Rapport de projet

Exploration/Exploitation

Merrouche, Aymen Sidhoum, Imad

Année universitaire : 2018-2019

Table des matières

Chapitre 1

Introduction

En machine Learning le compromis "exploitation vs exploration" est une question capitale. Étant donnés un nombre de choix possibles, comment déterminer quel choix est le plus rentable ? est-il préférable d'utiliser notre situation actuelle et donc d'exploiter notre connaissance acquise pour en tirer le meilleur parti ? Ou vaut-il mieux continuer à explorer d'autres options pour récolter de nouvelles informations.L'exploitation consiste à déterminer le meilleur choix à partir de l'information déjà collectée.En revanche l'exploration consiste à collecter de nouvelles informations.Chacun étant aussi insatisfaisant que l'autre il faut concilier ces deux impératifs, mais le problème est de définir quelle part on attribue à chacun.

Dans un premier temps nous étudierons des algorithmes classiques du dilemme "exploitation vs exploration". Par la suite nous nous intéresserons à ce problème dans la résolution de jeux (morpion et puissance4) en utilisant différentes approches : aléatoire,Monte Carlo et UCT.

Chapitre 2

Bandits manchots

2.1 Description générale :

2.1.1 Principe du jeu :

Les bandits manchots (ou machine à sous) est un jeu de hasard, le principe est le suivant : à chaque partie le joueur à le droit d'actionner un levier qui fait tourner des rouleaux, et en fonction de la combinaison obtenue sur les rouleaux, une récompense est attribuée au joueur.

2.1.2 Notations :

On suppose que chaque machine possède N leviers dénotés par l'ensemble $\{1, .., N\}$. A chaque partie le joueur dois choisir un levier à joueur parmi les choix possibles.

- On note par α_t l'action du joueur à l'instant t, c'est-à-dire le levier qu'il a choisis d'actionner et donc il s'agit d'un entier entre 1 et N.
- La récompense associée à chaque levier i suit une distribution de Bernoulli de paramètre μ^i,et donc en actionnant le levier i le joueur obtient 1 avec une probabilité de μ^i et 0 avec une probabilité de $1 - \mu^i$.On note par r_t la récompense obtenue à l'instant t ,et donc r_t est une variable aléatoire qui suit une loi de Bernoulli de paramètre μ^{α_t}. On suppose aussi que les μ^i sont constants durant toute la partie.
- On note par G_T le gain du joueur au bout de T parties ($G_T = \sum_{t=1}^{T} r_t$),le but du joueur est donc de maximiser ce gain.Il est donc impératif de détecter le levier au rendement le plus élevé.
- On note par i^* le levier au rendement le plus élevé ($i^* = argmax_{i\in\{1,..,N\}}\mu^i$).
- On note par μ^* le rendement associé au meilleur levier, i.e. $\mu^* = \mu^{i^*} = max_{i\in\{1,..,N\}}\mu^i$, donc la stratégie optimale pour un joueur est d'actionner à chaque fois le levier i^*.
- On note par G_T^* le gain maximal au temps T, i.e. $G_T^* = \sum_{t=1}^{T} r_t^*$, avec r_t^* la récompense aléatoire tirée de la distribution de Bernoulli de paramètre μ^*.
- On note par L_T le regret au temps T qui est la différence entre le gain maximal espéré G_T^* et le gain du joueur G_T, i.e. $L_T = G_T^* - G_T = \sum_{t=1}^{T} r_t^* - \sum_{t=1}^{T} r_t = \sum_{t=1}^{T}(r_t^* - r_t)$, le but pour le joueur est donc de minimiser ce regret.
- On note par $N_T(a)$ le nombre de fois ou le levier a a été actionné au bout de T parties.
- On note par $\hat{\mu_T^a} = \frac{1}{N_T(a)} \cdot \sum_{t=1}^{T} r_t . \mathbf{1}_{a_t=a}$ la récompense moyenne estimée du levier a à partir des actions du joueur.

Remarques : Le but du joueur est donc de maximiser son gain G_T, pour ce faire il doit détecter le levier i^* au plus grand rendement μ^* pour s'approcher au plus du gain maximal espéré G_T^* (il ne peut pas faire mieux) et par conséquent il doit minimiser le regret $L_T = G_T^* - G_T$.Le joueur n'a bien évidemment pas connaissance des vrais paramètres des lois de Bernoulli μ^i de chaque levier i, et donc il doit en faire une estimation que nous avons noté précédemment $\hat{\mu_T^a}$ (pour chaque levier a),cette information est mise à jour et affinée après chaque partie, ça constitue l'information connue du joueur et c'est en ce basant sur cette information qu'il choisira quel levier actionner.
Il serait intéressant d'effectuer un certain nombre(E) de parties en explorations pour affiner, avant de joueur réellement, l'information connue i.e. la récompense moyenne estimée de chaque levier a au bout de E parties ($\hat{\mu_E^a}$).

— On note par E le nombre de parties effectuées en exploration.

2.2 Étude théorique des algorithmes :

2.2.1 L'algorithme aléatoire :

Principe :

Cet algorithme choisi l'action α_t à jouer à l'instant t aléatoirement parmi toutes les actions possible. C'est la *baseline* que tout les autres algorithmes devront dépasser.

Analyse :

Vu que l'algorithme tire aléatoirement une action à jouer parmi les actions possibles à chaque fois, il n'utilise donc pas l'information connue (i.e. la récompense moyenne estimée de chaque levier à partir des actions du joueur),il ne fait que de l'exploration. Ainsi le résultat sera le même avec ou sans exploration préalable.

2.2.2 L'algorithme greedy (glouton) :

Principe :

On effectue un certain nombre d'itération au préalable pour affiner l'information connue (i.e. la récompense moyenne estimée de chaque levier à partir des actions du joueur) et ce uniformément en utilisant l'algorithme aléatoire.Par la suie le choix se portera toujours sur l'action qui a la meilleur estimation $a_t = argmax_{i \in \{1,..,N\}} \hat{\mu^i}$.

Analyse :

Après la première phase exploratoire, l'algorithme n'effectue plus d'exploration (de choix aléatoire) il exploite uniquement l'information acquise pour en tirer le meilleur parti, il fait uniquement de l'exploitation.

