CAZES Hadrien , SY VIRAK BOT Baramy
26 février 2019

Rapport projet 1:

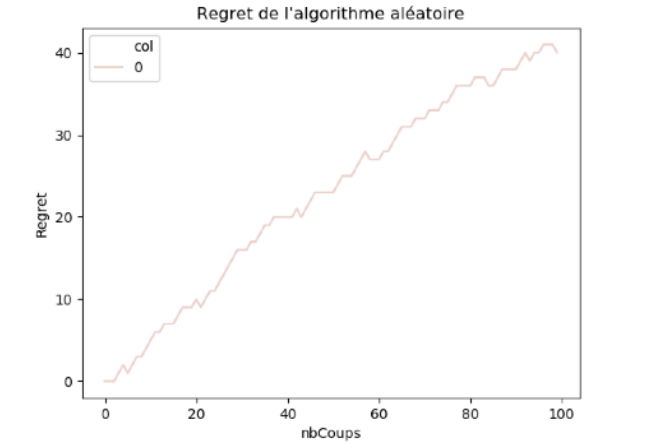
Exercice 1 :

Pour l'exercice sur les bandits manchots , nous devions représenter le regret en fonction des différents algorithmes à implémenter et des différents paramètres qui influent selon les algorithmes.

Pour avoir des résultats cohérents et pour permettre la comparaison des algorithmes , nous avons décidé de fixer le nombre de levier dès le début de notre script Python à 4.

Pour l'algorithme aléatoire , le seul paramètre ayant une influence est le nombre de fois qu'on tirera un levier , à savoir nbCoups dans notre main.

En effet , l'univers qui consiste au choix d'un levier de façon aléatoire est un ensemble fini et dénombrable :{0,…, nbLevier -1} où tout événement élémentaire w a la même chance d'apparition. Il y a donc équiprobabilité. La chance d'obtenir le meilleur levier est la même que celle d'obtenir un autre levier. On ne tiendra donc pas compte du nombre de leviers dans notre estimation du regret.

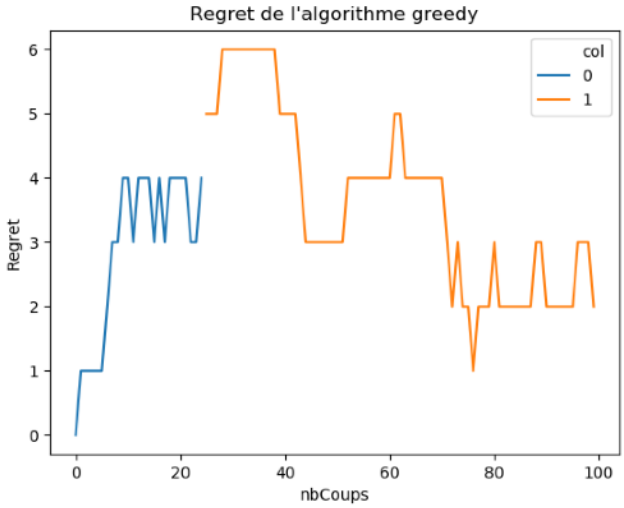


On constate que le regret est une fonction affine de la forme y = ax + b. La forme de la fonction de regret n'est pas totalement lisse d'après le graphe. Cela est dû au fait que la fonction déterminant le regret est basée sur une fonction qui retourne un rendement aléatoire (0 ou 1) en fonction de la probabilité de gain du levier choisi. Prendre le meilleur levier (à moins qu'il aie une probabilité de gain de 1.0) ne garantit pas de gagner à tous les coups. Il est normal que le gain maximal espéré puisse avoir tendance à se rapprocher de notre gain actuel.

Passons à l'algorithme glouton , qui est un algorithme nécessitant une exploration au préalable afin de pouvoir exploiter ses expériences pour choisir le meilleur levier. Pour cela , nous avons divisé dans notre script Python la partie exploration et la partie exploitation. L'appel à l'algorithme glouton est donc un appel à une exploitation et nous nous chargeons de faire l'exploration à l'intérieur de notre fonction main. On passera en argument de l'algorithme glouton : le tableau coups_effectuées et le tableau proba_estimée.

