Rapport : Projet Exploration/Exploitation

Hakim CHEKIROU & Idles MAMOU

27 fevrier 2019

Table des matières

Introduction

Le dilemme de l'exploration vs exploitation est un problème fondamental en informatique qui consiste entre choisir d'exploiter la connaissance acquise et choisir l'action estimée la plus rentable ou continuer a explorer. Toute la difficulté réside dans le fait de savoir quand l'exploration n'apportera plus d'informations supplémentaires et qu'il vaut mieux exploiter la connaissance acquise.
Ce problème est très présent dans le secteur de l'Epub (publicités en ligne) ainsi que les jeux de stratégies.
Le but de ce projet est d'étudier différents algorithmes du dilemme d'exploration vs exploitation pour des IAs de jeu. Le jeu étudié sera dans un premier temps le morpion. La partie 1) est dédiée à l'expérimentation des algorithmes classiques d'exploration vs exploitation dans un cadre simple. Dans la partie 2 sera implémenté un algorithme de Monte-Carlo pour la résolution du jeu du morpion et la partie 3 est consacrée à l'étude des algorithmes avancés pour les jeux combinatoires.

Chapitre 1

Bandits-manchots

Pour formaliser le problème de l'exploration/exploitation, on prend l'exemple des bandits-manchots ou machine à sous. Le principe est simple , on a le droit d'actionner un levier pour une mise et selon le résultat on obtient une récompense.
Supposons une machine a sous a N leviers dénotés par l'ensemble $\{1, 2, ..., N\}$. Le joueur doit choisir un des leviers, l'action choisie à l'instant t sera appelée $a_t \in \{1, ..., N\}$. la récompense associée a chaque levier i suit une distribution de Bernoulli de paramètre μ^i, la récompense est de 1 avec une probabilité μ_i et 0 avec une probabilité de $1 - \mu_i$, récompense qui sera dénoté r_t au temps t. On suppose aussi que les μ_i sont constants tout au long de la partie. Pour le joueur, le but est de maximiser le gain au bout de T parties $G_t = \sum_{t=0}^{T} r_t$. Il doit donc identifier le levier au meilleur rendement $i^* = argmax_{i\in\{1,...,N\}}\mu_i$et le rendement associé $\mu* = \mu^{i^*} = max_{i\in\{1,...,N\}}\mu_i$. si un autre levier que i^* est choisi il aura en moyenne un gain total inférieur au gain maximal qu'il peut espérer.Ce gain maximal a un temps T s'écrit $G_T^* = \sum_{i=1} Tr_t^*$ avec r_t la récompense aléatoire tirée de la distribution de Bernoulli de paramètre μ^*. Le but est donc de minimiser le regret qui est la différence entre le gain maximal et le gain du joueur, dénotée ici au temps T $L_T = G^* - \sum_{t=1}^{T} r_t \sum_{t=T}^{T} (r_t^* - r_t)$. Pour ce faire nous allons étudier 4 algorithmes, puis les tester individuellement pour observer leurs caractéristiques, nous procéderons a la fin a des tests comparatifs en faisant varier plusieurs paramètres

1.1 Algorithme aléatoire

Cet algorithme est très simple vu qu'il ne fait que choisir un levier uniformément entre les N disponibles. ce sera notre baseline.

1.1.1 Tests

Nous avons procédé à un premier test pour étudier cet algorithme. Nous l'avons fait tourner 1000 fois sur une distribution uniforme de 50 leviers.

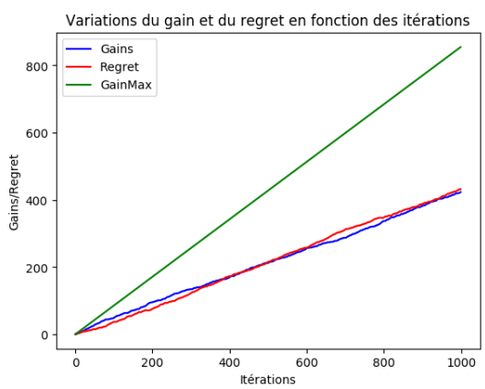

FIGURE 1.1 – test algorithme aléatoire

observation Nous remarquons que les tracés du gain et du regret se superposent et qu'a la fin des itérations nous avons $G_t = L_T$.

1.1.2 Conclusion

On en conclut qu'en optant pour la stratégie aléatoire, le joueur perd tout autant qu'il gagne, et il obtient la moitié des gains espérés.

1.2 Algorithme greedy

Dans cette stratégie, on joue uniformément chaque levier pendant k itérations. Par la suite on choisit toujours le levier dans le rendement estimé est maximal.

1.2.1 Tests

Pour étudier les caractéristiques de cette stratégie,nous avons procédé à deux tests. Pour le test représenté sur la figure 2 : nous avons fait tourner l'algorithme 1000 fois sur une distribution uniforme de 50 leviers avec $k = 200$. Quant au test représenté sur la figure 2 nous avons mis k à 5.