Sans une première phase exploratoire le comportement de l'algorithme est totalement imprévisible il peut donner de très bons comme de très mauvais résultats.
par exemple : soit une machine qui possède deux leviers $real[l1 : 0.01, l2 : 0.99]$ (un levier avec lequel on gagne presque toujours et un autre avec lequel on perd presque toujours), on décide de jouer 10 parties sans phase exploratoire

l'information connues est : $known[l1:0, l2:0]$
cas favorable :
Supposant que lors de la première partie l'algorithme renvoie le deuxième levier, le joueur gagne 1 et donc on obtient $known[l1:0, l2:1]$
Durant toutes les autres parties l'algorithme va renvoyer le deuxième levier, et on aura à la fin gagner 10 (ou presque puisque il y a un risque de 0,01 de perdre avec ce levier)
cas défavorable :
Supposant que lors de la première partie l'algorithme renvoie le premier levier, et par chance le joueur gagne 1 et donc on obtient $known[l1:1, l2:0]$
Durant toutes les autres parties l'algorithme va renvoyer le premier levier, et on aura à la fin gagner 1 (ou presque puisque il y a une chance de 0,01 de gagner avec ce levier)
D'où la nécessité d'une première phase exploratoire car sinon l'algorithme renverra toujours le premier levier qui lui a parmi de gagner, et la mise à jour de l'information connue n'aura pas de sens puisqu'elle ne concernera qu'un seul levier.

Les performances de cet algorithme vont dépendre du nombre de parties de la phase exploratoire qui doit être en adéquation avec le nombre de leviers (plus le nombre de leviers est important plus le nombre de parties de la phase exploratoire doit être grand),pour un nombre convenable de parties en phase exploratoire on pourra constituer une bonne estimation des paramètres de lois Bernoulli que suivent les gains associés à chaque levier et donc on réussira à détecter le bon levier.

2.2.3 L'algorithme ϵ-greedy :

Principe :

La première phase exploratoire est optionnelle.A chaque itération l'algorithme va renvoyer :

- Avec une probabilité de ϵ une action aléatoirement parmi toutes les actions possibles (même résultat que l'algorithme aléatoire)
- Avec une probabilité de $1-\epsilon$ $a_t = argmax_{i\in\{1,..,N\}}\hat{\mu^i}$,(même résultat que l'algorithme *greedy*)

Analyse :

- Renvoyer avec une probabilité de ϵ une action aléatoirement parmi toutes les actions possibles garanti l'exploration.
- Renvoyer avec une probabilité de $1-\epsilon$ le levier $a_t = argmax_{i\in\{1,..,N\}}\hat{\mu^i}$ garanti l'exploitation.

Cet algorithme explore et exploite en même temps.
Le paramètre ϵ permettra de gérer la part qui est attribuée à l'exploration et celle attribuée à l'exploitation.
Contrairement à l'algorithme *greedy* la première phase exploratoire n'est pas nécessaire puisque on explore continuellement en utilisant l'algorithme aléatoire avec une probabilité de ϵ, mais elle influence beaucoup sur les performance car on se base avec une probabilité de $1-\epsilon$ sur l'algorithme *greedy*.

2.2.4 L'algorithme UCB (Upper Confidence Bound) :

Principe :

l'action qui est renvoyée est : $a_t = argmax_{i \in \{1,..,N\}}(\hat{\mu_t^i} + \sqrt{\frac{2log(t)}{N_t(i)}})$.

Analyse :

- Le premier terme $\hat{\mu_t^i}$ identique aux autres algorithmes garanti l'exploitation.
- Le deuxième terme $\sqrt{\frac{2log(t)}{N_t(i)}}$ (bonus de confiance) 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, et donc il attribue un bonus aux actions qui ont été peu jouées il garanti ainsi l'exploration.

Cet algorithme exploite et explore en même temps.

2.3 Expérimentation :

- déroulement d'une simulation :
  - On génère aléatoirement N valeurs entre 0 et 1 , N étant le nombre de leviers de la machine, ce sont les μ^i.
  - On initialise l'information connue à 0 (le nombre de fois ou chaque levier à été actionné et le gain associé à chaque levier) c'est les $\hat{\mu^i}$ et les $N_0(i)$
  - Pour chaque tentative t :
    - On récupère l'action α^t selon la stratégie.
    - On récupère le gain r_t associé à cette action qui suit une Bernoulli de paramètre μ^{α_t}
    - On met à jour l'information connue, $\hat{\mu^t}$ et les $N_t(i)$
    - On récupère le gain maximal r_t^* associé à l'action i^* qui suit une Bernoulli de paramètre μ^*
    - On met à jour le gain réel du joueur G_T et le gain idéal G_T^* ($G_T = G_T + r_t$ et $G_T^* = G_T^* + r_t^*$)
    - On récupère le regret L_T

2.3.1 Algorithme aléatoire :

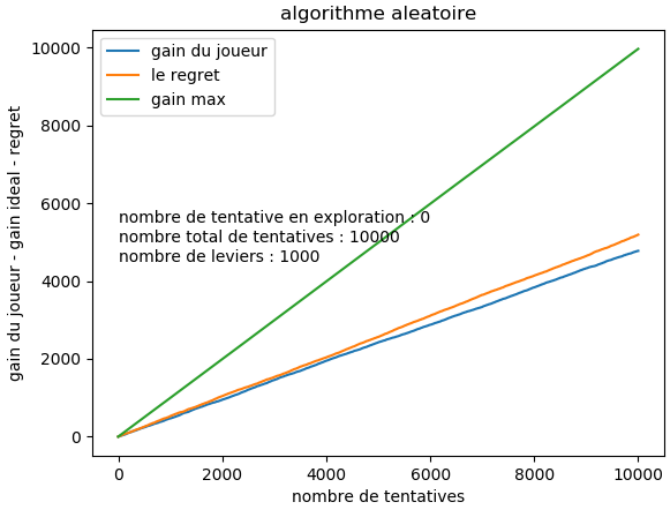


FIGURE 2.1 – Regret en fonction du nombre de tentatives - Aléatoire

Commentaires : Comportement correspondant avec la stratégie aléatoire, sachant que les paramètres des lois Bernoulli utilisées sont aléatoirement tirés entre 0 et 1.

2.3.2 Algorithme greedy :

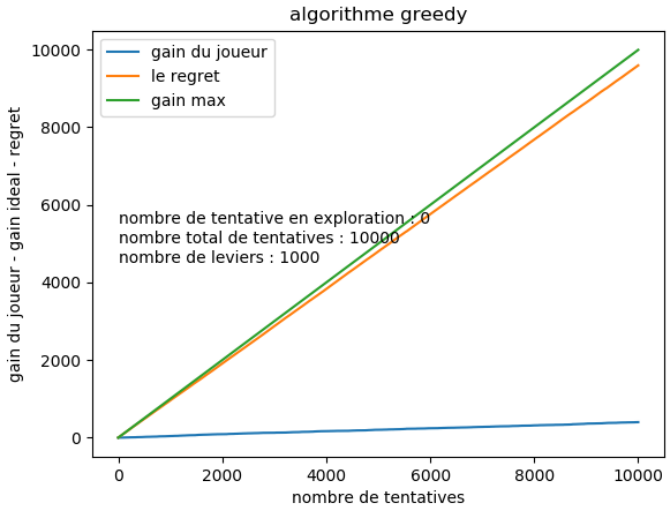


FIGURE 2.2 – Regret en fonction du nombre de tentatives - greedy - sans phase exploratoire

