoire augmente.



On remarque que même lorsque la phase d'apprentissage est de zéro, ses résultats sont bons (bien qu'au début son regret augmente légèrement car il joue au moins une fois sur chaque levier).

Ainsi, on en déduit qu'il n'est pas utile d'imposer une très grande phase d'apprentissage car cela fait augmenter le regret inutilement (en effet, on ne peut pas revenir dessus).

Conclusion

Après avoir fait varier plusieurs paramètres et analysé les résultats obtenus, nous pouvons en conclure que l'objectif est de minimiser tous les paramètres qui augmentent le regret inutilement ainsi que le temps nécessaire pour trouver le meilleur levier.

C'est ainsi que nous pouvons conclure que parmi les algorithmes proposés, l'algorithme greedy est le plus efficace et le plus recommandable dans ce cas de figure.

2 Morpion et Monte-Carlo

Le jeu du Morpion se joue à 2 joueurs sur un plateau de 3×3 cases. Chaque joueur à son tour marque une case avec son symbole (croix ou cercle) parmi celles qui sont libres. Le but est d'aligner pour un joueur 3 symboles identiques en ligne, en colonne ou en diagonale.

Nous avons implémentés deux algorithmes : (annexe page 13)

- l'algorithme du joueur aléatoire qui joue de manière aléatoire.

- l'algorithme de Monte-Carlo qui, initialise les récompenses des actions à 0 (elles représentent la moyenne des victoires par action)puis, pour chaque action, en choisit une au hasard et tant que la partie n'est pas finie, joue les deux joueurs au hasard puis met à jour la récompense de l'action en fonction du résultat puis retourne l'action avec la meilleure probabilité de victoire

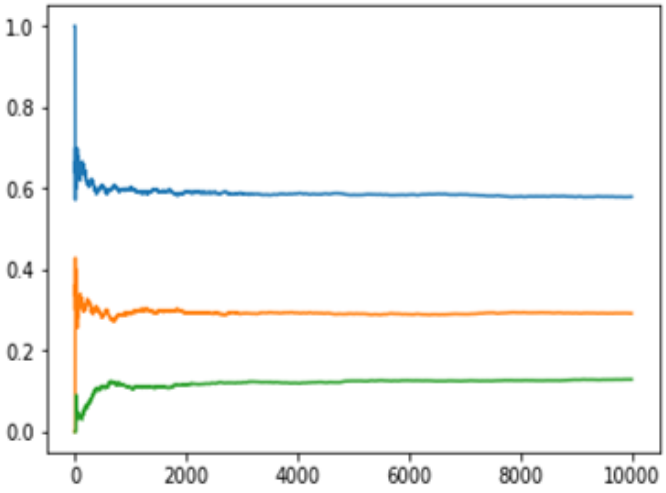
Expériences

– Nous avons fait jouer deux joueurs aléatoires afin d'observer le pourcentage de victoires, de défaites, et de matchs nuls.

Joueur 1(aléatoire) : bleu

Joueur 2(aléatoire) :jaune

Égalité : vert



On observe que le joueur 1 a environ 58 % de victoires a son actif. Le joueur 2 : 30 % et il y a 12 % de matchs nuls.

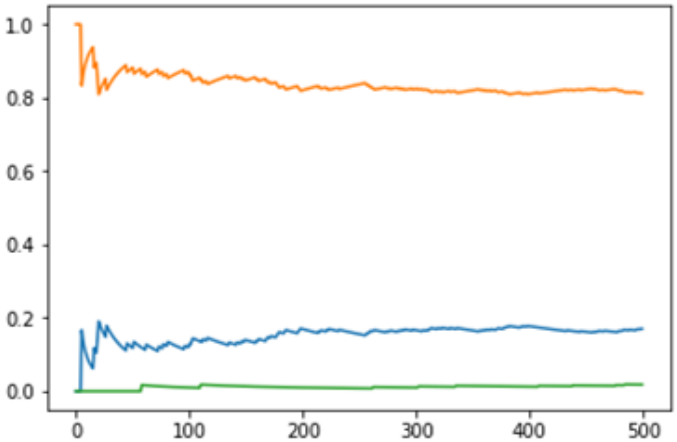
On en conclut qu'il est largement avantageux de jouer en premier pour deux joueurs aléatoires.