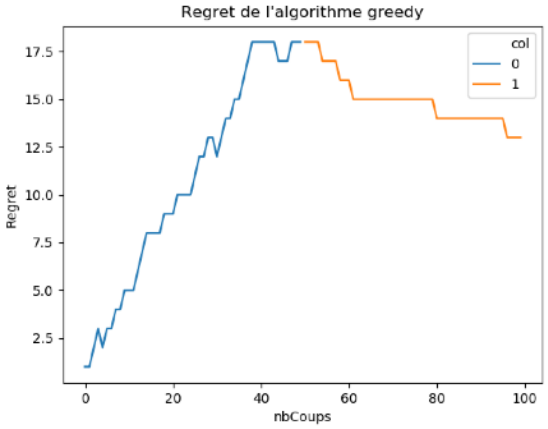
- Le tableau coups_effectuées recense pour chacun des leviers le nombre de fois que le levier a été utilisé.
- Pour chacun des leviers et en fonction de notre expérience (le gain) au travers de l'exploration et de l'exploitation , on met à jour la probabilité que l'on pense être correcte dans le tableau proba_estimée.

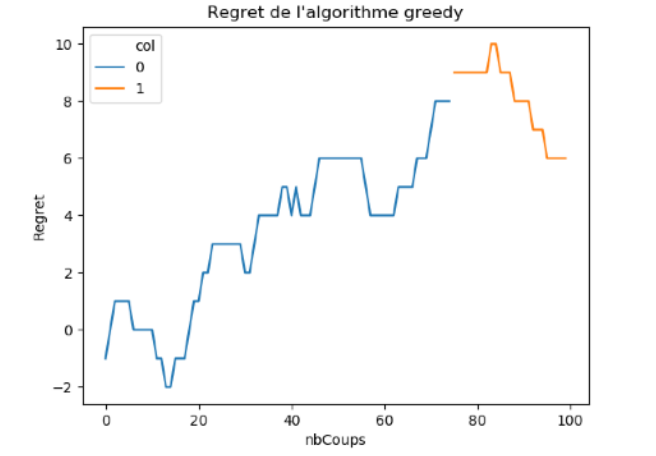
Les paramètres qui influent sur le regret sont le nombre de fois qu'on joue un des leviers et le nombre de fois qu'on va explorer. Prenons nbCoups = 100 et faisons varier le nombre de coups dédiés à l'exploration: nbExploration = 25



On constate que le regret est une fonction croissante lors de l'exploration représentée en bleu sur la courbe. Il est normal de constater un regret de cette forme , la raison est que l'exploration exploite l'algorithme aléatoire pour sélectionner un levier uniformément. Une fois les 25 coups dédiés à l'exploration terminée , on obtient un tableau des probabilités estimées qui ne contient pas obligatoirement des probabilités proches de la vraie probabilité exacte de gagner , cela est dû au choix uniforme des leviers qui ne considère pas forcément tous les leviers ou prends des leviers avec de moins bonnes probabilités. L'algorithme glouton prends alors ce qu'il estime comme le meilleur levier actuel d'après ses probabilités calculées et prends au final un « mauvais levier » (au début de l'exploitation sur le graphe). À l'issue de ses expériences lors de l'exploitation , il raffine ses probabilités évaluées auparavant pour trouver un autre meilleur levier , d'où la baisse du regret vers les dernières coups à jouer.

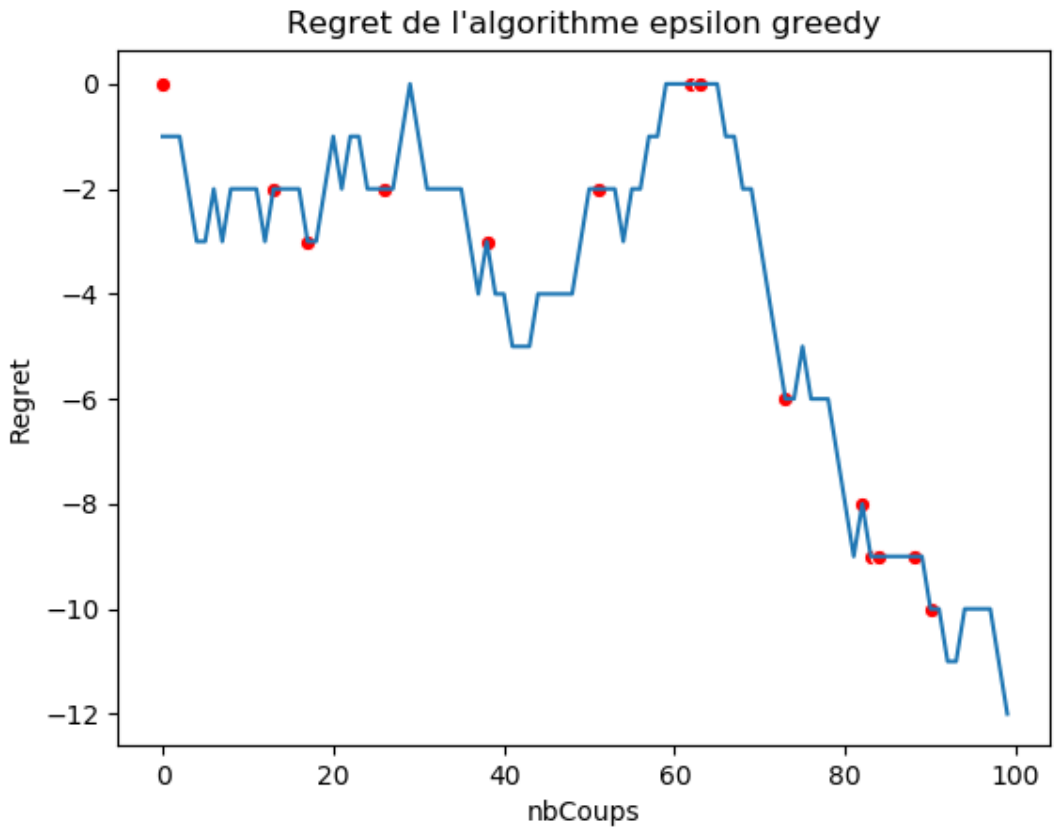
Prenons un autre nombre de coups dédiées à l'exploration : nbExploration = 50 puis 75. On obtient les graphes suivants:





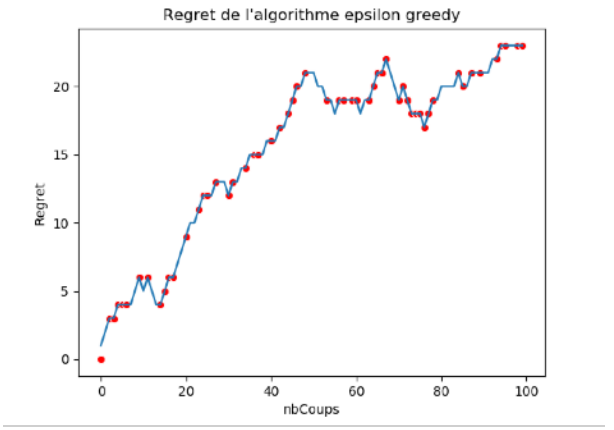
On constate que l'exploration ne change pas et est toujours croissante hormis le fait que dans le graphe de droite , il y a de rares occasions où l'exploration nous fait jouer un bon levier. Lorsque le nombre de coups dédiées à l'exploration est suffisamment grand par rapport au nombre de coups total , on remarque assez facilement que le regret diminue assez rapidement , car nos probabilités estimées ont été suffisamment raffinées pour se « rapprocher » des probabilités exactes.

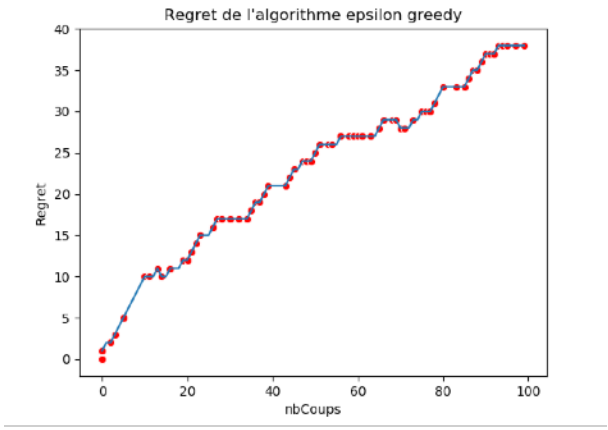
Prenons epsilon greedy maintenant , epsilon est la probabilité de choisir entre l'exploration et l'exploitation. Dans notre main , nous avons choisi donc de lancer l'algorithme nbCoups = 100 fois. Le seul paramètre faisant varier le regret sera epsilon . Dans notre code , epsilon est considéré comme une variable globale. Prenons epsilon = 0.1, nous avons donc 90% de chance de faire de l'exploitation et 10% de chance de faire de l'exploration. On obtient ce graphe:



Les points rouges représentent des coups dédiées à l'explorations. L'exploitation est représentée par la courbe bleue. On constate que peu d'exploration ont été effectuées donc le regret n'est pas optimal au début. Puis au fur et à mesure que les coups sont joués et que les probabilités estimées sont mises à jour , on a bien une descente du regret car l'exploitation choisit des probabilités plus raffinées.