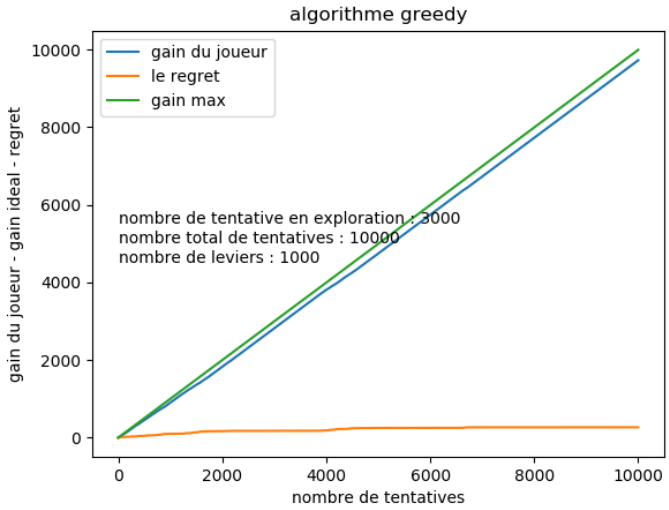


FIGURE 2.3 – Regret en fonction du nombre de tentatives - greedy - avec phase exploratoire

Commentaires : Sans phase exploratoire le comportement de *greedy* est imprévisible.Avec 2000 parties en exploration préalable uniforme et pour 1000 leviers l'algorithme *greedy* est capable de détecter un bon levier à jouer, le regret est minimisé et le gain du joueur s'approche du gain idéal.

2.3.3 Algorithme ϵ-greedy :

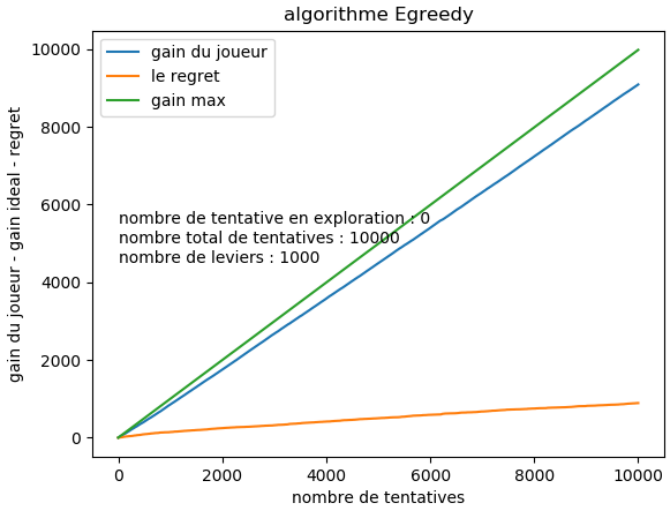


FIGURE 2.4 – Regret en fonction du nombre de tentatives - ϵ-greedy - $\epsilon = 0.1$

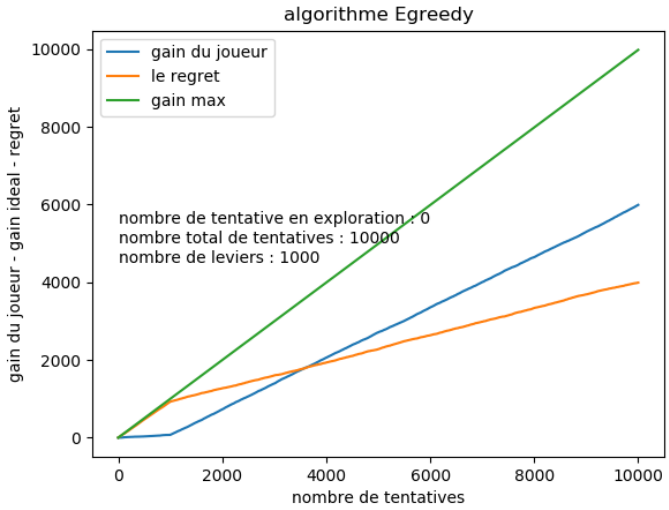


FIGURE 2.5 – Regret en fonction du nombre de tentatives - ϵ-greedy - $\epsilon = 0.001$

Commentaires : Même sans phase exploratoire préalable et pour 1000 leviers en utilisant l'algorithme ϵ-greedy, le regret est minimisé et le gain du joueur s'approche du gain idéal.ceci est du à l'exploration uniforme continue avec une probabilité de $\epsilon = 0.1$ qui va lui permettre au fur et à mesure de juger quels sont les meilleurs leviers à jouer.On peut diminuer la part d'exploration. Par exemple avec un $\epsilon = 0.001$ trop petit, l'algorithme n'arrive pas à explorer suffisamment.

2.3.4 Algorithme UCB :

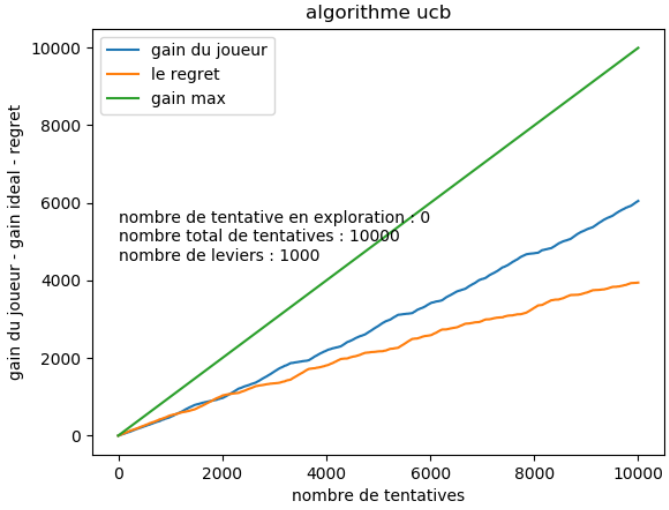


FIGURE 2.6 – Regret en fonction du nombre de tentatives - UCB - sans facteur

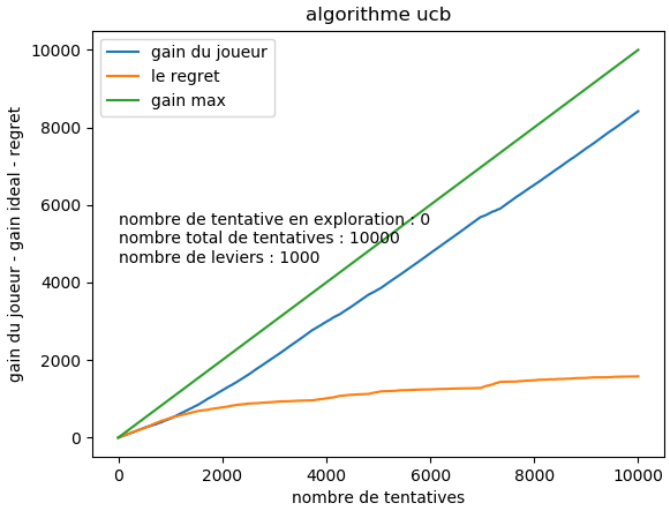


FIGURE 2.7 – Regret en fonction du nombre de tentatives - UCB - facteur=0.2

Commentaires : Même sans phase exploratoire préalable et pour 1000 leviers en utilisant l'algorithme UCB , le regret est minimisé et le gain du joueur s'approche du gain idéal.ceci est du à l'exploration uniforme continue en attribuant un bonus de confiance au leviers qui ont été peu joués.On peut moduler la part d'exploration et d'exploitation en rajoutant un facteur multiplicatif devant le facteur de confiance, avec un facteur de $\epsilon = 0.2$ les résultats sont améliorés.

