RAPPORT
PROJET N°1

Puissance 4
Dilemme Exploration/Exploitation

Marwan Tragha 28615820
Mehdi Chikh 3677161

Introduction :

L'informatique décisionnelle est une branche de l'informatique dont le but est d'aider à la prise de choix. Soit un problème donné, on cherche à trouver la meilleure solution possible.
Les jeux sont un bon moyen de représenter ces problèmes.

Dans le cadre de ce projet, le jeu en question est le puissance 4.
Le but est simple : Dans un tableau vertical de 6x7 cases, deux joueurs s'affrontent en plaçant un à un leurs pions. Le premier joueur arrivant à aligner 4 de ses pions gagne.

Ici le problème est donc le suivant :
- Comment choisir le meilleur coup à jouer pour arriver à la victoire ?
Ce problème est connu en théorie des probabilités sous le nom de « Problème du bandit manchot ».

Nous allons donc utiliser les solutions connues à ce problème pour élaborer un bon algorithme de prise de décision pour le puissance 4.

Plan :

1 – Algorithme aléatoire :
Dans cette partie nous utiliserons un algorithme aléatoire pour jouer au puissance 4. Nous étudierons ses performances en termes de prise de décision.
Cet algorithme nous servira pour la suite de baseline.

2 – Le problème du bandit manchot et algorithmes associés :
Dans cette partie nous présenterons différents algorithmes permettant de répondre au problème du bandit manchot.

3 – Algorithme Monte-Carlo :
Dans cette partie nous utiliserons un algorithme de Monte-Carlo pour jouer au puissance 4. Nous étudierons ses performances en termes de prise de décision comparé à l'algorithme aléatoire.

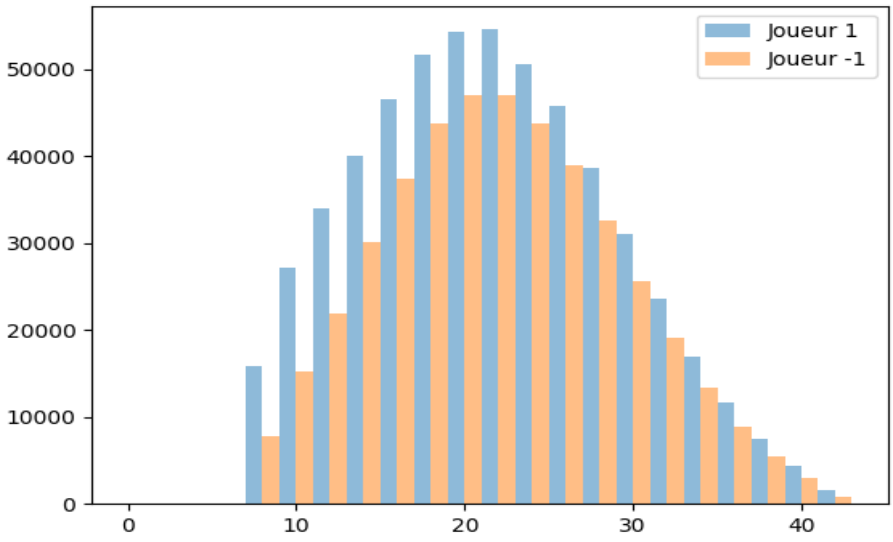
4 – Algorithme UCT :
Dans cette partie nous utiliserons un algorithme UCT pour jouer au puissance 4. Nous étudierons ses performances en termes de prise de décision comparé aux algorithmes précédents.

1 – Algorithme aléatoire :

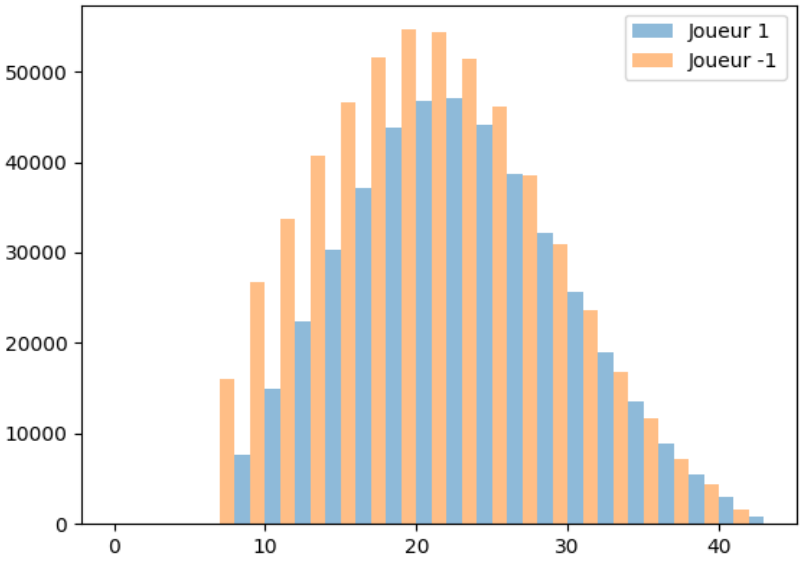
Nous avons implémenté le moteur du jeu.
Chaque algorithme sera représenté par un joueur qui l'utilisera pour jouer ses coups.
Ici l'algorithme est aléatoire, le joueur associé placera ses pions dans une colonne disponible au hasard, sans prise en compte de l'état actuel du jeu.

1 – Étude de la distribution :

Nous avons simulé 1 000 000 parties opposant deux joueurs aléatoires.
Pour étudier la distribution du nombre de coups avant une victoire, nous avons enregistré cette donnée dans un fichier à chaque partie et tracé l'histogramme correspondant :



Le joueur 1 commence.



Le joueur 2 commence.

En abscisse : Le nombre de coups avant une victoire.
En ordonnée : Le nombre de victoire associé.

On a un nombre de coup minimum de 7, c'est bien le nombre de coup minimum pour gagner au puissance 4 (4 coups du joueur 1 + 3 coups du joueur 2). Le nombre de coups maximal est bien 42, lorsque le deuxième joueur joue son dernier pion.

On remarque que le joueur qui commence gagne plus souvent, cet avantage disparaît après le 25ème coup. Étant donné que les deux joueurs suivent la même stratégie aléatoire, il est normal que le joueur qui commence ait l'avantage.
Aussi cela est dû au fait que le puissance 4 est un jeu résolu. Le premier s'il fait les meilleurs choix peut toujours gagner au maximum en 41 coups. Ce qui n'est pas le cas pour le deuxième joueur.

On voit aussi, dans les deux graphiques, que la distribution suit une courbe de Gauss.
Étant donné que l'on a beaucoup d'échantillon, cela semble suivre une loi normale. Or la loi normale est une loi à densité.

Nos variables de nombre de coups sont des variables discrètes indépendantes prenant leurs valeurs entre 0 et 42, impair pour le joueur qui commence, pair pour l'autre joueur.
Une explication possible est le théorème central limite, qui affirme en particulier que :
« Étant donné une variable Xn de loi binomiale B(n; p), la variable centrée réduite associé, converge en loi vers la loi normale centrée réduite lorsque n tend vers l'infini ».