Prenons un epsilon plus élevé , cela sous-entend qu'il y aura une plus grande probabilité de faire de l'exploration et moins de faire de l'exploitation. Si on choisit epsilon = 0.5 puis 0.7 , nous aurons les graphes suivants:

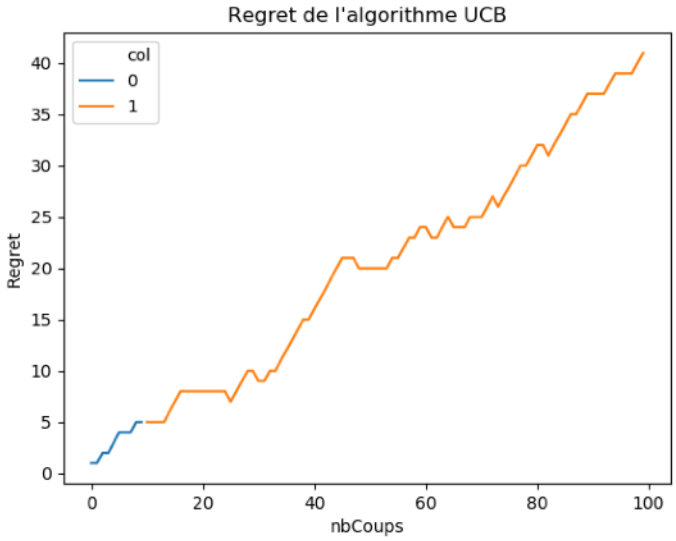




On constate que les 2 graphes suivent la même tendance que l'algorithme aléatoire , ce qui est logique car nous faisons principalement de l'exploration (d'où le nombre important de points). Il est donc logique que le regret soit élevé et ressemble davantage au regret de l'algorithme aléatoire. Il est donc plus intéressant de prendre un epsilon petit afin d'obtenir un regret bas.

Regardons maintenant le comportement du regret sur l'algorithme UCB (upper confidence bound) qui est une variante de l'algorithme glouton dans notre script Python. De la même façon que l'algorithme glouton , les paramètres à varier seront le nombre de coups à jouer nbCoups et le nombre de coups dédiés à l'exploration nbExploration. Nous fixons nbCoups à 100 et faisons varier nbExploration.

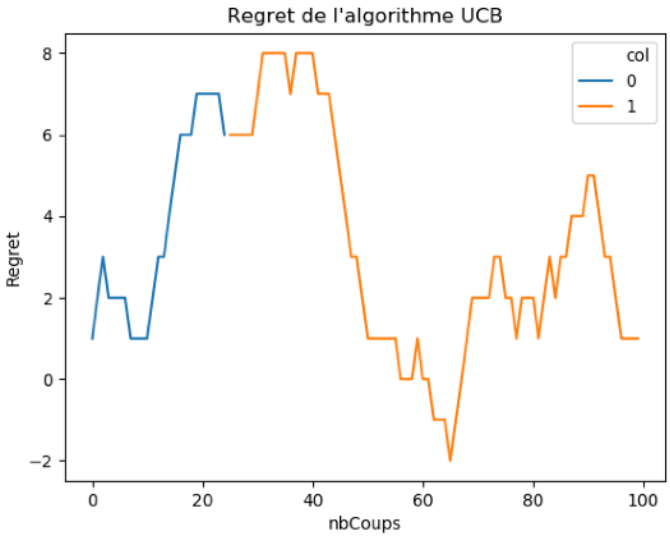
Prenons nbExploration = 10 , voici le graphe :



On constate que l'exploration n'est pas bonne et il en résulte qu'avec le coefficient de confiance déterminée lors de l'exploitation ,

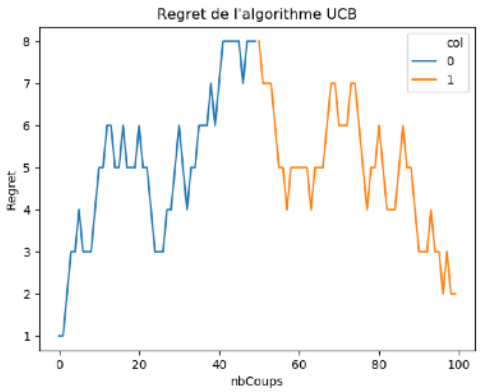
UCB fait plus confiance à un choix de levier qu'il a déjà employé par rapport à un levier qu'il n'a jamais utilisé alors qu'ils sont potentiellement meilleurs que celui choisi par UCB.