$$a_t = argmax_{i \in \{1,..,N\}}(\hat{\mu_t^i} + 0.2\sqrt{\frac{2log(t)}{N_t(i)}})$$

2.3.5 Comparaison entre les différents algorithmes :

En fonction du nombre de parties :

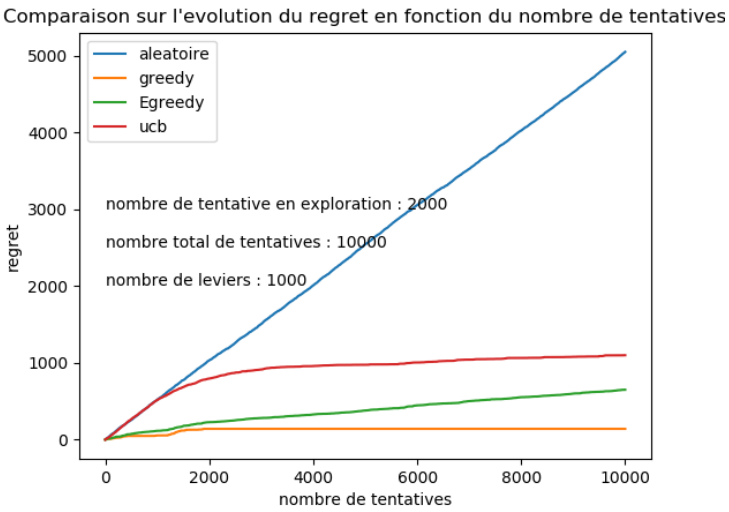


FIGURE 2.8 – Regret en fonction du nombre de tentatives - avec exploration préalable

- Pour $n = 1000$ leviers.
- Pour $n = 10000$ tentatives.
- Le paramètres des lois Bernoulli sont tirées aléatoirement entre 0 et 1.
- Avec $n = 2000$ tentatives en phase d'exploration uniforme préalable.
- Avec $\epsilon = 0.1$ pour greedy.
- Avec $facteur = 0.1$ pour UCB.

Commentaires : Le regret augmente pour toutes les stratégies c'est logique puisque les gains sont cumulées après chaque partie et donc les grandeurs augmentent.Avec 2000 tentatives en phase exploratoire l'algorithme greedy est le meilleur puisque il aura détecter le bon levier.La stratégie ϵ-greedy est aussi efficace étant donne qu'elle se base sur la stratégie greedy avec une probabiliste de 0.9.

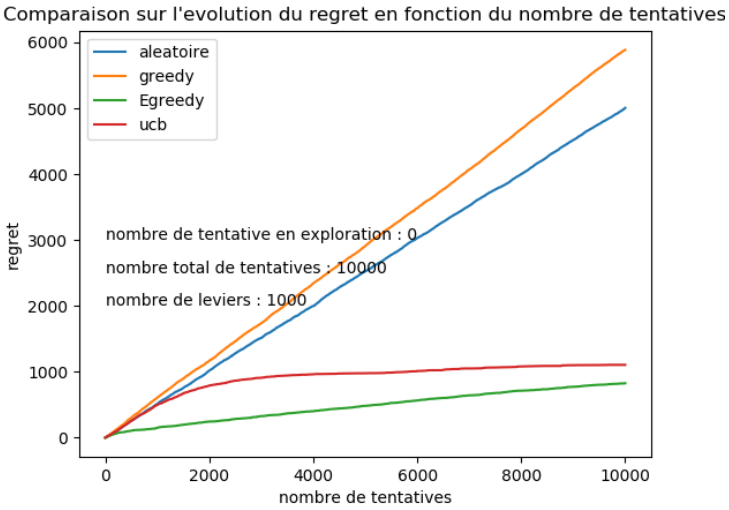


FIGURE 2.9 – Regret en fonction du nombre de tentatives - sans exploration préalable

- Pour $n = 1000$ leviers.
- Pour $n = 10000$ tentatives.
- Le paramètres des lois Bernoulli sont tirées aléatoirement entre 0 et 1.
- Avec $n = 0$ tentatives en phase d'exploration uniforme préalable.
- Avec $\epsilon = 0.1$ pour greedy.
- Avec $facteur = 0.1$ pour UCB.

Commentaires : Le regret augmente pour toutes les stratégies c'est logique puisque les gains sont cumulées après chaque partie et donc les grandeurs augmentent.Sans exploration l'algorithme greedy est imprévisible il fait même pire que l'aléatoire.Grace à l'exploration continue de ϵ-greedy et UCB on arrive au fur et à mesure à se faire une idée sur les bons leviers, le regret est minimisé.

En fonction du nombre de leviers :

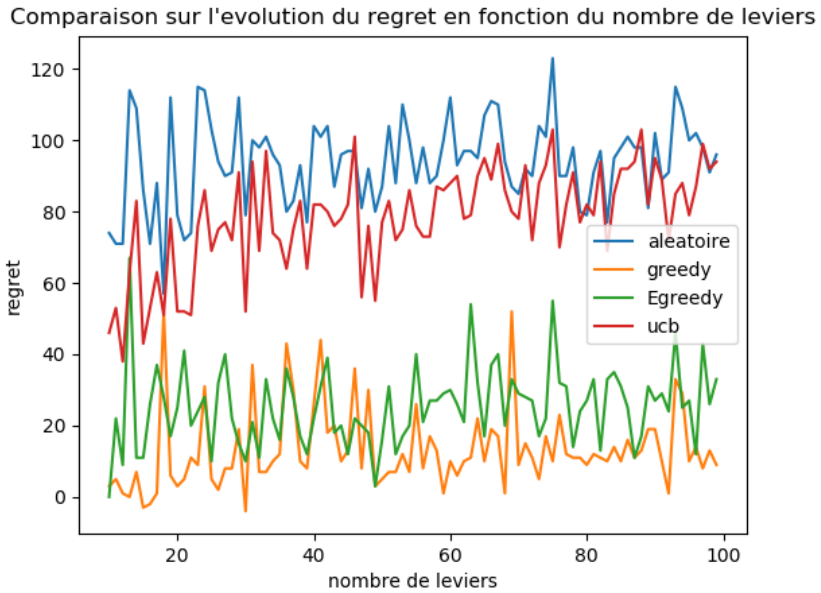


FIGURE 2.10 – Regret en fonction du nombre de leviers - avec exploration préalable

Commentaires L'algorithme aléatoire n'est pas affecté par le nombre de leviers puisque il tire aléatoirement un leviers quelque soit leurs nombre.Le regret augmente pour les autres, l'algorithme UCB étant le plus sensible.