Résultat

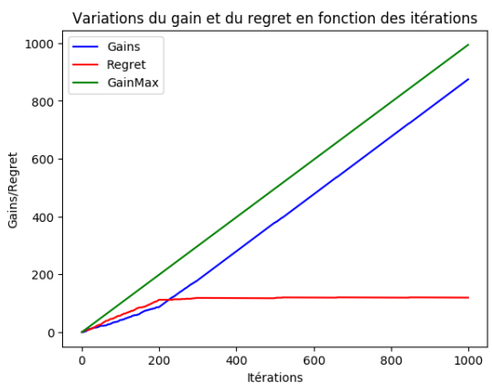

FIGURE 1.2 – $k = 200$

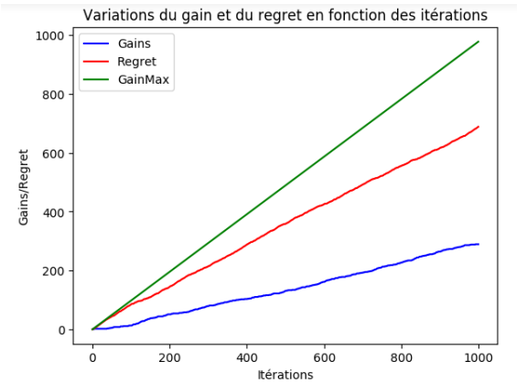

FIGURE 1.3 – $k = 5$

Observations

Pour $k = 200$: Nous pouvons voir que durant la phase d'exploration (les k premières itérations) l'algorithme se comporte de façon aléatoire où le gain et le regret sont égaux. Par la suite l'algorithme ayant acquit assez de connaissances sur les leviers, il prend à chaque fois le meilleur. c'est pour cette raison que la tangente du gain après la k^{me} itération est égale a la tangente du Gain Maximum et que le regret ne croit plus. Cette phase caractérise bien l'exploitation.

Pour $k = 5$: Sur cet exemple, le regret dépasse de loin le gain. Ceci s'explique par le fait que la phase d'exploration est très courte, donc une fois passé à l'exploitation , le joueur choisit un levier non optimal car il n'a pas pu tout exploré. ce qui conduit à de très mauvaises performances.

1.2.2 Conclusion

On peut donc en déduire que pour cette méthode, la phase d'exploration doit être suffisamment longue pour permettre l'acquisition des connaissances sur les leviers, car une fois terminée l'algorithme ne se basera que sur les informations déjà acquises. Il est efficace en cas d'exploration suffisante, mais pour des problèmes ou l'exploration est couteuse, ceci peut être un désavantage.

1.3 Algorithme ϵ-greedy

Cette stratégie commence d'abord par une première phase d'exploration optionnelle, puis a chaque itération, avec une probabilité ϵ on suit une stratégie aléatoire et avec une probabilité $1 - \epsilon$ on applique l'algorithme greedy.

1.3.1 Tests

Pour étudier les caractéristiques de cette stratégie, nous l'avons fait tourner 1000 fois sur une distribution uniforme de 50 leviers avec $k = 5$ et une fois $\epsilon = 0.1$ et une autre $\epsilon = 0.5$ pour constater les différences que cela peut induire.

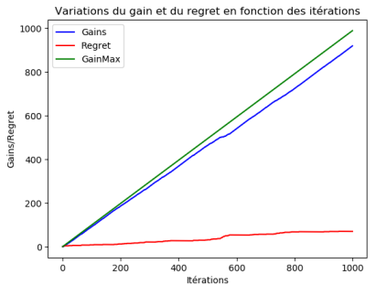

FIGURE 1.4 – $\epsilon = 0.1$

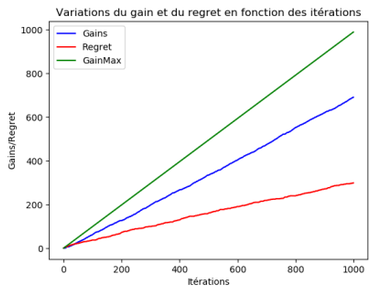

FIGURE 1.5 – $\epsilon = 0.5$

Observations

Pour$\epsilon = 0.1$ **:** pour ce test, la courbe du gain approche du gain Maximal, quant au regret il croit très lentement. ceci peut s'expliquer par le ϵ choisie. En effet pour une valeur de 0.1, l'algorithme va exploiter dans 90% des cas ce qui fait que l'action choisi est la meilleure au temps t et dès que t devient grand le levier sélectionné sera l'optimal

Pour $k = 5$ **:** Sur cet exemple, le gain ne s'approche pas autant du gain maximal . Ceci est due au fait qu'un ϵ de 0.5 oblige l'algorithme a explorer dans la moitié des cas même quand le t devient grand et qu'on a assez de connaissances sur les leviers, cette valeur handicap l'algorithme en le forçant a choisir une option non optimale.

1.3.2 Conclusion

On en conclut que pour cet algorithme que la valeur d'ϵ doit être choisi précautionneusement pour assurer que l'algorithme explore suffisamment mais qu'il ne soit pas pénalisé une fois avoir acquis assez d'informations.