En prenant un nombre d'explorations plus élevés (nbExploration = 25) , on a alors ce graphe:



On constate que lors de l'exploitation , en fonction du gain binaire obtenu pour le levier choisi , on a un choix de levier qui va varier. En effet , les premières décisions de UCB sont les mêmes et mènent à un regret « stable » mais élevé. Puis UCB va « hésiter » en alternant entre 2 leviers qu'il considère comme meilleur à un instant t donné. Le facteur de confiance et le gain binaire à l'issu du choix du levier affecte la probabilité estimée. Il en résulte que UCB hésite et n'est pas consistant dans le choix de son levier. Cependant avec chaque itération , les probabilités estimées sont raffinées et donc il en résulte un regret plus bas comme dans l'algorithme glouton.

Prenons nbExploration = 50 , nous avons alors ce graphe:



On constate que l'exploitation suit approximativement les décisions de l'exploration car ses décisions sont dictées par les résultats de l'exploration. Le regret est plus bas et il semble démontrer que le levier choisi est meilleur vers la fin de la représentation graphique.

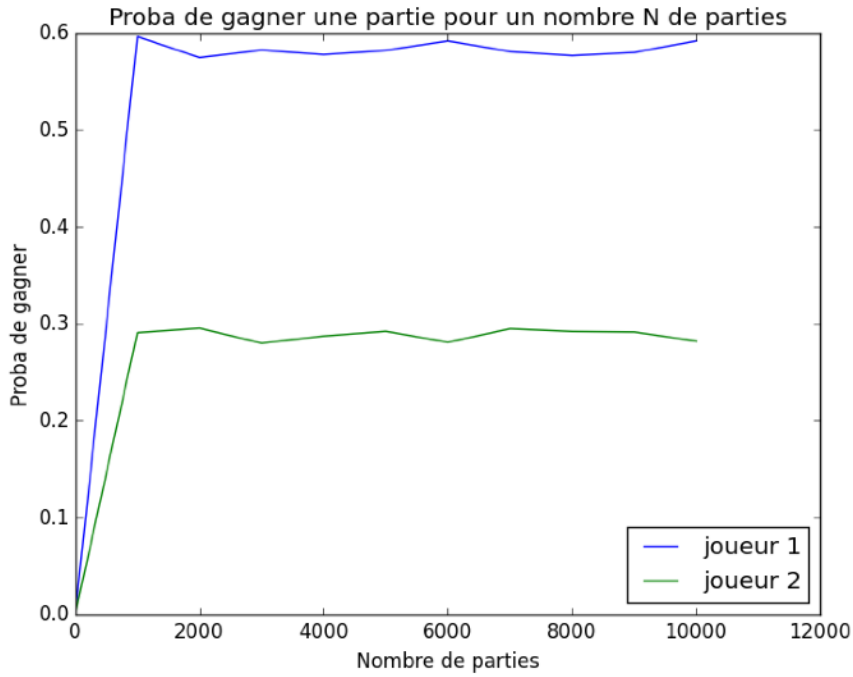
On peut conclure que UCB dépend très fortement de sa phase d'exploration. Si dans la phase d'exploration , nous n'avons pas pris un bon levier (dépend aussi des valeurs des probabilités de gain de la machine) , alors il y a aura une très grande probabilité que la phase d'exploitation soit mauvaise également.

En conclusion de cet exercice , nous pouvons affirmer les faits suivants :

- L'algorithme aléatoire ne retourne pas le bon choix de façon consistante , donc il est logiquement omis des comparaisons
- L'algorithme greedy et UCB sont de bons algorithmes mais ils dépendent fortement de la phase d'exploration. Si la phase d'exploration n'a pas exploité plusieurs fois chacun des leviers/possibilités , il y a de grande chance que le regret soit élevé. On a alors une représentation graphique très variable et les probabilités de la machine sont générées en plus aléatoirement avec random.random(). L'expérience issu de l'exploration variera et par conséquent l'exploitation aussi. Le nombre de coups dédiés à l'exploration permet de trouver de meilleures probabilités mais il y a toujours le risque de tomber sur de mauvais levier et de gagner avec par « chance ».
- L'algorithme epsilon greedy est un bon candidat pour notre problème , il alterne entre exploration et exploitation et permet un choix plus logique et plus proche de la réalité malgré des probabilités pour la machine générées aléatoirement. On le voit assez facilement avec epsilon = 0.1 . Les mises à jour à chaque coup sont mieux prises en compte lors de l'exploitation que dans les algorithmes UCB et greedy et permettent un meilleur résultat. Il faudra cependant faire attention à ne pas prendre un epsilon trop grand car dès lors que l'on effectue beaucoup d'exploration de façon consécutive , le regret est alors moins bon. Epsilon = 0.1 permet d'alterner de façon optimale entre greedy et exploration.