Chapitre 3

Morpion et Monte Carlo :

3.0.1 Principe du jeu :

Le morpion est un jeu qui se joue à deux sur un damier de 3 cases par 3 cases, chaque joueur à son tour inscrit son symbole sur une case vide, le but du jeu est de réussir à aligner ses trois symboles horizontalement, verticalement ou en diagonal, on remporte alors la partie.

3.1 Étude théorique des algorithmes :

3.1.1 Joueur Aléatoire

Principe :

Ce joueur joue une case choisie aléatoirement parmi toutes les cases possibles (une case libre est une case possible).

Analyse :

Supposons que deux joueurs aléatoires j_1 et j_2 jouent un certain nombre de parties N.

— On note par g_i la variable aléatoire qui dénote la victoire de j_1 à la partie i.

$$i = \begin{cases} 1 & \text{si} \quad j_1 \text{ gagne} \\ 0 & \text{si} \quad j_1 \text{ perd ou fait match nul} \end{cases}$$

— Supposons que $P(g_i = 1) = p$ et que $P(g_i = 0) = 1 - p$ (on estimera cette valeur à partir de l'expérimentation).

On en conclut que g_i suit une loi de Bernoulli de paramètre p.

$$g_i \longrightarrow B(p)$$

$$E(g_i) = p \text{ et } Var(g_i) = p(1-p)$$

Notons G la variable aléatoire $G = \sum_{i=1}^{N} g_i$,les g_i sont indépendantes et identiquement distribuées (car les deux joueurs sont aléatoires), la variable aléatoire G suit une loi Binomial de paramètres (N, p).

$$G \longrightarrow B(N, p)$$

$$E(G) = Np \text{ et } Var(G) = Np(1-p)$$

3.1.2 Joueur Monte Carlo

Principe :

Cette algorithme consiste à effectuer un certain nombre de simulation N pour chaque action possible en utilisant deux joueurs au hasard, enfin l'action avec la moyenne de victoire la plus haute est renvoyée.

Analyse :

Pour un nombre d'itération N suffisamment grand, la stratégie Monte Carlo permet d'explorer tout les cas possibles. Dans ce cas un joueur Monte Carlo ne peut pas perdre, il gagne toujours s'il commence et il gagne ou fait un match nul si son adversaire commence.

3.2 Expérimentation :

3.2.1 Joueur Aléatoire

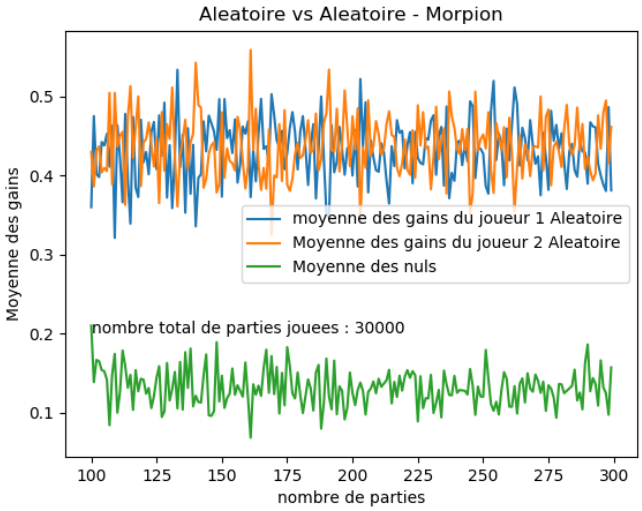


FIGURE 3.1 – Gain en fonction du nombre de parties - Aléatoire vs Aléatoire

• Pour la variable aléatoire g_i qui dénote la victoire de j_1 : Pour $N = 1000$ parties jouées aléatoirement on trouve que :

— $E(g_i) = 0.41434262948207173$ (en utilisant l'estimateur de l'espérance $E(X) = \frac{1}{1000}X_i$)

— $var(g_i) = 0.2426628148759541$ (en utilisant l'estimateur de la variance $var(x) = \frac{1}{1000}\sum_{t=1}^{n}(X_i - \overline{X})^2$)

vérifions avec les résultats théoriques :

$$p = E(g_i) = 0.41434262948207173$$

$$1 - p = 0.58565738$$

on aura

$$p(1-p) = 0.2426$$

ce qui correspond exactement à la variance retrouvé par l'expérience.

- Pour la variable aléatoire $G = \sum_{i=1}^{N} g_i$ dénote la somme des victoires de j_1 au bout de N parties : Pour 200 itérations et à chaque itération $N = 200$ parties jouées aléatoirement on trouve que :
  - $E(G) = 87.62$ (en utilisant l'estimateur de l'espérance $E(X) = \frac{1}{200}X_i$)
  - $var(g_i) = 53.70560000000004$ (en utilisant l'estimateur de la variance $var(X) = \frac{1}{200}\sum_{t=1}^{n}(X_i - \overline{X})^2$)

vérifions avec les résultats théoriques :

$$Np = 200 * 0.41434262948 = 82.86$$

ce qui correspond à l'espérance retrouvée par l'expérience

$$Np(1-p) = 48.5332$$

ce qui correspond à la variance retrouvé par l'expérience

- Cela veut dire que pour 200 parties un joueur aléatoire gagne en moyenne 87 partie, (42%). Le nombre de gains peut s'écarter de 53 parties par rapport à 87 parties, ce qui veut dire que c'est très instable.

3.2.2 Joueur Monte Carlo

- Détails de l'algorithme :
  - A partir de l'état courant on récupère le joueur courant pour savoir plus tard si cet état est favorable ou pas
  - On initialise les récompenses des actions possibles à 0.
  - Pour un certain nombre d'itérations :
    - On récupère aléatoirement une action parmi toutes les actions possibles.
    - On simule cette action en utilisant deux joueurs aléatoires.
    - On met à jour le tableau des gains, +1 si le joueur courant a gagné.
  - L'action avec le nombre de gains le plus élevé est renvoyée.
  - Dans toutes les simulations le joueur qui commence est tiré aléatoirement à chaque nouvelle partie.