Dans notre cas nous prenons de très grands échantillons (1 000 000), le théorème peut donc s'appliquer. Cela expliquerait pourquoi la forme de la courbe est gaussienne.

Nous avons donc décidé d'écrire des fonctions pour calculer l'espérance et l'écart type des histogrammes trouvés afin de trouver ses paramètres.

Pour le deuxième histogramme, nous trouvons une espérance à 20,62 et un écart type de 7,46 pour le joueur 1. Le joueur -1 a une espérance à 22,07 et un écart type de 7,05.

Pour le deuxième histogramme, nous trouvons une espérance à 22,06 et un écart type de 7,05 pour le joueur 1. Le joueur -1 a une espérance à 20,06 et un écart type de 7,44.

Donc le plus souvent, une partie finit entre le 20ème et le 22ème coup, de plus il est très rare de finir sur une partie nulle. Nous allons vérifier cela par la suite.

1 – Étude des parties nulles :

-Approche théorique :

Voici l'approche que nous avons choisi pour trouver la borne théorique d'une partie nulle :

Grâce à notre fonction *quadrupletsGagnants()*, nous sommes certains que le jeu ne possède que 69 quadruplets. Donc si un des joueurs obtient un quadruplet gagnant, la partie ne sera pas nulle.

Pour effectuer le calcul nous allons nous soumettre à une hypothèse h_1 :
« Jouer une partie jusqu'à ce que toutes les cases soient occupées même si un des deux joueurs à gagner avant ».

Soit l'événement N : *« La partie est nulle »*
Le résultat est nul si aucun quadruplet présent sur le plateau rempli n'est gagnant.
Soit Q_x l'évènement :

« Le quadruplet n°x n'est pas gagnant ».

Soit un quadruplet, on fixe la couleur d'un de ces pions.
Nous savons que la probabilité que le pion soit rouge ou jaune est équiprobable.
Donc la probabilité qu'un pion soit de la même couleur qu'un pion fixé est de $\frac{1}{2}$.
On peut donc en conclure que :

$P(\text{complémentaire}(Q_x)) = (\frac{1}{2})^3$ -> mettre le vrai symbole conjugué

(Pour les 3 pions différents de celui fixé)

On peut donc en déduire Q_x :

$$P(Q_x) : 1 - (\frac{1}{2})^3 = 1 - \frac{1}{8} = \frac{7}{8}.$$

La partie est nulle si on a :

$$Q_1 \wedge Q_2 \wedge Q_3 \ldots \wedge Q_{64}.$$

Donc :

$$P(N) = (\frac{7}{8})^{69} = 0.00009966798 \simeq 0{,}01\ \%.$$

<u>- Approche pratique :</u>

Afin de trouver la probabilité d'avoir une partie nulle, nous simulons 10 000 parties. Pour chaque partie nulle nous incrémentons un compteur de 1. À la fin des simulations, nous obtenons donc le nombre de parties nulles totales.

On réitère cette expérience 100 fois. Au-delà le rapport entre la précision que nous gagnons sur notre résultat et le temps passé à réaliser l'expérience n'est plus intéressant.
Ensuite, nous faisons une moyenne du nombre de parties nulles obtenu sur chacun de nos paquets de simulations et ce chiffre représente bien la probabilité d'une partie nulle.

- Application :

Nous avons écrit les valeurs trouvées dans un fichier. Nous trouvons une moyenne de :

$$\frac{25,46}{10\ 000} = 0,002546\%$$

Le résultat expérimental est différent de celui théorique. Cela est sûrement dû aux hypothèses qui sont trop fortes.

3 – Nombre de parties possibles :

- Approche théorique :

Pour calculer le nombre de parties possibles du jeu, nous avons décidé de négliger le cas des parties nulles. Nous pouvons nous le permettre car la probabilité d'en faire une est très faible (cf. sous-partie précédente).

Nous allons donc uniquement calculer le cas des parties où un des joueurs sort vainqueur. De plus, nous allons reprendre l'hypothèse h_1 de la partie précédente :
« *Jouer une partie jusqu'à ce que toutes les cases soient occupées même si un des deux joueurs à gagner avant* ».

On se place donc dans le cas d'un plateau rempli et dans lequel il y a un quadruplet gagnant.

On ajoute également l'hypothèse h_2 :
« *On ne fera les calculs qu'en fonction d'un seul des quadruplets fixés* »

Cela nous évitera de traiter les cas où deux quadruplets gagnants sont présents sur un même plateau rempli. »

On prend par exemple le quadruplet n°1 : il est fixé.
Nous allons calculer le nombre de plateau rempli possible avec ce quadruplet.
Il reste donc 38 (42-4) pions non fixés dans le plateau avec 17 (21-4) pions restants de la même couleur que les pions du quadruplet n°1 et 21 pions de l'autre couleur.
Le nombre de combinaisons des pions restants est :

$$\frac{38!}{21!*17!}$$

Comme 69 quadruplets sont possibles, pour une couleur fixée on a :

$$69 * \frac{38!}{21!*17!}$$

Pour les deux joueurs on obtient donc :

$$2 * 69 * \frac{38!}{21!*17!} = 2.3830787e+13$$

Soit 2.3830787e+13 plateaux possibles.

<u>- Approche pratique :</u>

Nous avons fait une fonction « CalculNbPartiesDifferentes » qui calcul le nombre de parties possibles dans un jeu Puissance 4.
Voici le code python :

```
def copiePlateau(plateau):
        nouveauPlateau = Plateau(nb_ligne,nb_colonne)
        for x in range(0,nb_ligne):
                for y in range(0,nb_colonne):
                        nouveauPlateau.plateau[x][y] = plateau.plateau[x][y]


        return nouveauPlateau

# Retourne le nombre de parties différentes possibles
def calculNbPartiesDifferentes(jeu,cpt):


        compteurPartie = 0
```

```
        if(jeu.isFinished()):
                return 1

        elif(cpt % 2 == 0):
                listeCoupsPossibles = jeu.plateau.coupsPossibles()

        for coup in listeCoupsPossibles:
                nouveauJeu = Jeu(jeu.plateau,jeu.joueur1,jeu.joueur2)
                nouveauJeu.plateau = copiePlateau(jeu.plateau)
                nouveauJeu.plateau.ajouterPion(coup,jeu.joueur1)
                compteurPartie = compteurPartie +
                calculNbPartiesDifferentes(nouveauJeu,cpt + 1)

        else:

                listeCoupsPossibles = jeu.plateau.coupsPossibles()

                for coup in listeCoupsPossibles:
                nouveauJeu = Jeu(jeu.plateau,jeu.joueur1,jeu.joueur2)
                nouveauJeu.plateau = copiePlateau(jeu.plateau)
                nouveauJeu.plateau.ajouterPion(coup,jeu.joueur2)
                compteurPartie = compteurPartie +
calculNbPartiesDifferentes(nouveauJeu,cpt + 1)

        return compteurPartie
```

Description :
La fonction prend en paramètre l'état actuel du jeu.
Pour chaque coup possible (colonne non remplie), un nouveau plateau est créé dans lequel le joueur courant joue ce coup, puis on fait un appel récursif avec le nouvel état du jeu.
Si la partie est finie on retourne 1, ce qui correspond à une partie possible.