– Joueur Alea vs Joueur Monte-Carlo

Joueur 1(Aléatoire):bleu

Joueur 2(Monte-Carlo):jaune

Égalité : vert

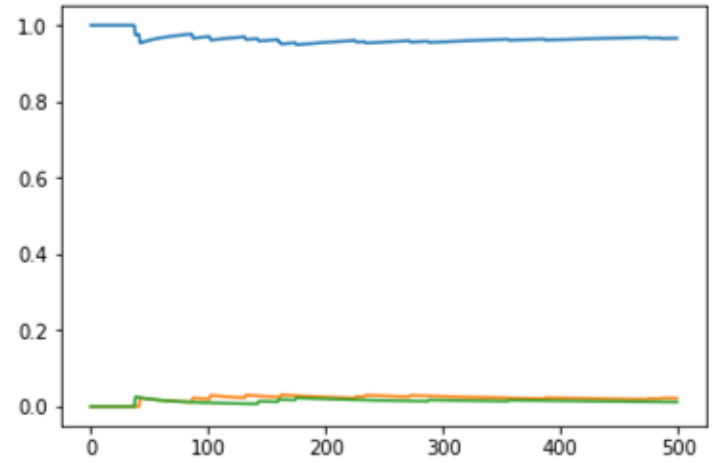


21.41241693496704
0.17 0.812 0.017999999999999905

La remarque faite ci dessus ne s'applique pas dans ce cas. En effet, bien que le joueur Alea joue en premier, il se fait largement battre par le joueur Monte-Carlo et le pourcentage de matchs nuls est faible (1,8% environ)

Nous avons ensuite inversé l'ordre de commencement. Nous avons donc fait jouer le joueur Monte-Carlo en premier :

Dans ce cas là, le joueur Monte-Carlo a 97 % de victoires à son actif. Et le joueur aléatoire à environ le même pourcentage de victoires qu'il y a d'égalités (soit entre 1 % et 2%)



26.605039596557617
0.966 0.022 0.012000000000000031

– Enfin, nous avons fait jouer deux joueurs Monte-Carlo l'un contre l'autre :

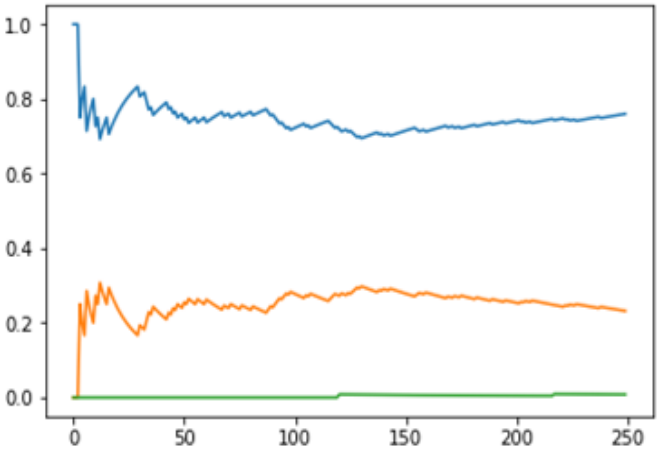
Joueur 1(Monte-Carlo):bleu

Joueur 2(Monte-Carlo):jaune

Égalité : vert

On remarque que le joueur Monte-Carlo est plus fort lorsqu'il joue en premier. (77 % de victoires)

La différence de pourcentage de victoires est plus marquée que dans la partie Alea vs Alea.



44.29265475273132
0.76 0.232 0.00799999999999998

Conclusion

Il semble que pour le jeu du morpion il est avantageux de jouer en premier, qu'on soit un joueur expérimenté ou non.

En effet, dans un duel entre deux joueurs 'aléatoires' celui qui joue le premier voit ses chances de victoire être le double de celles de son adversaire. Dans un duel entre un joueur aléatoire et un joueur suivant un algorithme de Monte-Carlo, on remarque que ce dernier a bien plus de difficultés à ne pas concéder de victoires à son adversaire lorsqu'il joue en second, même avec un grand nombre de parties jouées en interne pour prévoir chaque coup.

Enfin, dans un duel entre deux joueurs avec une stratégie de Monte-Carlo, la tendance se retrouve une nouvelle fois, encore plus prononcée car le premier joueur voit son taux de victoire trois fois supérieur à celui du second joueur, bien qu'ils aient exactement la même stratégie.

3 Morpion et UCT

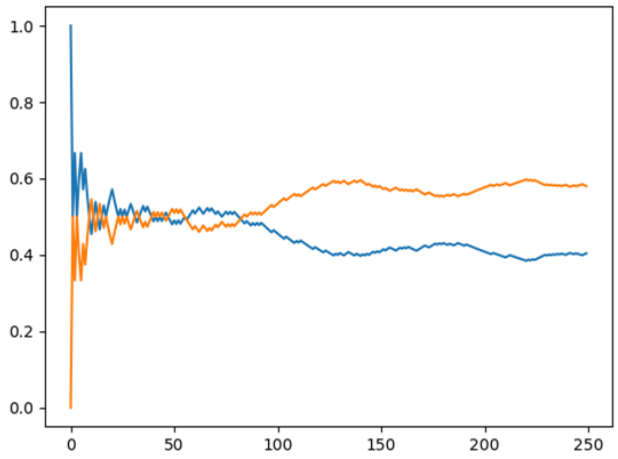
Dans cette partie, nous avons tenté une implémentation de l'algorithme UCT appliqué au morpion.