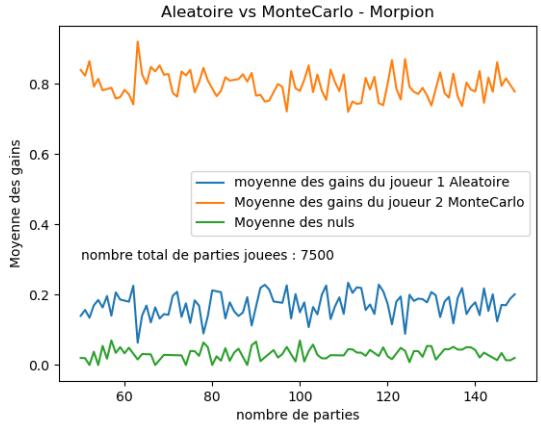


FIGURE 3.2 – Gain en fonction du nombre de parties - Aléatoire vs Monte Carlo

- Il est clair que la stratégie Monte Carlo est meilleure que l'aléatoire, le joueur aléatoire perd presque toujours.

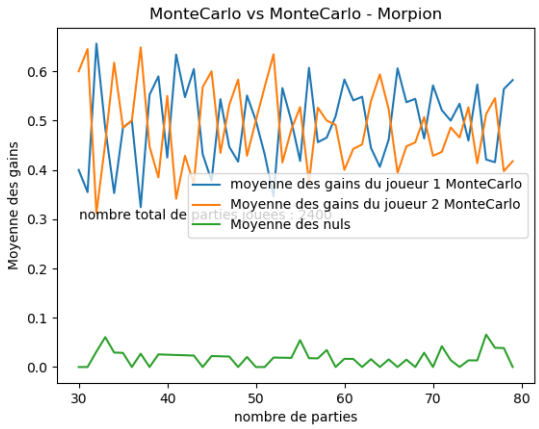


FIGURE 3.3 – Gain en fonction du nombre de parties - Monte Carlo vs Monte Carlo

- Puisque les deux joueurs utilisent tout les deux la stratégie Monte Carlo ils ont la même intelligence et donc la même espérance de gain (0.5).

Chapitre 4

Arbre d'exploration et UCT (Monte Carlo Tree Search) :

4.1 Principe :

Contrairement à l'algorithme Monte Carlo qui lui choisi aléatoirement les coups à simuler, l'algorithme UCT analyse en priorité les coups les plus prometteurs tout en continuant à explorer d'autres actions éventuelles ce qui nous renvoie au dilemme exploitation vs exploration, en effet est-il préférable de simuler un coup qui à déjà un bon score pour s'assurer qu'il est bon a jouer ? ou vaut-il mieux explorer d'autres actions qui peuvent être meilleurs que celles déjà explorées, la part de l'exploration et de l'exploitation est gérée par l'algorithme UCB qu'on a vu précédemment.Cet algorithme se base sur un arbre d'exploration chaque nœud de l'arbre correspond a un état du jeu, la racine correspond a l'état initial,et le fils d'un nœud sont les états qu'on peut engendrer a partir de l'état de ce nœud en jouant une action seulement.Les étapes de cet algorithme sont les suivantes :

Initialisation : A fin d'initialiser les nœuds fils de la racine on va simuler chaque action un fois en utilisant deux joueurs aléatoires, on stocke dans chaque nœud le nombre de fois ou il a été simulé et le nombre de fois ou cette simulation a mené a une victoire. A partir d'un arbre en partie exploré on répète le processus suivant un certain nombre de fois :

Sélection : Il s'agit de sélectionner un nœud a simuler, pour cela on va parcourir l'arbre a partir de la racine et tant qu'on a pas trouver un nœud feuille ou un nœud auquel on peut rajouter des fils (un nœud dont le nombre de fils est inférieur strictement au nombre de case vides de l'état de ce nœud, puisque chaque case vide engendre un état), on utilise l'algorithme UCB pour choisir le prochain nœud parmi les fils du nœud courant.

Expansion : On va créer un nouveau fils au nœud sélectionné dans l'étape prétendante, l'état correspondant à ce nouveau nœud est l'état du nœud sélectionné après avoir jouer une action jamais effectuée.

Simulation : On simule le nouveau nœud crée en utilisant deux joueurs aléatoires.

Rétro-propagation de résultats : Selon le résultat de la simulation du nouveau nœud, on doit le rétro-propager(faire remonter ce résultat jusqu'à la racine de l'arbre, et donc mettre à jour le nombre de simulation et le nombre de gain de chaque nœud sur le chemin entre la racine de l'arbre et le nouveau nœud). Il est indispensable dans cette étape de se rappeler que le morpion se joue à deux, si on prend un nœud et son fils, les états correspondants à ces deux nœuds ont été générés par deux joueurs différents. Donc si le résultat de la simulation d'une feuille de l'arbre abouti à une victoire ceci est considéré comme défaite pour le nœud père de cette feuille.

- A la fin de ce processus le fils direct de la racine qui a le meilleur tau de victoires est renvoyé.

Remarques :

— On peut remarquer la similitude entre UCB et Monte Carlo, puisque à la fin de UCB on aura simuler plusieurs fois les actions possibles à partir de l'état initial (les fils directs de la racine) en rétro-propageant les résultats, la différence entre les deux algorithmes réside dans le choix des actions à simuler.

— On peut gérer la part de l'exploration et de l'exploitation en rajoutant un paramètre ϵ devant le bonus de confiance dans l'algorithme UCB.

$$a_t = argmax_{i \in \{1,..,N\}}(\hat{\mu_t^i} + \epsilon.\sqrt{\frac{2log(t)}{N_t(i)}})$$

4.2 Expérimentations :

- Détails de l'implémentation :

On peut utiliser une classe pour modéliser un nœud, on y stockera :

- L'état correspondant.
- Le coup qui à engendré cet état.
- Le nombre de fois ou ce nœud a été simulé.
- Le nombre de fois ou la simulation a mener à une victoire.
- Un tableau qui pointe ses fils.
- Un tableau qui contient la route complète de la racine jusqu'à lui.
- Une méthode pour propager une victoire dans l'arbre et une autre pour propager une défaite.

— Dans toutes les simulations le joueur qui commence est tiré aléatoirement à chaque nouvelle partie.

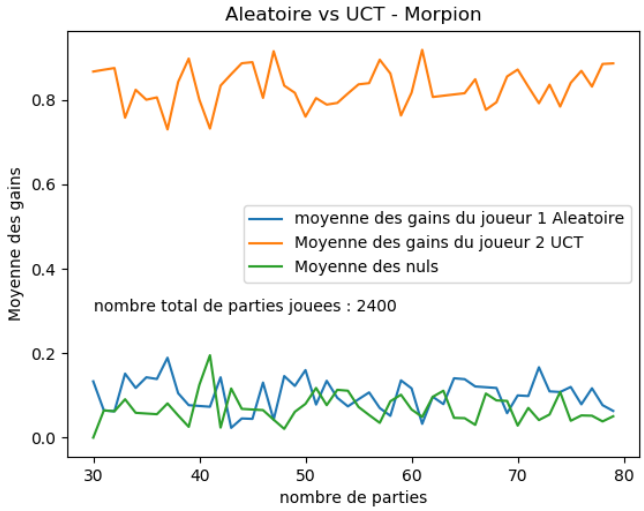


FIGURE 4.1 – Gain en fonction du nombre de parties - UCT vs Aléatoire - sans facteur multiplicatif

- Il est clair que la stratégie UCT est meilleure que l'aléatoire, le joueur aléatoire perd presque toujours.