Exercice 2 : Implémentation des agents pour le jeu du morpion

- Joueur1 Aléatoire vs Joueur2 Aléatoire



On remarque le joueur qui joue en 1er a une plus grande chance en moyenne de gagner la partie car il a plus de chance de sélectionner la case centrale qui augmente grandement les chances d'avoir un alignement.

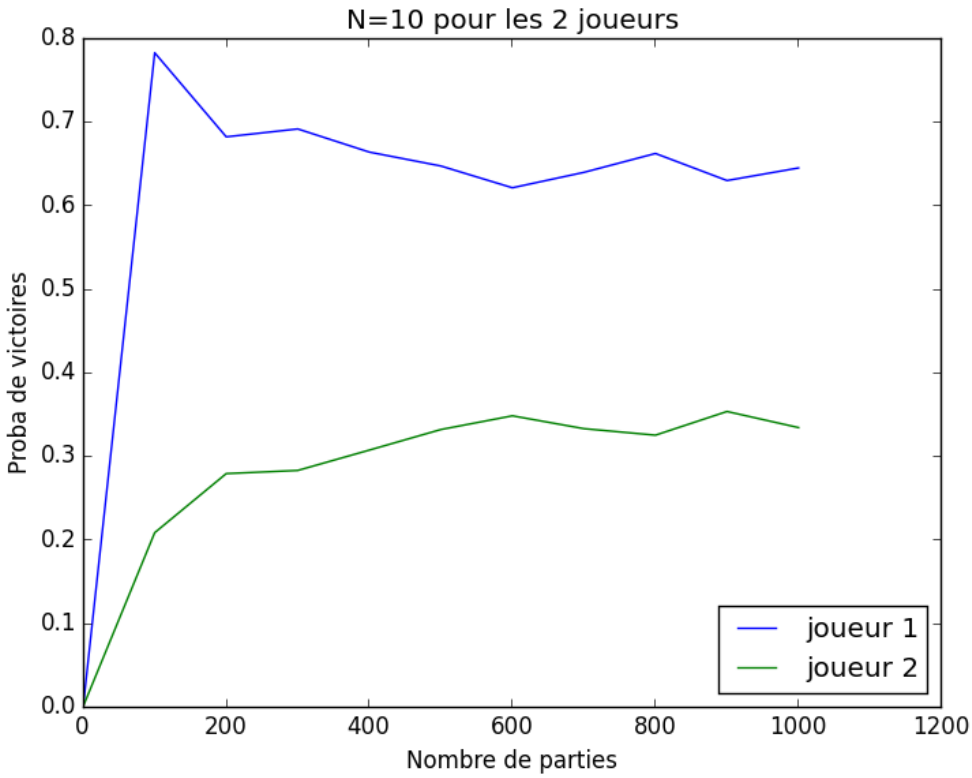
On définit une variable aléatoire X tel que X = 0 désigne un échec et X=1 désigne une victoire.

La loi de probabilité suivie est celle de Bernoulli de paramètre p où p est le rapport entre le nombre de victoires et le nombre de parties jouées. Prenons p = 0.6 qui correspond à la

probabilité de gagner du joueur 1 évaluée sur 10000 parties. On a alors l'espérance de victoire qui est p = 0.6 et sa variance est p(1-p) = 0.24.
Par un raisonnement équivalent pour le joueur 2 , on constate que sa probabilité de gagner (le paramètre p) est beaucoup plus faible par rapport à celle du joueur 1.
On voit que sur 10000 parties , son espérance de gain est p = 0.3 approximativement et sa variance est p(1-p) = 0.21.

Dans les 2 cas , la variance est faible, ce qui signifie que dans la majeure partie de nos simulations , nous aurons une simulation assez souvent proche de celle présentée ci-dessus où le joueur 1 gagnera. Ce qui est logique car nous avons affaire ici à deux joueurs aléatoires qui auront des nombres de victoires et de défaites assez similaire.

- Joueur1 Monte-Carlo (N=10) vs Joueur2 Monte-Carlo (N=10)



Un joueur Monte-Carlo a tendance à prendre la case centrale lors de son 1er coup si celle-ci est encore disponible.

Pour N=10 itérations pour chaque décision à chaque coup, le 1er joueur est toujours plus avantagé que le 2ème grâce donc à cette case centrale.