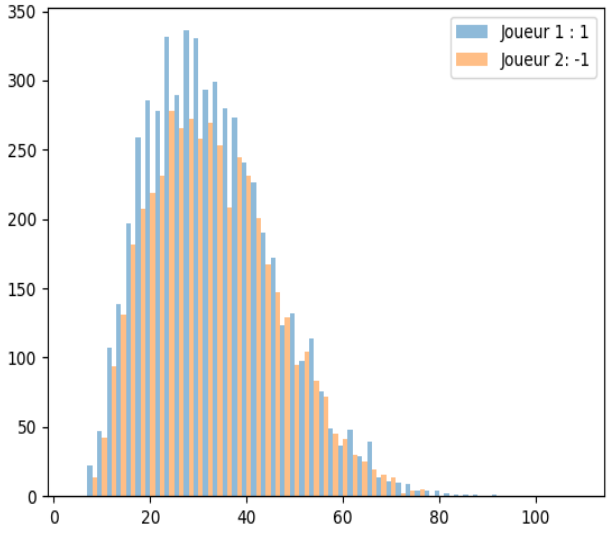
Nous ne pouvons pas vérifier notre résultat théorique étant donné que nos machines ne nous permettent pas d'arriver au terme d'un tel programme. Cependant nous avons testé la fonction pour des plateaux de dimensions inférieures et les résultats correspondaient.

4 – Études avec différentes dimensions :

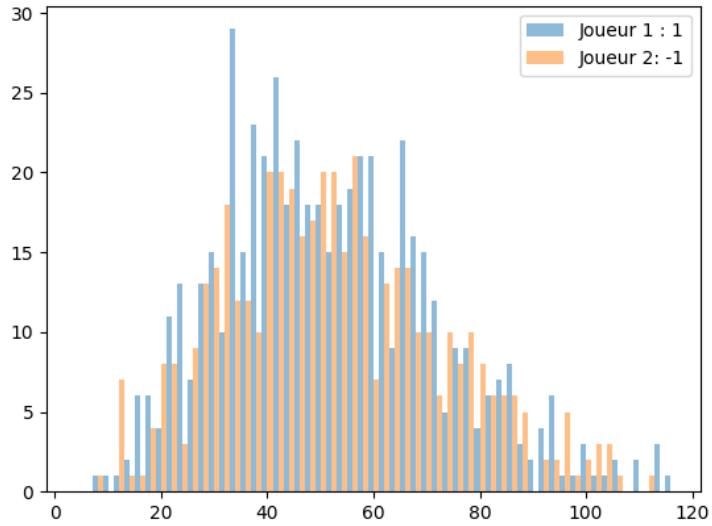
Nous allons nous intéresser ici à l'impact des dimensions du plateau sur les lois que nous avons étudié précédemment.

- Étude de la distribution du nombre de coups avant une victoire :

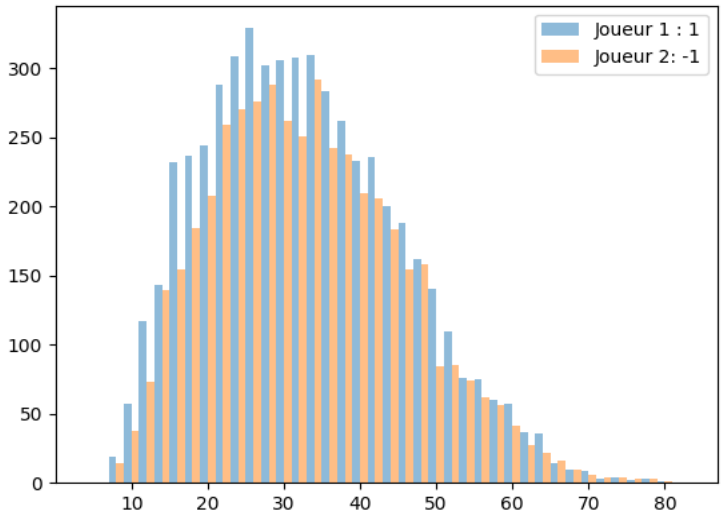
Pour 19 lignes et 20 colonnes (10 000 parties) :



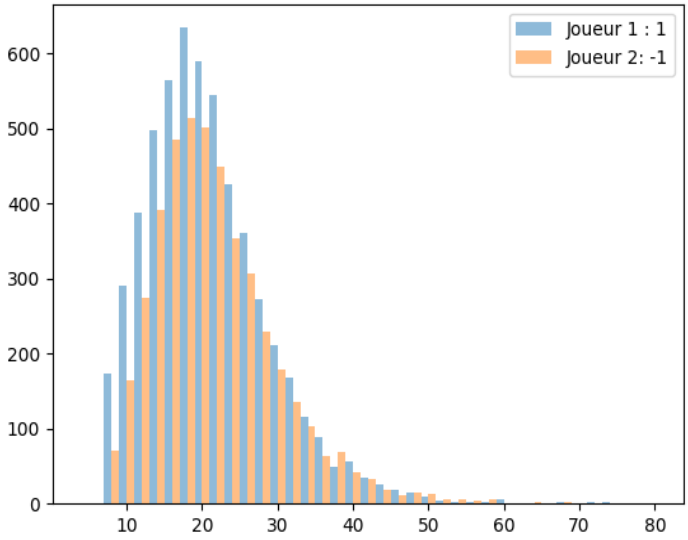
Pour 49 lignes et 20 colonnes (1000 parties) :



Pour 5 lignes et 20 colonnes (10 000 parties):



Pour 20 lignes et 5 colonnes (10 000 parties) :



Pour toutes les dimensions appliquées, le joueur qui joue en premier conserve son avantage et les histogrammes suivent encore une Gaussienne. Comme précédemment, on en déduit que la loi binomiale a convergé en une loi normale.
Le changement sera au niveau de la distribution du nombre de coups avant victoire qui n'est plus égale. Le minimum reste toujours 7 coups mais le maximum devient nbLignes ¿ nbColonnes.

<u>- Probabilité partie nulle :</u>

Nous allons répéter la même expérience que dans la partie 2 :

Pour 8 lignes et 9 colonnes (10 000 parties) :
On trouve un total de 0 parties nulles avec 153 quadruplets.

Pour 7 lignes et 8 colonnes (10 000 parties) :
On trouve un total de 0,02 % parties nulles avec 107 quadruplets.

Pour 5 lignes et 6 colonnes (10 000 parties) :
On trouve un total de 3,5% de parties nulles avec 39 quadruplets.