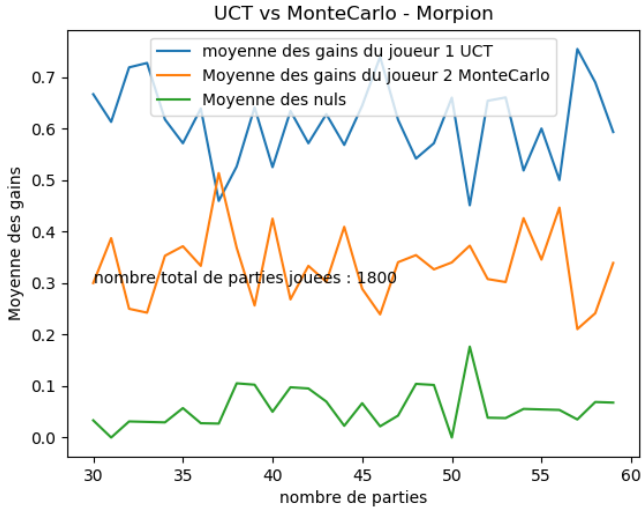


FIGURE 4.2 – Gain en fonction du nombre de parties - UCT vs Monte Carlo

- Les performances d'un joueur UCT contre un joueur Monte Carlo vont dépendre du nombre de fois qu'on ré-exécute le processus de UCT et du nombre de fois ou chaque action est simulée dans Monte Carlo, pour mettre en exergue le fait que UCT sélectionne les coup à simuler il faut initialiser ces deux parcomètres de façon à ce que le nombre de simulation de coups est le même, et il ne faut pas que le nombre d'itérations de Monte Carlo soit trop grand sinon il explorera

toute les solutions. Pour un nombre d'itérations cohérent entre les deux, l'algorithme UCT est meilleur.

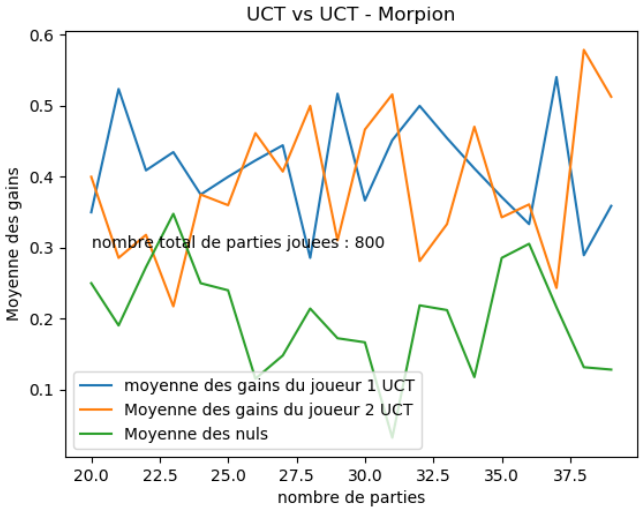


FIGURE 4.3 – Gain en fonction du nombre de parties - UCT vs UCT

- Puisque les deux joueurs utilisent tout les deux la stratégie UCT ils ont la même intelligence et donc la même espérance de gain.

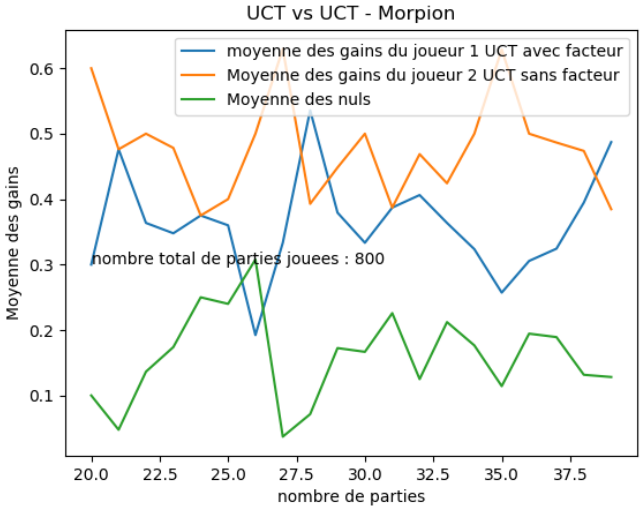


FIGURE 4.4 – Gain en fonction du nombre de parties - UCT vs UCT - avec facteur multiplicatif

- En rajoutant un facteur multiplicatif devant le bonus de confiance pour l'algorithme UCB on peut moduler l'exploration et l'exploitation. Avec un $facteur = 0.1$ un jouer UCT est moins bon qu'un autre UCT sans facteur.

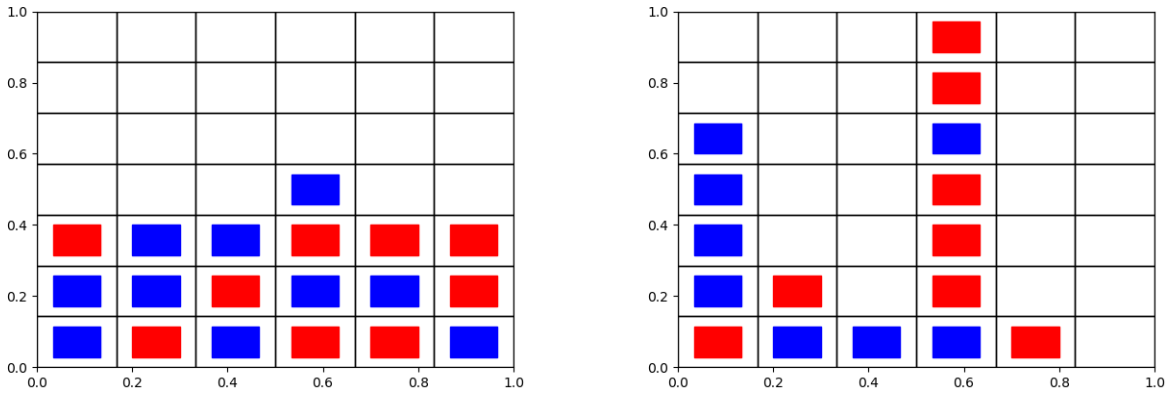
Chapitre 5

Puissance 4 :

5.1 Principe du jeu :

Le but du jeu est d'aligner une suite de 4 pions de même couleur sur une grille $6X7$. Chaque joueur dispose de 21 pions d'une couleur. Tour à tour, les deux joueurs placent un pion dans la colonne de leur choix, le pion coulisse alors jusqu'à la position la plus basse possible.Le vainqueur est le joueur qui réalise le premier un alignement (horizontal, vertical ou diagonal) consécutif d'au moins quatre pions de sa couleur. Si, alors que toutes les cases de la grille de jeu sont remplies, aucun des deux joueurs n'a réalisé un tel alignement, la partie est déclarée nulle.