1.4 Algorithme UCB

Pour cet algorithme, à un temps t,l'action choisie est $a_t = argmax_{i \in \{1,\dots,N\}}(\widehat{\mu}_t^i + \sqrt{\frac{2\log(t)}{N_t(i)}})$ le premier terme garantit l'exploitation alors que le deuxième devient important quand un levier est peu joué ce qui garantit l'exploitation.

1.4.1 Tests

Pour étudier les caractéristiques de cette stratégie, nous l'avons fait tourner 1000 fois sur une distribution uniforme de 50. Nous avons ensuite voulu experimenter sur l'equilibre exploration/exploitation en rajoutant un facteur multiplicatif au deuxieme terme qui assure l'exploration, nous avons testé pour un facteur de 0.05.

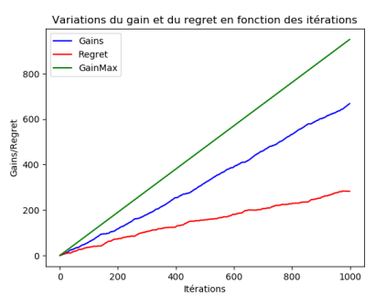

FIGURE 1.6 – UCB

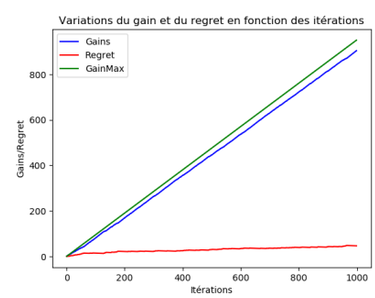

FIGURE 1.7 – UCB avec un facteur de 0.05

Observations

pour UCB : On observe que le gain suit le gain max mais ne lui est pas parallèle, le regret

aussi, ne se stabilise pas mais croit de façon constante. Ceci est due au fait que l'algorithme explore tout au long du processus, et donc il ne choisit pas toujours un levier car il est le meilleur mais parfois parce qu'il a été peu joué. l'algorithme est donc obligé de prendre de mauvais leviers même après avoir assez exploré.
pour UCB avec facteur de 0.05 : Pour cet exemple le gain approche du gain Maximum, le regret se stabilise dés le début et ne croit que de façon infime. Ceci est due au fait que l'algorithme n'est plus contraint d'explorer quand il a eu assez d'informations, donc après l'exploration il choisira toujours un levier optimal.

1.4.2 Conclusion

Cette méthode assure l'équilibre entre le besoin d'explorer et d'exploiter tout au long du processus.

1.5 Analyses Comparatives

1.5.1 Selon le nombre d'itérations

Pour cette comparaison nous avons fait tourner les 4 algorithmes sur une distribution uniforme de $N = 1000$ leviers et pour 10000 itérations. Le paramètre k de l'algorithme greedy a été mis à 1000(pour lui permettre d'explorer tout les leviers).Le paramètre ϵ de l'algorithme ϵ-greedy a été mis à 0.05 et nous avons choisi de prendre la version factorisé de UCB qui a de bien meilleures performances, le paramètre de factorisation a été mis à 0.05. Tout les paramètres ont été choisi de façon que tout les algorithmes aient les meilleurs performances et que nous puissions faire des comparaisons significatives.