Pour 4 lignes et 5 colonnes (10 000 parties) :
On trouve 25% de parties nulles avec 17 quadruplets.

Quand les dimensions du plateau sont supérieures ou égales à celle d'un plateau 8x9, le nombre de partie nulle est de 0.
En effet la probabilité d'en obtenir une est tellement faible qu'on ne peut pas la représenter.

On remarque qu'il y a une relation entre la dimension du plateau et la probabilité d'obtenir une partie nulle en faisant s'affronter des joueurs aléatoires.
Plus le nombre de quadruplets possibles dans le plateau est faible plus le pourcentage de parties nulles est élevé. En effet le nombre de quadruplets possibles définit les manières de gagner.
Aussi cela concorde avec l'approche théorique faite :

$$P(N) = (\frac{7}{8})^n$$

Avec n le nombre de quadruplets possibles sur le plateau.

2 – Bandit manchot et algorithmes associés :

Principe :

Imaginons une machine à sous à k leviers. Chacun de ses leviers possèdent une probabilité de succès p qui lui est propre. Le tirage d'un levier suit donc une épreuve de Bernoulli de paramètre p.
Nous ne connaissons rien des leviers.

Le problème est le suivant :
« Pour un nombre de parties N, comment s'approcher au maximum du gain optimal ? »
Le gain optimal étant le gain obtenu si l'on tire N fois le levier avec le meilleur rendement.
Nous sommes donc face à un dilemme Exploration/Exploitation :
À chaque tirage, est-ce que l'on doit choisir le levier avec le meilleur rendement selon nous (Exploitation) ou alors tirer sur un autre levier pour espérer en trouver un meilleur (Exploration) ?

Dans cette partie nous allons essayer plusieurs algorithmes suivant différentes stratégies d'Exploration/Exploitation pour nous approcher du gain optimal, les voici :

- Aléatoire (Baseline) : À chaque tirage, le levier est choisi aléatoirement.
- Greedy : Un certain nombre de tirage sont d'abord consacrés uniquement à l'exploration, puis on exploite en jouant uniquement le levier avec le meilleur rendement selon nos approximations.
- ε-Greedy : Un certain nombre de tirage sont d'abord consacrés uniquement à l'exploration. Ensuite à chaque tirage, on a une probabilité ε de choisir aléatoirement un levier (explorer) et $(1 - \varepsilon)$ de choisir le levier avec le meilleur rendement selon nos approximations (exploiter).
- UCB : À chaque tirage, on calcule pour chaque levier :

$$Rit + \sqrt{\frac{2log(t)}{Nt(i)}}$$

Avec :
t : le nombre de tirage actuel
Nt(i) : le nombre de fois où le levier i a été choisi jusqu'au temps t
Rit : le rendement approximé du levier i au temps t

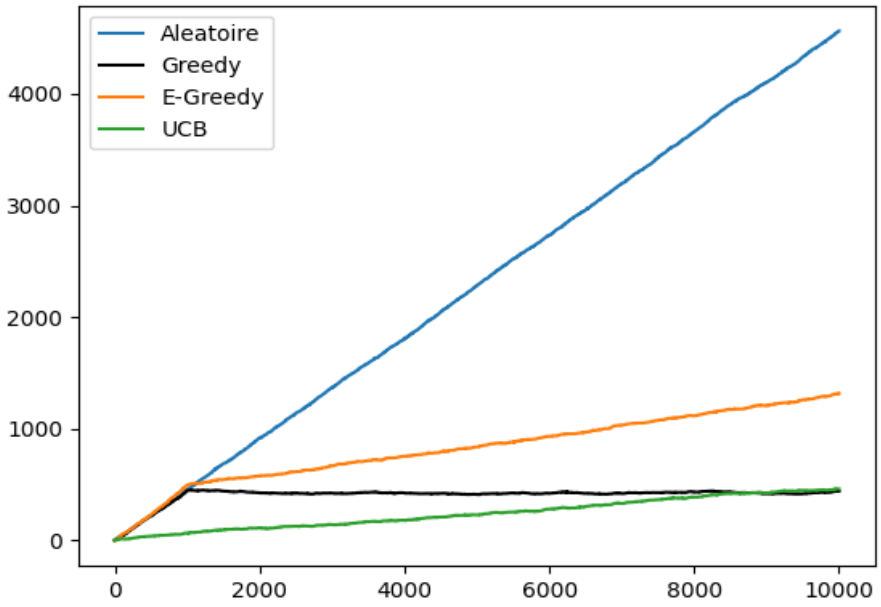
On choisit le levier pour lequel cette valeur est un maximum.
On voit que le deuxième terme décroit lorsque Nt(i) augmente, ce qui veut dire que cet algorithme va choisir le levier avec le meilleur rendement sauf si celui-ci a été joué beaucoup plus de fois que les autres auxquels cas il choisira celui pour lequel Nt(i) est un minimum.

1 - Comparaison des différents algorithmes :

Nous allons comparer les algorithmes en changeant certains paramètres pour étudier leur impact.
Pour toutes les simulations qui suivent, on aura 10% d'exploration et 90% d'exploitation.

- Variation du nombre de leviers :

Nous commençons par fixer 10 leviers ayant des valeurs (probabilité de succès) comprises entre 0 et 1 avec un pas de 0,1.
On obtient ce graphique qui correspond au regret (Gain Optimal – Gain obtenu) par rapport au nombre de tirage :



L'algorithme aléatoire suit une droite linéaire, en effet il ne fait qu'explorer et n'exploite jamais. À chaque tirage on a :

$$P(\text{« Choisir le meilleur levier »}) = \frac{1}{k}$$

Avec k le nombre de leviers.
Donc, comme les tirages sont indépendants :

$$P(\text{« Le regret augmente »}) = \frac{k-1}{k}$$

$$P(\text{« Le regret reste constant »}) = \frac{1}{k}$$

D'où l'augmentation linéaire.

L'algorithme Greedy est très efficace dans ce cas.
On remarque une augmentation du regret, identique à celle de l'algorithme aléatoire, sur l'intervalle [0 ; 1 000] qui correspond à la phase d'exploration. Cette phase d'exploration revient à faire exactement l'algorithme aléatoire sur cet intervalle, ce qui explique pourquoi les deux droites sont identiques.
Sur l'intervalle]1 000 ; 10 000], la droite devient constante. C'est l'intervalle où l'algorithme passe à la phase d'exploitation, comme ici le nombre de levier est faible on est sûr que l'algorithme a trouvé le meilleur levier et qu'il le joue pendant toute cette phase.
On a donc :

$$P(\text{« Choisir le meilleur levier »}) = 1$$

Le regret reste donc constant sur]1 000 ; 10 000]. (Il décroit légèrement, l'algorithme Greedy a donc eu plus de succès que l'algorithme optimal pour le même levier.)