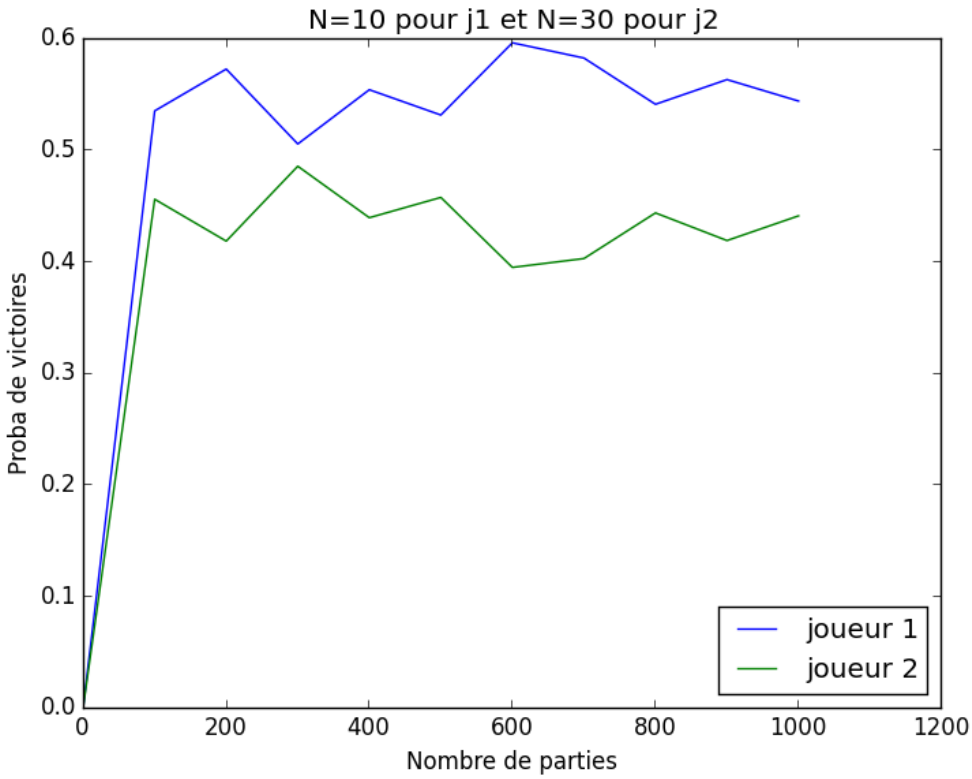
La probabilité que le Joueur1 remporte la partie est toujours plus élevée que celle du Joueur2.

L'espérance de gain pour le joueur 1 est ici approximativement p = 0.7. On a une variance égale à p(1-p) = 0.21

L'espérance de gain pour le joueur 2 est approximativement p = 0.3. On a une variance égale à 0.21.

On voit au travers de la variance calculée et de la représentation graphique que de futurs simulations mèneront souvent à la victoire du joueur 1.

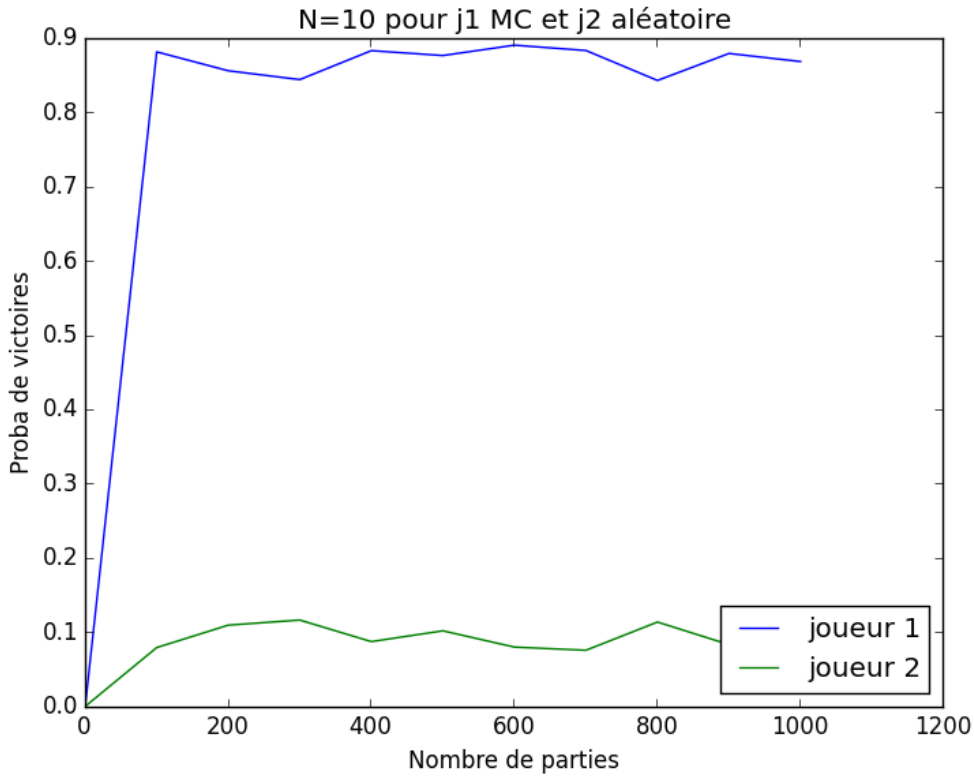
- Joueur1 Monte-Carlo (N=10) vs Joueur2 Monte-Carlo (N=30)