Par exemple dans les deux parties suivante le joueur bleu gagne car il a connecté 4 pions bleus en diagonal durant la premier partie et 4 pions bleus en vertical durant la deuxième.



(a) 4 pions en diagonal

(b) 4 pions en vertical

FIGURE 5.1 – Exemple d'une partie de puissance 4

5.2 Expérimentations :

L'application des algorithmes : joueur aléatoire,joueur monte Carlo et joueur UCT dans la résolution du jeu puissance4 donne exactement les mêmes résultats que pour le jeu morpion.

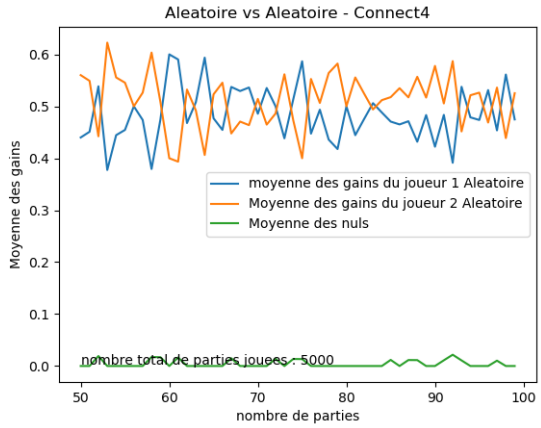


FIGURE 5.2 – Gain en fonction du nombre de parties - Aléatoire vs Aléatoire

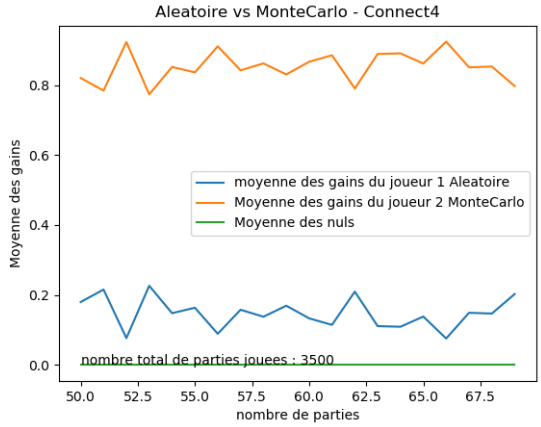


FIGURE 5.3 – Gain en fonction du nombre de parties - Aléatoire vs Monte Carlo

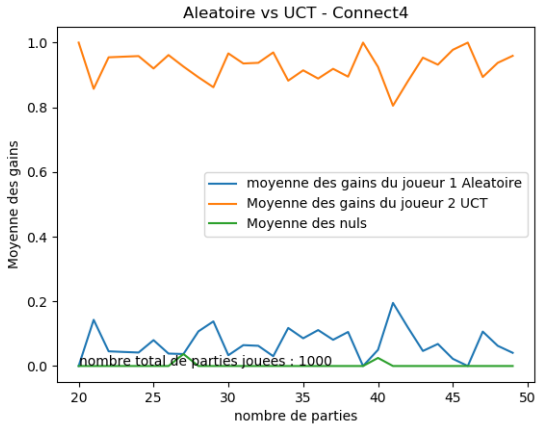


FIGURE 5.4 – Gain en fonction du nombre de parties - UCT vs Aléatoire

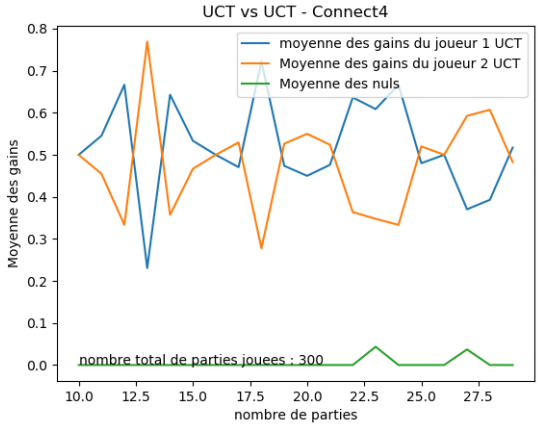


FIGURE 5.5 – Gain en fonction du nombre de parties - UCT vs UCT

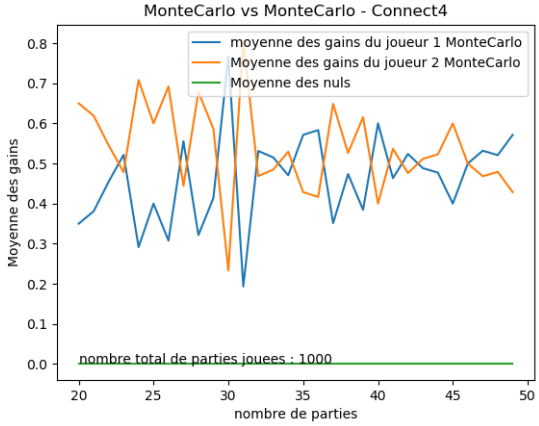


FIGURE 5.6 – Gain en fonction du nombre de parties - Monte Crlo vs Monte Carlo

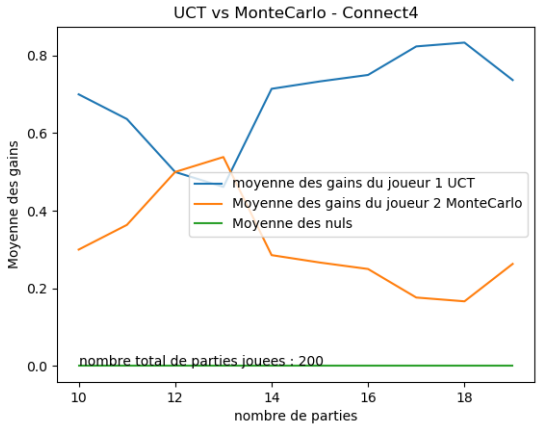


FIGURE 5.7 – Gain en fonction du nombre de parties - Monte Crlo vs UCT

Chapitre 6

Conclusion :

Se baser sur les connaissances acquises pour effectuer un choix, ou chercher à compléter la base de connaissance en explorant des possibilités non encore vues ? c'est à cette question que nous nous somme intéressé. Au terme de ce projet nous avons pu étudier des algorithmes classiques qui traitent du dilemme "Exploration vs Exploitation" .Nous nous sommes intéressé à ce problème dans la résolution de jeux à travers l'examen de différentes stratégies. Grâce aux outils statistiques et informatiques à notre portée, nous avons été capables de réaliser une étude théorique de ces algorithmes dans différentes situations et selon différents critères, et de vérifier par la suite ces résultats théoriques en implémentant ces algorithmes.

Table des figures