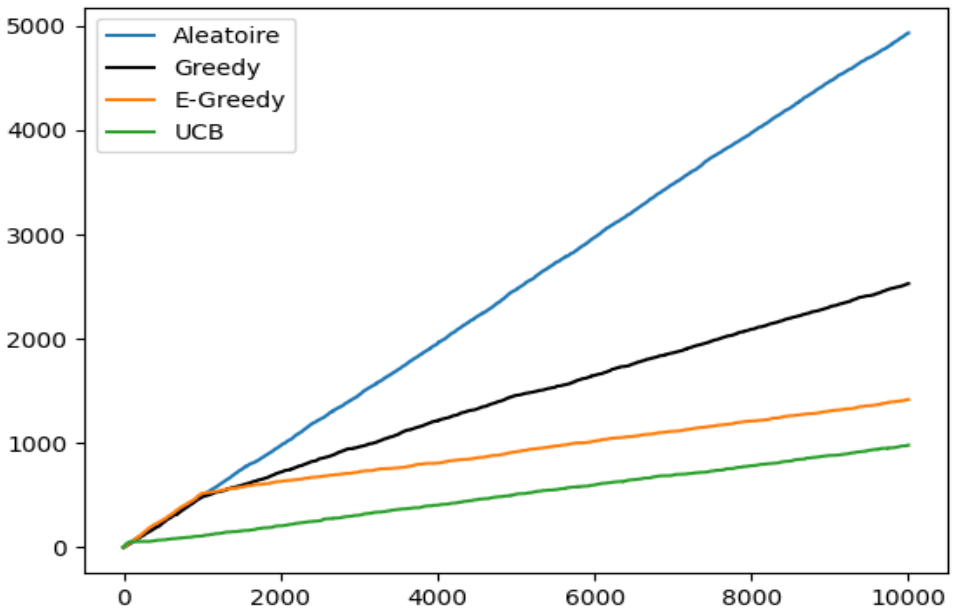
Pour l'algorithme ε-Greedy, on remarque l'augmentation du regret associé à la phase d'exploration initiale sur l'intervalle [0 ; 1 000].
Sur l'intervalle]1 000 ; 10 000], on constate une augmentation modérée du regret.
En effet on a :

$$P(\text{« Choisir le meilleur levier approximé »}) = (1 - \varepsilon)$$

Dans ces simulations, $\varepsilon = 0{,}3$, d'où la croissance modérée.
Le fait que Greedy soit meilleur est dû au nombre de leviers. Nous verrons cette influence dans la partie suivante.

Pour l’algorithme UCB, il n’y a pas d’augmentation initiale du regret car nous n’avons pas de phase d’exploration. L’augmentation est modérée tout au long des tirages car il choisit en priorité l’exploitation mais alterne avec des tirages d’exploration sur les leviers les moins joués pour améliorer ses approximations sur ceux-ci.
Il est ici meilleur que Greedy mais tend à ne plus l’être sur les derniers tirages. En continuant d’exploiter il augmente son regret dans le temps, là où Greedy ne le fait plus.

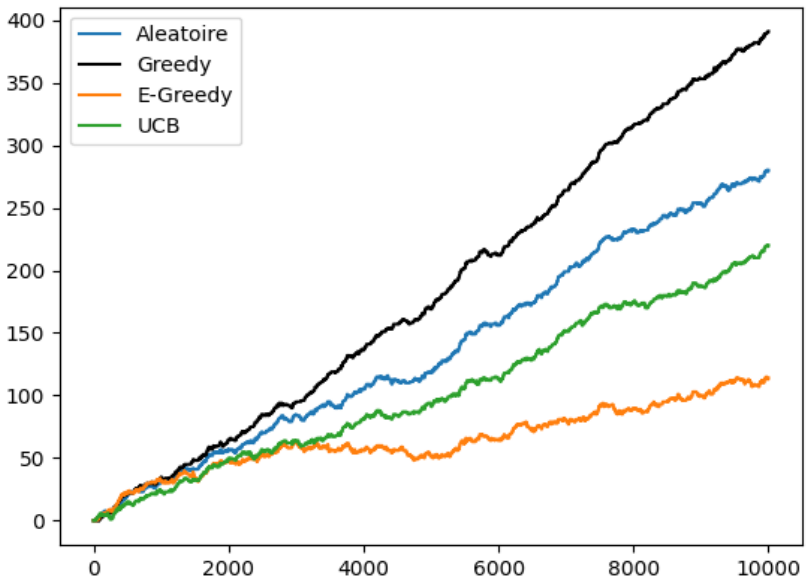
Nous allons maintenant fixer 100 leviers à valeurs dans [0 ; 1] avec un pas de 0,01.
Nous obtenons ce graphique :



On voit que l’augmentation du nombre de leviers n’affecte que l’algorithme Greedy.
En effet dans ce cas-là, nous ne sommes plus sûr qu’il trouve le meilleur levier. Le nombre de leviers ayant augmenté, les approximations des rendements de chacun seront moins précises et il pourra passer à côté du meilleur.
Les algorithmes explorant continuellement comme UCB ou ε-Greedy sont donc avantagés car leurs approximations s’affineront avec l’augmentation du nombre de tirages. Ils trouveront donc à un moment le meilleur levier.

- Variations de l'écart entre les lois de Bernoulli :

Dans le graphique suivant, nous avons mis 200 leviers à valeurs dans [0 ; 0,15]. L'écart est donc très faible entre les valeurs.
Nous obtenons ce graphique :



Nous remarquons ici que tous les algorithmes sont affectés.
Le regret associé à l'algorithme Greedy est supérieur à celui de la baseline (aléatoire).
En effet les probabilités de succès de chaque levier sont très proches et ceux-ci sont nombreux. La phase d'exploration de Greedy ne permet pas de déterminer efficacement un bon levier.
On a que le nombre de tirage dédié à un levier est : $\frac{nbTirage}{nbLeviers} = \frac{1\ 000}{200} = 5$
Ce nombre est trop faible pour discerner un bon rendement d'un mauvais dans le cas où p appartient à [0 ; 0,15].

L'algorithme ε-Greedy est le meilleur dans ce cas. Cela peut s'expliquer par le fait que son exploration se produit de manière aléatoire, dans un cas particulier comme celui-ci où les rendements sont très proches, l'exploration aléatoire peut s'avérer être une bonne stratégie. Elle peut, pour un nombre conséquent de tirage, se rapprocher d'une exploitation.