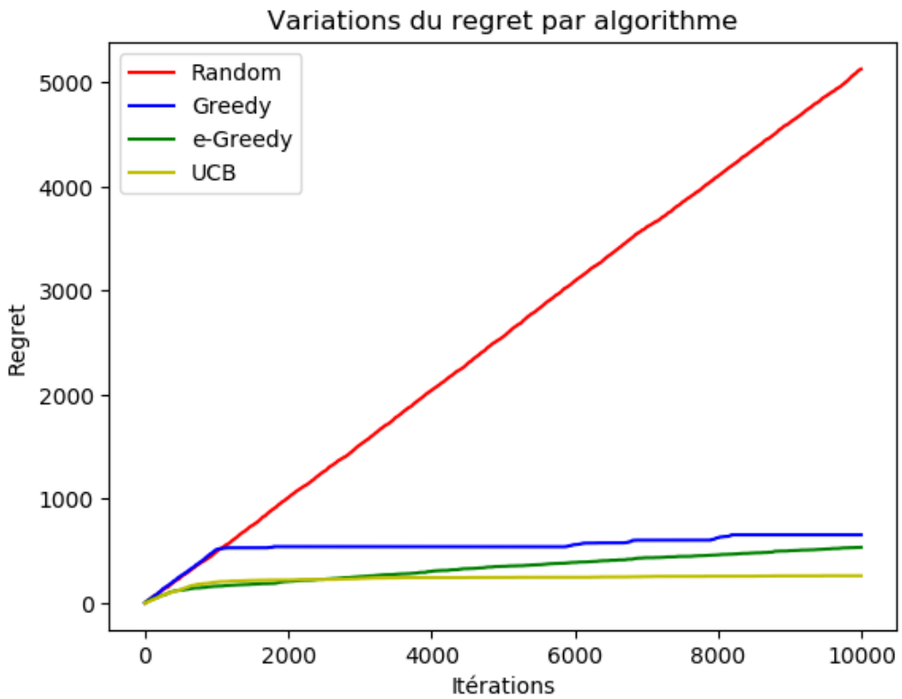

FIGURE 1.8 – Comparaison des quartes algorithmes

Observations
On remarque d'abord que l'algorithme aléatoire est le moins bon des quartes. Nous pouvons aussi observer la phase d'exploration de greedy puis la lente croissance du regret.Au tout début le regret de UCB est plus élevé que celui de ϵ-Greedy mais la courbe se stabilise et ne croit plus, tandis que pour ϵ-Greedy le regret croit lentement mais de façon constante ce qui amène les deux courbes a se croiser autour de la 2000^{me} itération. UCB a donc un regret moins élevé a la fin du processus.

Conclusion

On en conclut que l'algorithme UCB est le plus performant des quartes.

1.5.2 Selon le nombre de leviers

Pour cette comparaison nous avons fait tourner les 4 algorithmes sur 300 distributions uniformes de leviers de taille $1, ..., 300$ et pour 1000 itérations sur chaque distributions. Le paramètre k de l'algorithme greedy a été mis égale au nombre de leviers (pour lui permettre d'explorer tout les leviers).Le paramètre ϵ de l'algorithme ϵ-greedy a été mis à 0.05 et nous avons choisi de prendre la version factorisé de UCB qui a de bien meilleures performances, le paramètre de factorisation a été mis à 0.05. Tout les paramètres ont été choisi de façon a ce que tout les algorithmes aient les meilleurs performances et que nous puissions faire des comparaisons significatives. Nous avons également testé avec UCB sans facteur.

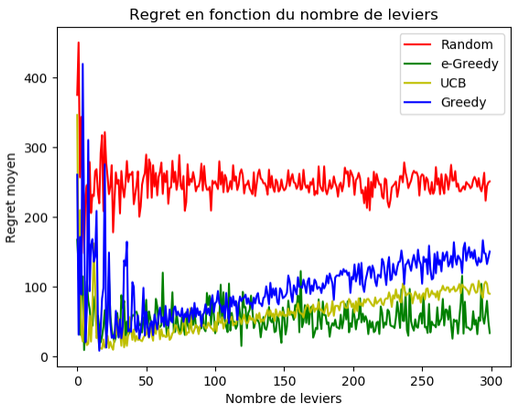

FIGURE 1.9 – UCB factorisé

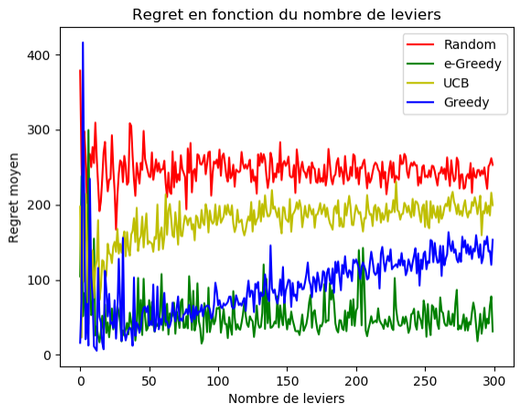

FIGURE 1.10 – UCB non factorisé

Observations et analyse

variation des leviers avec UCB factorisé :

On remarque d'abord que l'algorithme Greedy et UCB sont affectés. En effet le regret moyen pour les autres algorithmes fluctue autour des mêmes valeurs, mais pour Greedy et UCB elles croient. Ceci s'explique pour Greedy par le fait que le paramètre k est égale au nombre de leviers, il est nécessaire que sa valeur soit modifié car avec un nombre de leviers croissant, la phase d'exploration doit croitre aussi.Quant à UCB, la valeur du facteur multiplicatif ne change pas donc plus il y a de leviers plus l'algorithme va choisir d'explorer alors qu'il a assez de connaissances sur les leviers.

variation des leviers avec UCB non factorisé :

On remarque que UCB n'est pas affecté, il se rapproche de l'aléatoire au début puis se stabilise, seul Greedy est affecté de la même façon que l'exemple précédent.

Conclusion

On en conclut que les algorithmes UCB (avec facteur) et Greedy sont sensibles aux nombre de leviers.

1.5.3 Selon la distribution (poisson)

Pour cette comparaison nous avons fait tourner les 4 algorithmes sur une distribution de poisson de paramètre 1(les valeurs sont ensuite normalisé entre 0 et 1) de $N = 100$ leviers et pour 1000 itérations. Le paramètre k de l'algorithme greedy a été mis à 100(pour lui permettre d'explorer tout les leviers).Le paramètre ϵ de l'algorithme ϵ-greedy a été mis à 0.1 et nous avons choisi de prendre la version factorisé de UCB qui a de bien meilleurs performances, le paramètre de factorisation a été mis à 0.05. Tout les paramètres ont été choisit de façon a ce que tout les algorithmes aient les meilleures performances et que nous puissions faire des comparaisons significatives.

il y a donc beaucoup de leviers dont la probabilité de gagner est faible et de très rares leviers dont la probabilité est forte.