Pour différentes valeurs du nombre de parties jouées en interne à chaque coup, notre algorithme (joueur 1) a du mal à atteindre les 50% de victoire contre un Monte-Carlo (joueur 2), ce qui est sûrement lié à un problème venant de notre implémentation.

Voici deux courbes de résultats des algorithmes en fonction du temps, pour N = 40 :

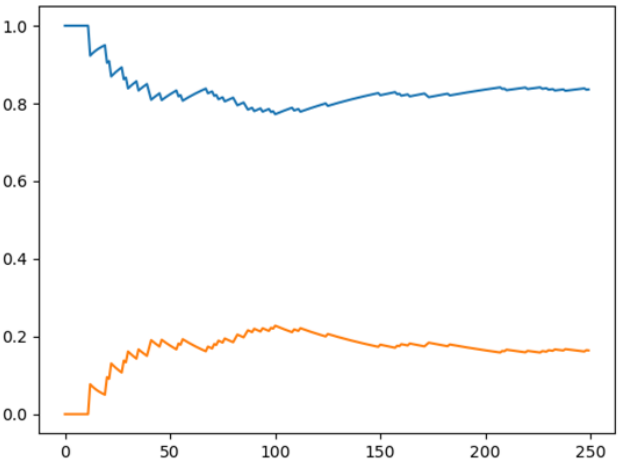
Joueur 1(UCT):bleu

Joueur 2(Monte-Carlo):jaune



Joueur 1 (Monte-Carlo): bleu

Joueur 2(UCT): jaune



4 Puissance 4 et Monte-Carlo

Nous avons ensuite implémenté un puissance 4, plateau de 6 cases de haut et 7 cases de large.

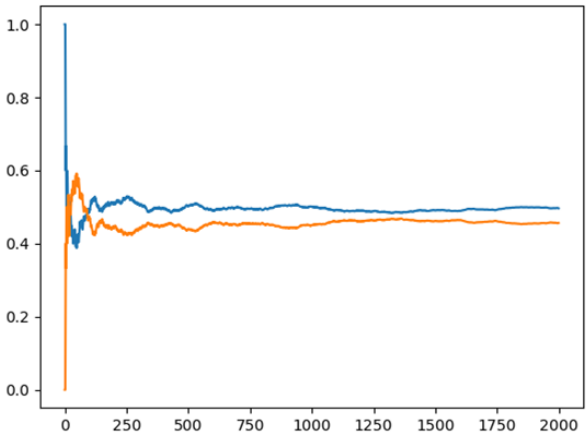
Le code de la classe est joint en annexe (page 11).

Lors d'un premier test, nous faisons jouer deux joueurs aléatoire l'un contre l'autre et remarquons que contrairement au morpion, il ne semble pas y avoir d'avantage à jouer en premier.

Dans le cas de deux joueurs aléatoires, on note environ 4.5% d'égalité au cours des parties

Joueur 1 (Alea): bleu

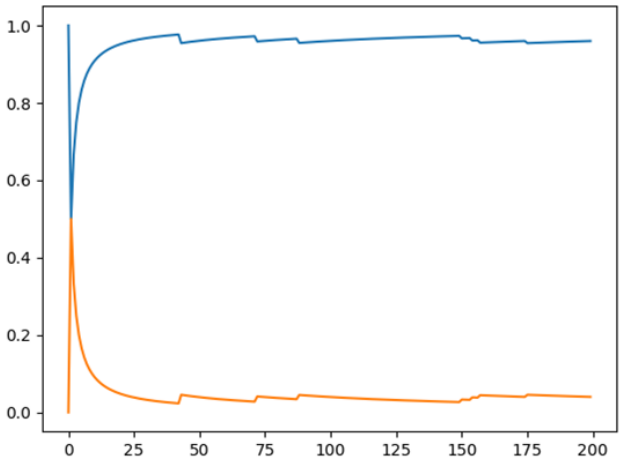
Joueur 2(Alea): jaune



Ici, on fait jouer un joueur Monte-Carlo (N = 40) contre un joueur aléatoire. On observe alors moins de 1% d'égalités entre les deux joueurs et un taux de victoire de 94% pour le Monte-Carlo. Lorsqu'on augmente N jusqu'à N = 80, on obtient un taux de victoire de 98% pour le Monte-Carlo. Ces valeurs se retrouvent lorsque l'on inverse l'ordre des joueurs.