3 – Algorithme Monte-Carlo :

Principe :

Le but de l'algorithme Monte-Carlo est d'ajouter une phase de simulation avant de choisir le coup à jouer.
Cette phase de simulation est la suivante :

Pour un certain nombre d'itérations, à chacune d'entre elles :
- L'algorithme choisit un coup possible aléatoirement
- Il joue ce coup dans une copie du plateau actuel
- Il termine la partie en jouant aléatoirement deux joueurs
- Le rendement du coup joué est mis à jour en fonction du résultat

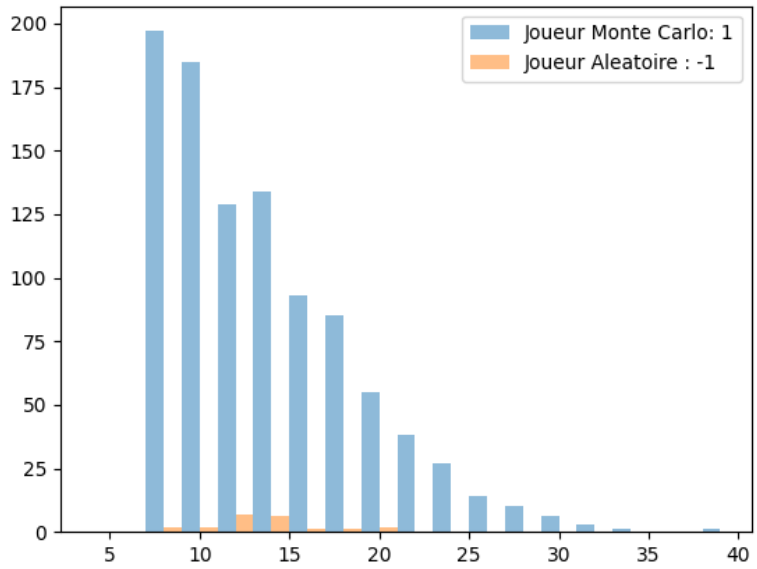
L'algorithme joue ensuite le coup avec le meilleur rendement.

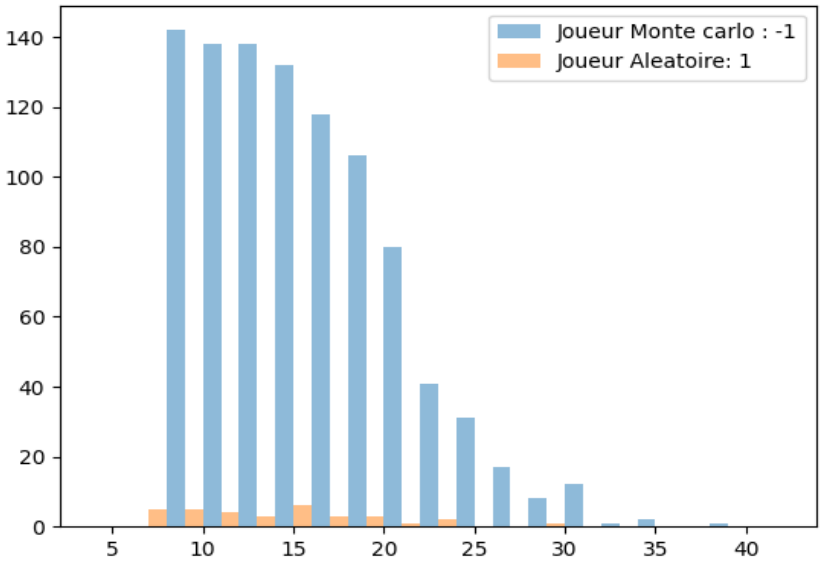
On reconnait le problème du bandit manchot, ici les leviers sont les coups possibles lors d'une partie de puissance 4 et le nombre de tirage est le nombre d'itérations.
L'algorithme utilisé dans le cas de ce Monte-Carlo pour répondre au problème du bandit manchot est l'algorithme aléatoire, la baseline.

Pour la suite, chaque graphique comportera 1 000 simulations et le nombre d'itérations i pour Monte-Carlo sera de 100.
Aussi le joueur 1 est celui qui commence.

1 – Joueur Monte-Carlo contre le joueur aléatoire :

Voici les graphiques obtenus :





Le Joueur Monte-Carlo gagne presque toutes les parties, dans notre simulation, il obtient $\frac{9\ 975}{10\ 000}$ parties gagnées. Cela revient à un pourcentage de victoire de 99,75%. On peut d'ailleurs compléter ce qu'on a dit dans la partie sur l'algorithme aléatoire, la partie de puissance 4 suit une épreuve de Bernoulli de paramètre 0,9975 (approximation).

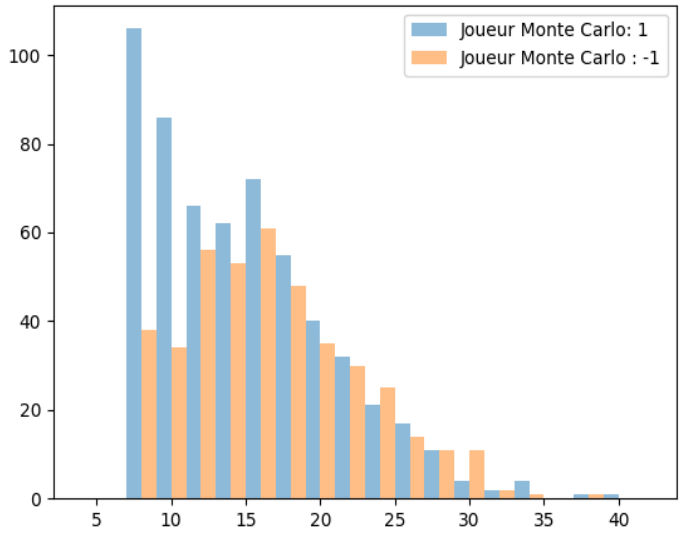
On remarque que plus la partie avance, plus il est rare que Monte-Carlo gagne. Encore une fois le nombre de quadruplets gagnants diminue donc le nombre de façon de gagner diminue.

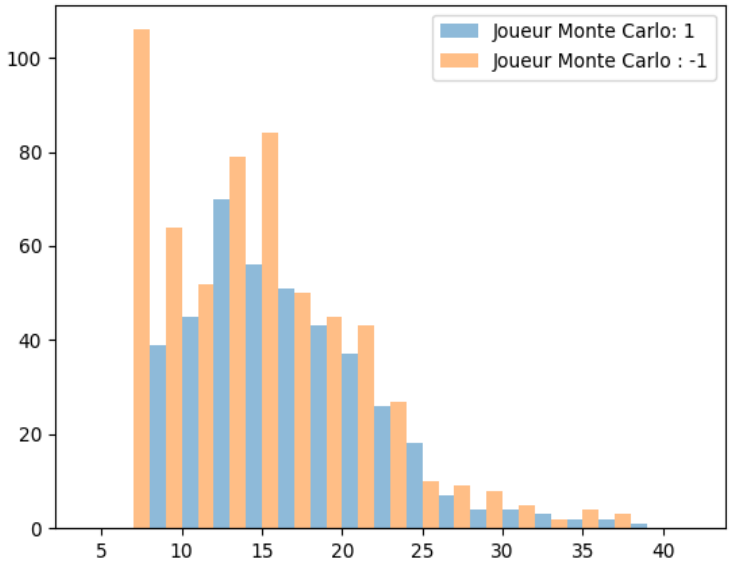
L'avantage conféré au premier joueur est maintenu, on remarque que l'algorithme aléatoire gagne un peu plus quand il commence.