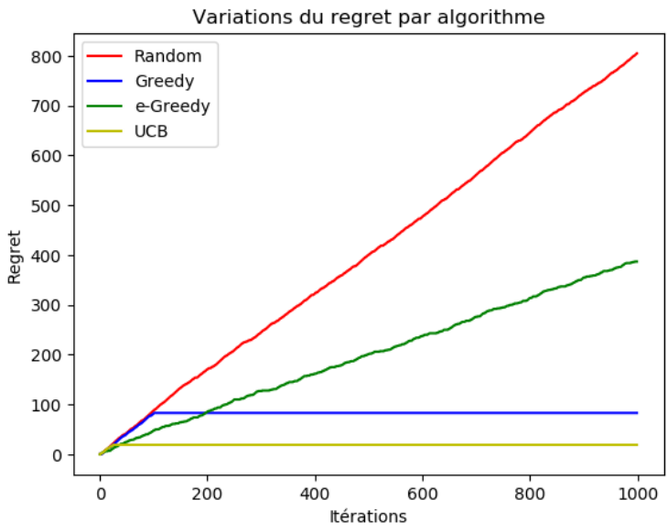

FIGURE 1.11 – Comparaison des quartes algorithmes

Observations et analyse

On remarque que UCB et Greedy ont des regrets stables, ceci est due pour Greedy au fait qu'il a explorer tout les leviers. Pour UCB, la décision d'explorer se base sur le levier ainsi que sur le nombre de fois ou il l'a essayé, il ne va donc pas explorer les leviers a faibles rendement. le regret d'ϵ-greedy croit et ceci et du au fait qu'il se doit d'explorer et vu que la plus part des leviers ont de faibles rendement, cela diminue sa performance.

Chapitre 2

Morpion et Monte-Carlo

Nous allons traiter dans cette partie le dilemme de l'exploration/exploitation à travers le jeu du morpion. 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.
Notre but est d'implémenter deux types de joueurs, un aléatoire et un joueur Monte-Carlo.

2.1 Joueur Aléatoire

Ce joueur décide de prendre aléatoirement une case du plateau parmi celles vides et y place son symbole.

2.1.1 Test : deux joueurs aléatoires

Pour ce Test, nous avons confronté deux joueurs aléatoires ,le joueur qui commence est toujours le même à chaque itération. Nous avons donc procédé à deux tests où on inversait les positions des joueurs mais pour ce cas précis de deux joueurs aléatoires, il n y a pas besoin de montrer les deux tracés.

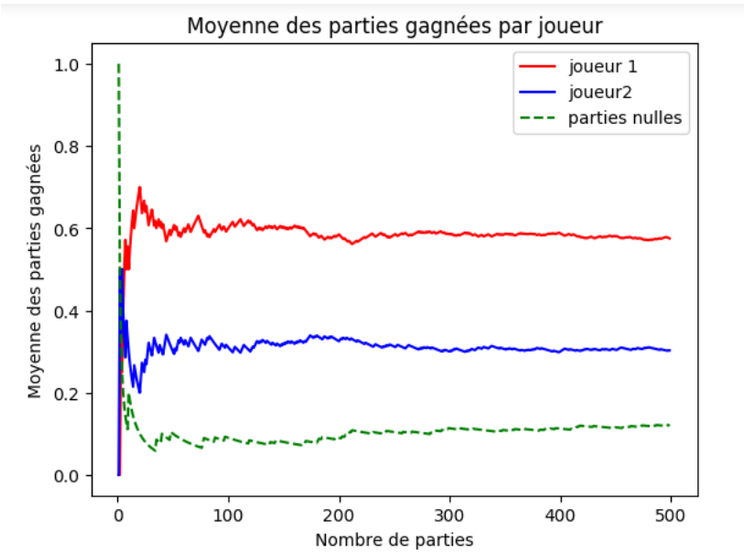

FIGURE 2.1 – deux joueurs aléatoires

analyse des résultats
Le joueur 1(qui commence) gagne dans 60% des cas, 30% pour le joueur 2 et les 10% restants sont des parties nulles. La variable aléatoire x qui dénote la victoire du premier joueur suit une loi de Bernoulli de paramètre $p = 0.58$ et de variance $var(x) = p(1-p) = 0.2436$.

2.2 Joueur MonteCarlo

L'algorithme de Monte-Carlo est un algorithme probabiliste qui vise à donner une approximation d'un résultat trop complexe à calculer : il s'agit d'échantillonner aléatoirement et de manière uniforme l'espace des possibilités et de rendre comme résultat la moyenne des expériences. Dans le cas d'un jeu tel que le morpion, à un état donné, plutôt que de calculer l'arbre de toutes les possibilités pour choisir le meilleur coup, il s'agit de jouer pour chaque action possible un certain nombre de parties au hasard et d'en moyenner le résultat. L'algorithme détaillé est le suivant :

algorithme MonteCarlo(state)
initialiser les récompenses des actions à 0
pour *s de 1 à N* **faire**
  choisir une action a au hasard **tant que** *la partie n'est pas finie* **faire**
    jouer les deux joueurs au hasard
  fin
  mettre à jour la récompense de l'action a en fonction du résultat
fin
retourner l'action avec la meilleure probabilité de victoire

2.2.1 Test Montecarlo vs aléatoires

Pour ce Test, nous avons confronté un joueur monte-carlo et un joueur aléatoire, nous allons représenter le cas ou monte-carlo commence et celui ou le joueur aléatoire commence.