Joueur 1 (Monte-Carlo): bleu

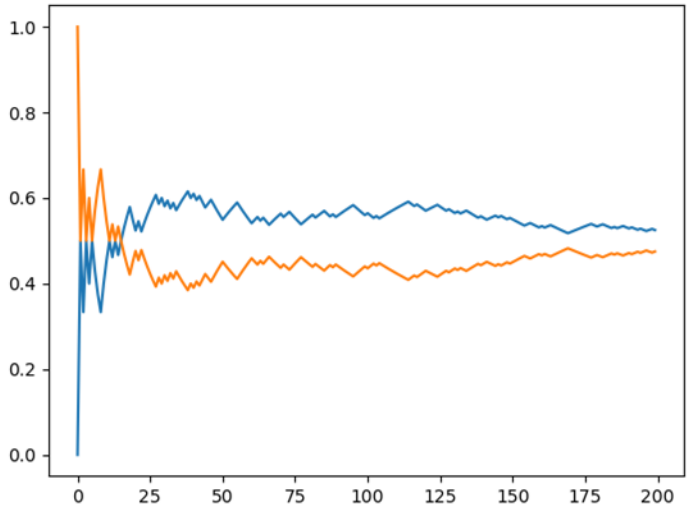
Joueur 2(Alea): jaune



On constate que lorsque l'on fait jouer deux joueurs Monte-Carlo l'un contre l'autre, on retrouve un équilibre entre le joueur 1 et le joueur 2 ce qui tend à confirmer qu'il y a peu voire pas d'avantage à jouer premier à ce jeu.

Joueur 1 (Monte-Carlo): bleu

Joueur 2(Monte-Carlo): jaune



On en conclut donc que s'il existe un avantage à jouer premier à ce jeu, il semble minime pour les joueurs avec lesquels on a pu expérimenter.

Conclusion

Apres avoir étudié différents algorithmes adaptés à différentes tactiques de jeu, nous nous sommes rendus compte que l'utilité des algorithmes dépend de la situation dans laquelle on les utilise. Ainsi, un algorithme très fort dans un jeu, peut ne pas du tout être efficace dans un autre. Il faut donc bien comprendre la stratégie du jeu avant d'y faire jouer un de ces algorithmes intelligents. Nous avons également remarqué que l'utilisation d'un algorithme aléatoire permet d'obtenir des informations pour mieux appréhender la suite des recherches et le choix des algorithmes à utiliser.

Annexe

Code de la classe Puissance4 :

```
class Puis4State(State):
NX,NY = 7,6

def __init__(self,grid=None,courant=None):
super(Puis4State,self).__init__(grid,courant)

def next(self,coup):
state =  Puis4State(self.grid,self.courant)
for i in range(len(state.grid)):
if state.grid[i][coup] !=0:
state.grid[i-1][coup]=self.courant
break
if i == len(state.grid)-1:
state.grid[i][coup]=self.courant
break
state.courant *=-1
return state

def get_actions(self):
res = []
for i in range(len(self.grid[0])):
if self.grid[0][i] == 0:
res.append(i)
return res
```

```
def win(self):
for i in range(len(self.grid)-3):
for j in range(len(self.grid[i])-3):
#print(i,j)
t =  self.grid[i][j]
victoire = False
if t!= 0:
if (self.grid[i+1][j] == t and  self.grid[i+2][j] == t and  self.grid[i+3][j] == t):
victoire = True
if (self.grid[i][j+1] == t and  self.grid[i][j+2] == t and  self.grid[i][j+3] == t):
victoire = True
if (self.grid[i+1][j+1] == t and  self.grid[i+2][j+2] == t and  self.grid[i+3][j+3] == t):
victoire = True
if victoire:
return t
return 0

def stop(self):
return self.win()!=0 or (self.grid==0).sum()==0
def __repr__(self):
return str(self.hash())
```

Code des fonctions de décision des agents aléatoire et Monte-Carlo :

Aleatoire:

```
def get_action(self,state):
return random.choice(state.get_actions())
```

Monte-Carlo:

```
def get_action(self,state):
N = 40
lenstate = len(state.get_actions())
res = [0.0]*lenstate
res2 = [0]*lenstate
if(lenstate > 1):
for i in range(1,N+1):
index = random.randint(0,lenstate-1)
a = state.get_actions()[index]
nouvelEtat = state.next(a)
j1tmp = Agent_alea()
j2tmp = Agent_alea()
jeutmp = Jeu(nouvelEtat,j1tmp,j2tmp)
vic,useless = jeutmp.run()
if state.courant == vic:
res[index]+=1
res2[index]+=1
for i in range(lenstate):
if res2[i] != 0:
res[i] = res[i] / res2[i]
return state.get_actions()[res.index(max(res))]
return state.get_actions()[0]
```