En ce qui concerne l'allure Gaussienne que l'on avait dans la partie sur l'algorithme aléatoire, celle-ci n'est plus maintenue. Ceci est dû au fait que le joueur Monte-Carlo ne joue pas au hasard. Il n'y a donc plus équiprobabilité entre les coups. On ne peut donc pas retrouver une loi binomiale. En plus de cela, l'échantillon restant de victoire pour le joueur aléatoire est trop faible pour être analysé.

2 – Joueur Monte-Carlo contre lui-même :

Voici les graphiques obtenus :





On voit que les deux histogrammes ont des allures assez symétriques ce qui veut dire que le seul facteur influant ici est l'ordre de commencement. Cela s'explique par le fait que les deux joueurs suivent une stratégie identique.
Le joueur commençant ayant un net avantage au début.
Cet avantage diminue avec l'avancement de la partie. En réalité l'avantage repose sur le fait que le joueur 1 puisse empiler ou aligner 4 de ses pions sans que le joueur 2 ne l'ai bloqué (ce qui explique le pic à x = 7), si celui-ci le bloque alors le joueur 1 devra à son tour bloquer les pions placés par le joueur 2 et la partie suivra donc son cours normal.

Encore une fois l’allure Gaussienne n’est pas maintenue pour les mêmes raisons.

4 – Algorithme UCT :

Principe :

Le but de l’algorithme UCT est d’améliorer la phase de simulation de l’algorithme Monte-Carlo.
Les différences sont les suivantes :

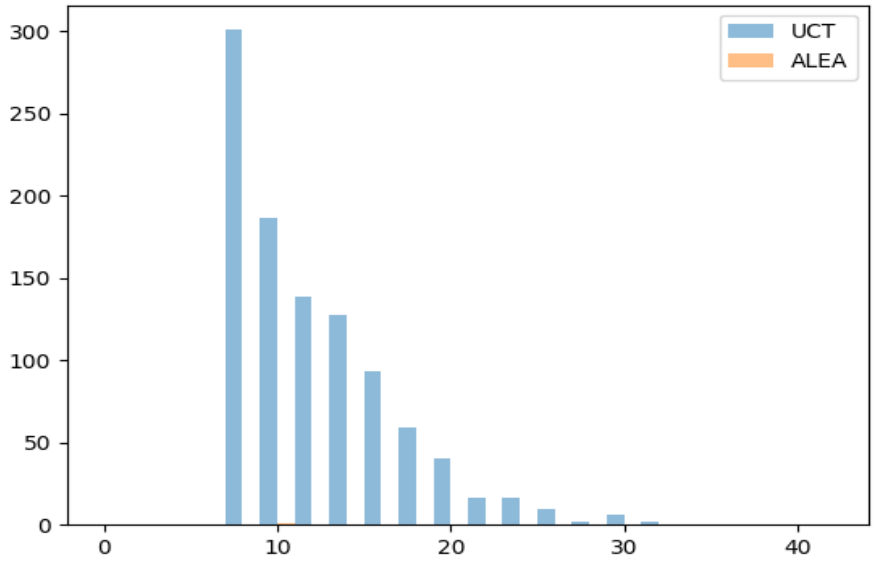
- Au lieu de jouer uniquement les coups possibles à partir de l’état courant, UCT jouera aussi les coups possibles à partir des nouveaux états qu’il crée et ainsi de suite. Cela permet de ne pas se baser uniquement sur l’aléatoire pour déterminer le résultat d’une partie.
- Au lieu de choisir aléatoirement un coup parmi tous les coups possibles à partir d’un état, UCT choisira ce coup en utilisant l’algorithme UCB.

L’algorithme utilisé dans le cas d’UCT pour répondre au problème du bandit manchot est donc l’algorithme UCB.

Pour la suite, le nombre de simulations par graphique sera de 1 000 et le nombre d’itérations sera de 100 quand on ne le précisera pas.

1 – UCT contre aléatoire :

Nous avons d'abord fait jouer le joueur UCT contre le joueur aléatoire afin de vérifier son bon fonctionnement. Nous avons fait un échantillon de 1 000 parties.

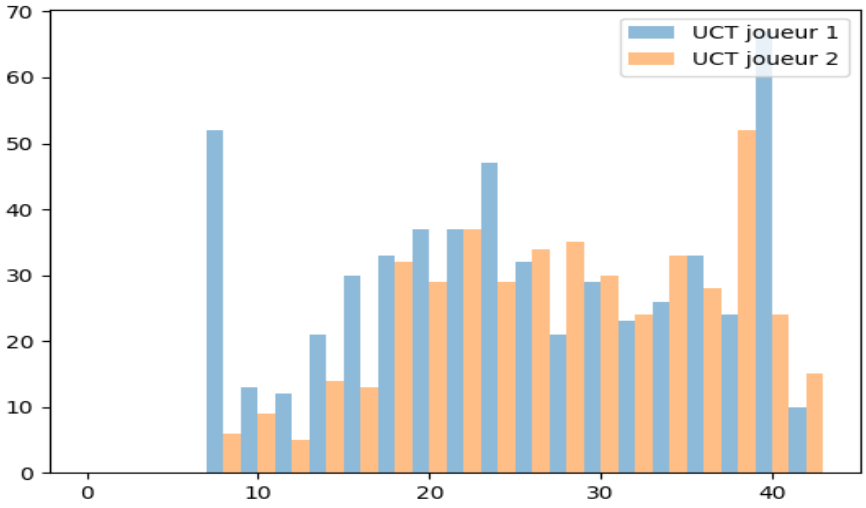


On voit bien que sa victoire est totale, l'algorithme UCT a gagné toutes ses parties contre le joueur aléatoire

Nous allons donc maintenant le tester face à lui-même pour voir si le joueur qui commence la partie a toujours l'avantage.

2 – UCT contre lui-même :

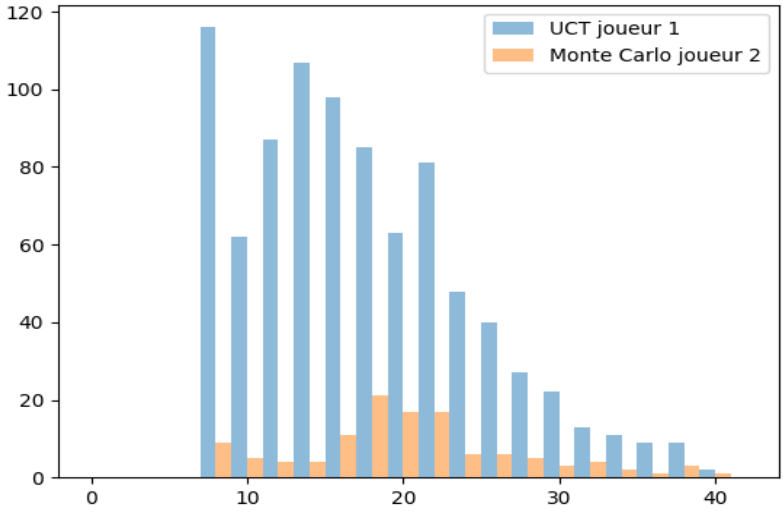
Nous avons tracé l'histogramme des coups gagnants de UCT contre UCT.