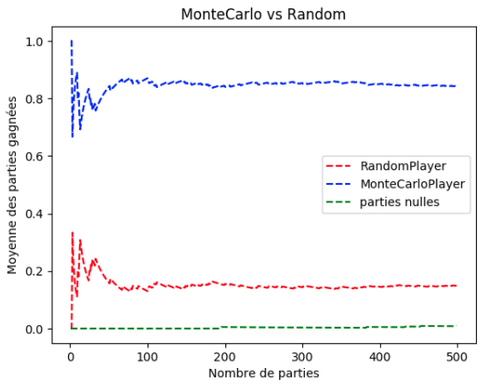

FIGURE 2.2 – Random joue en premier

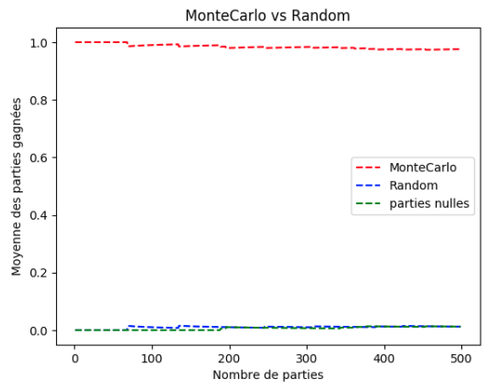

FIGURE 2.3 – Monte-Carlo commence

Analyse des résultats
Nous pouvons voir que quelle que soit le joueur qui commence, Montecarlo gagne la majorité du temps. Quand c'est lui qui commence ,il gagne dans plus de 90% des cas et dans 80% des cas lorsqu'il ne commence pas.C'est due au fait que le joueur qui commence est toujours avantagé dans le jeu du morpion.

2.2.2 monte-carlo vs monte-carlo

Pour ce Test, nous avons confronté deux joueurs MonteCarlo. Le joueur qui commence est toujours le même a chaque itérations.Nous avons donc procédé à deux tests ou on inversait les positions des joueurs mais pour ce cas précis de deux joueurs Montecarlo, il n y a pas besoin de montrer les deux tracés.

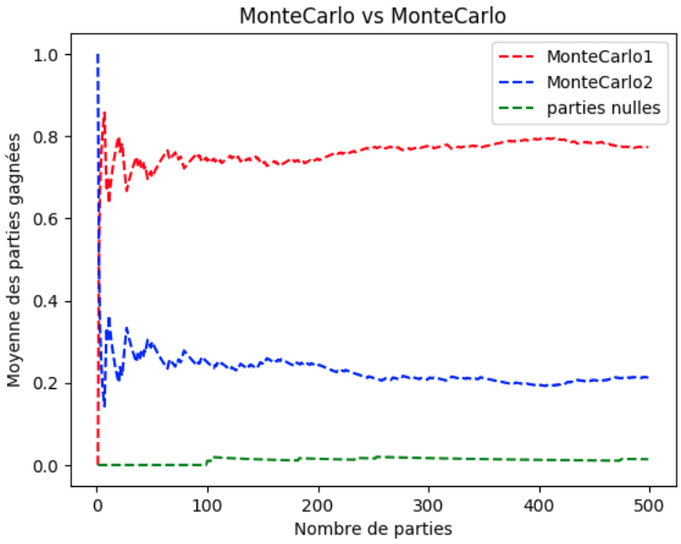

FIGURE 2.4 – deux joueurs Monte-Carlo

Analyse des résultats

Le joueur 1(qui commence) gagne dans 78% des cas, 20% joueur 2 et les 2% restants sont des parties nulles. Ceci s'explique par le fait que le joueur qui commence au morpion est toujours avantagé.

Chapitre 3

Arbre d'exploration et UCT

UCT désigne l'algorithme UCB adapté aux arbres de jeu (communément appelé également Monte Carlo Tree Search). Contrairement à l'algorithme précédent qui explore complètement et aléatoirement l'arbre des possibilités, l'idée de l'algorithme est d'explorer en priorité les actions qui ont le plus d'espoir d'amener à une victoire, tout en continuant à explorer d'autres solutions possibles.
Il consiste en 4 étapes qui sont itérés en fonction du nombre de simulation (ou temps) disponible :
1-selection : un nœud à explorer est sélectionné dans l'arbre en fonction de l'algorithme UCB, jusqu'à tomber soit sur une feuile de l'arbre (un nœud sans enfant) soit sur un état où une des actions n'a jamais été explorée.
2-Expansion : une fois le nœud sélectionné, pour une action jamais effectué à ce nœud, un nœud fils est créé et simulé.
3-Simulation : à partir d'un nœud à simuler, un jeu entre deux joueurs aléatoires est déroulé jusqu'à atteindre un état final (victoire, défaite ou match nul). Les états visités lors de la simulation ne sont pas ajoutés à l'arbre, ils ne sont pas stockés.
'-retro-propagation : On rétro-propage le résultat de la simulation à tout le chemin menant au fils simulé : pour chaque nœud sur le chemin, on met à jour en fonction du résultat le nombre de victoires et le nombre de visites.

3.0.1 Test UCT vs aléatoire

Pour ce Test, nous avons confronté un joueur UCT et un joueur aléatoire, nous allons représenter le cas ou UCT commence et celui ou le joueur aléatoire commence.Nous avons itéré les 4 étapes de l'algorithme UTC 20 fois

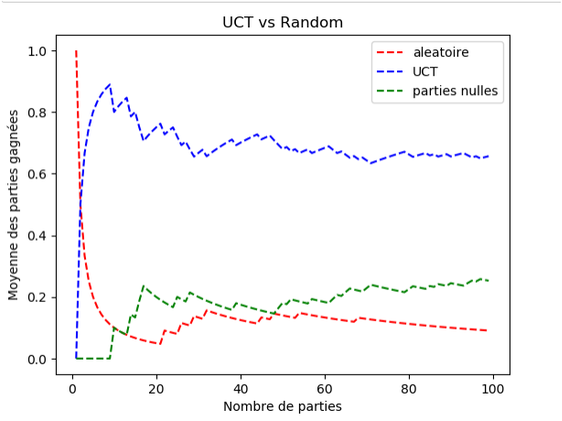

FIGURE 3.1 – Random joue en premier

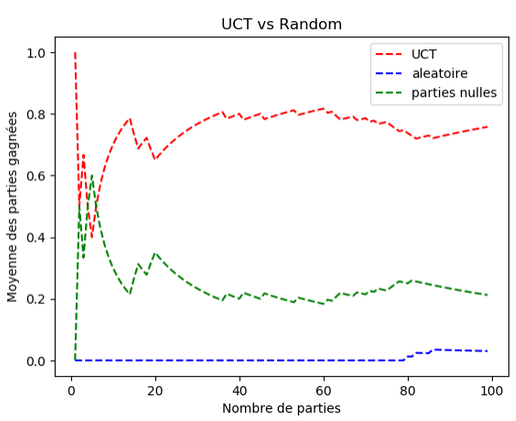

FIGURE 3.2 – UTC en premier

analyse des résultats

Comme on peut le voir UCT gagne le plus de fois qu'il commence en premier ou pas (70%-80% des parties jouées).

3.0.2 Test UCT vs Monte-Carlo

Pour ce Test, nous avons confronté un joueur UCT et un joueur Monte-carlo, nous allons représenter le cas ou UCT commence et celui ou le joueur Monte-carlo commence.Nous avons itéré les 4 étapes de l'algorithme UTC 45 fois, et nous avons fait 45 simulations pour monte-carlo. Nous avons pris ces valeurs pour assurer une équité entre les deux algorithmes et pouvoir les comparer.

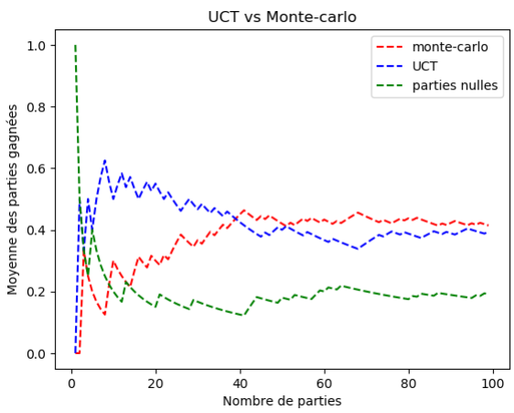

FIGURE 3.3 – monte-carlo en premier

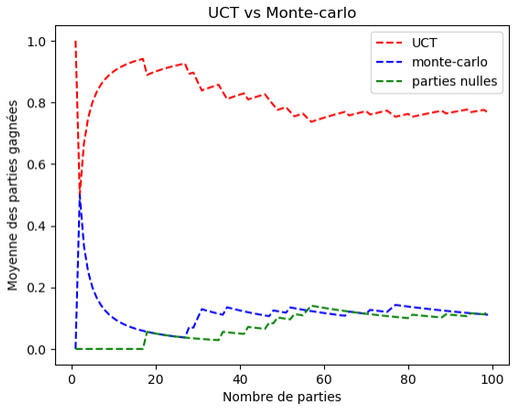

FIGURE 3.4 – UCT en premier

Analyse des résultats

Le cas ou monte-carlo est le premier à jouer : il gagne 40% des parties, UCT en gagne légèrement moins et 20% sont des matchs nuls. Au debut UCT gagne plus de parties puis les deux courbent convergent et se stabilisent a 40%.Pour les paramétrés choisis ,les deux algorithmes se valent car même si UCT choisi le prochain coup a simuler qui a le plus de chance d'amener a une victoire , monte-carlo a un avantage en commençant a jouer le premier

Dans le cas ou UCT est le premier a jouer : il gagne 80% des parties, Monte-Carlo en gagne 10%, le reste sont des matchs nuls.