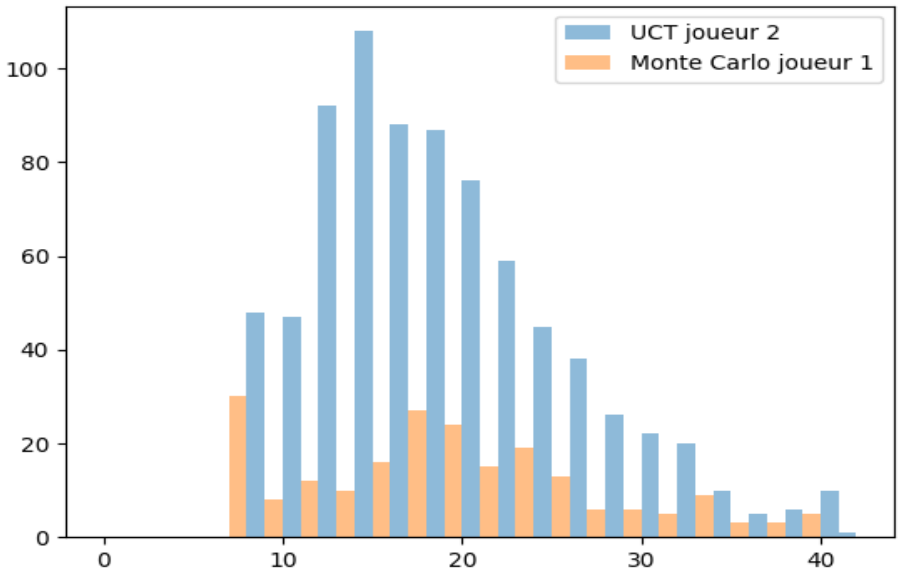


On remarque que l'avantage de premier joueur est toujours présent.

Nous allons maintenant étudier l'algorithme UCT contre Monte-Carlo afin de comparer la méthode de simulation au sein des deux algorithmes.

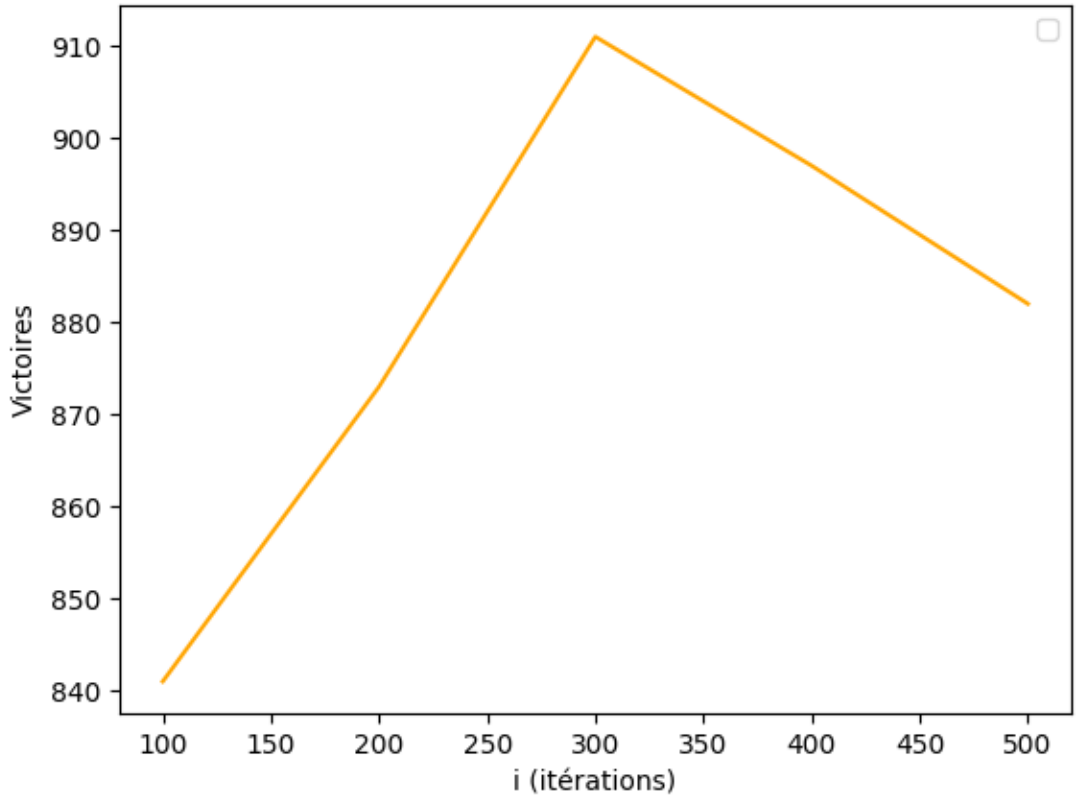
3 – UCT contre Monte-Carlo :





Dans le cas où le joueur UCT commence, il a gagné 840 fois, contre 788 quand il a joué en deuxième. Il reste tout de même très efficace.

Nous allons maintenant augmenter le nombre d'itérations des deux algorithmes de 100 à 500 par pas de 100. Nous comparerons le nombre de victoire d'UCT afin de voir si une amélioration du ratio : $\frac{victoireUCT}{défaiteUCT}$ est visible.



De 100 à 300 itérations, on voit une amélioration nette jusqu'à plus de 90% de victoires. Ensuite l'augmentation de l'itération ne fait pas évoluer le ratio.

5 – Conclusion :

Nous avons cherché à trouver un bon algorithme de décision pour le puissance 4.
La stratégie de base est la stratégie aléatoire, cela nous permet de lui confronter nos autres algorithmes pour prouver leur efficacité.

Le problème étant de trouver le meilleur coup à jouer, nous pouvions le représenter à l'aide du problème du bandit manchot.
Plusieurs algorithmes sont adaptés pour répondre à ce problème, l'efficacité de certains dépend de l'environnement dans lequel ils sont utilisés (ex : nombre de coups (Leviers) possibles, distribution des rendements), c'est le cas de l'algorithme Greedy.

Nous avons donc pu ajouter, dans nos algorithmes, une phase de simulation basée sur le bandit manchot.

Pour Monte-Carlo, on applique l'algorithme aléatoire de bandit manchot pour sélectionner les coups possibles.
Pour UCT, on applique l'algorithme UCB de bandit manchot pour sélectionner les coups possibles.

Pour conclure, UCT est le meilleur algorithme que nous avons fait. C'est, en général, un très bon algorithme pour cette tâche. Il est notamment utilisé dans les algorithmes de Go.