3.0.3 Test UCT vs UCT

Pour ce Test, nous avons confronté deux joueurs UCT sur 500 parties. Le nombre de simulations de UCT est égale à 30 pour les deux joueurs.

Resultats

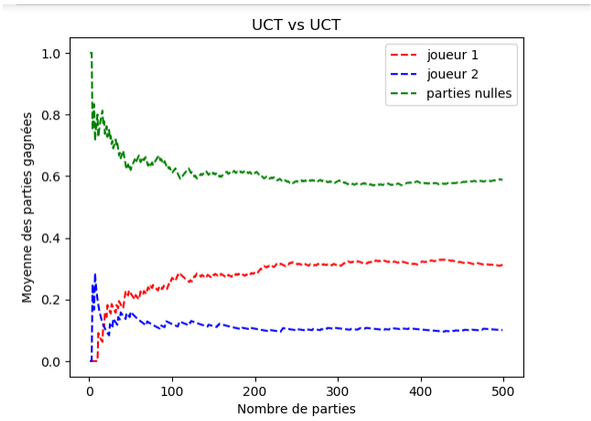

FIGURE 3.5 – deux joueurs UCT

analyse des résultats

On remarque que 60% des parties sont nulles, 30% sont gagnés par le joueur 1 et 10% par le joueur 2. le joueur 1 gagne plus que le 2 car il a commencé en premier donc il a un avantage.

3.0.4 Variation du comportement exploratoire

Dans cette partie nous allons étudiez le comportement exploratoire de UCT en ajoutant un facteur multiplicatif devant le deuxième terme.
Pour ce faire nous allons tester sur plusieurs cas de figure , un joueur UCT et un joueur Aléatoire (Aléatoire en premier) ainsi qu'un joueur UCT et un joueur Monte-carlo (UCT en premier) pour 10 valeurs du facteur de 0.1 à 0.9 et nous représentons la moyenne des parties gagnés sur 100 parties pour chaque valeur du facteur. Le nombre de simulations de UCT est égale à 20, et le nombre de simulations de Monte-carlo est mis à 18.
Résultats

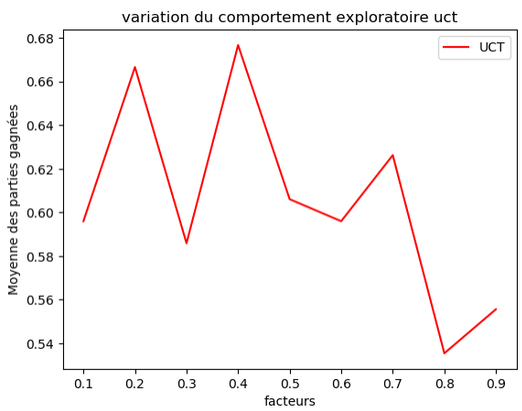

FIGURE 3.6 – UCT vs Aléatoire

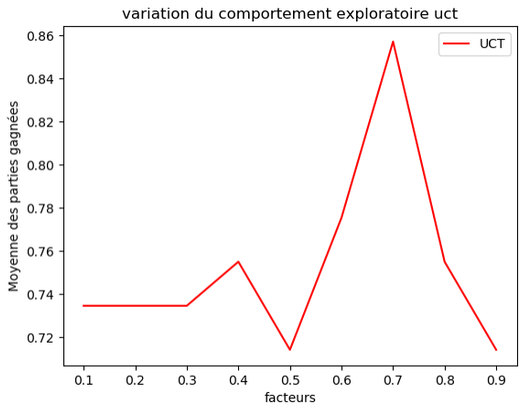

FIGURE 3.7 – UCT vs Monte-Carlo

Analyse des résultats

pour le premier test : on remarque pour le facteur $f = 0.4$, UCT fournit les meilleurs résultats contre aléatoire .
pour le deuxième test : on remarque pour le facteur $f = 0.7$, UCT fournit les meilleurs résultats contre Monte-carlo.

Ces valeurs permettent à l'algorithme de faire le meilleur choix des coups a explorer puis exploiter, pour d'autres valeurs, soit l'exploitation n'est pas assuré ,soit elle l'est trop et l'exploitation n'est pas assurée.

Conclusion générale

A travers les différentes stratégies de jeux étudiées durant ce projet. Nous avons réussi a caractérisé le dilemme de l'exploration/exploitation sous différents points de vus. Nous avons exploré des méthodes qui optent pour des phases d'explorations puis exploitations, d'autres qui explorent en continue suivant différents critères. Nous pourrions envisager plusieurs types d'autres approches à ce problème comme dans le cas où l'exploration aurait un cout ou que la rapidité était décisive. c'est un problème ardu et nécessitant plus d'études et d'expérimentations.