En donnant un N au J2 3 fois supérieur à celui du J1, l'étau se resserre et la probabilité du J2 de gagner augmente énormément malgré qu'il commence la partie après le J1.

On peut en conclure qu'avec un grand N, la fonction de décision du prochain coup devient meilleur.

- Joueur1 Monte-Carlo (N=10) vs Joueur2 Aléatoire

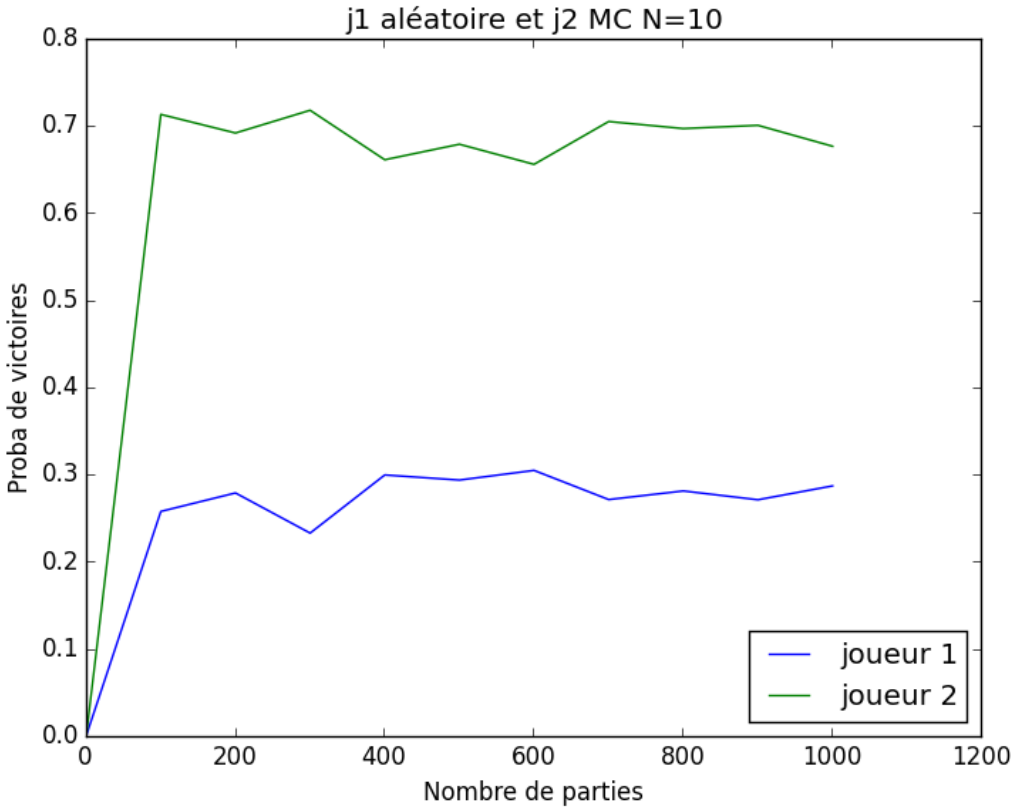


On peut voir que le Joueur MC a une probabilité écrasante de victoire sur le Joueur Aléatoire.

En plus de jouer ses coups "intelligemment", c'est aussi le 1er à commencer la partie, ce qui lui procure donc cette forte chance de victoire.

On peut également en déduire qu'avec un N plus grand, le Joueur Aléatoire aurait une probabilité encore plus proche de 0.

- Joueur1 Aléatoire vs Joueur2 Monte-Carlo (N=10)



Même en procurant l'avantage du 1er coup au Joueur Aléatoire, celui-ci gagne en moyenne 2 à 3 fois moins que son adversaire.

Exercice 3: Implémentation de l'UCT pour le jeu du Morpion

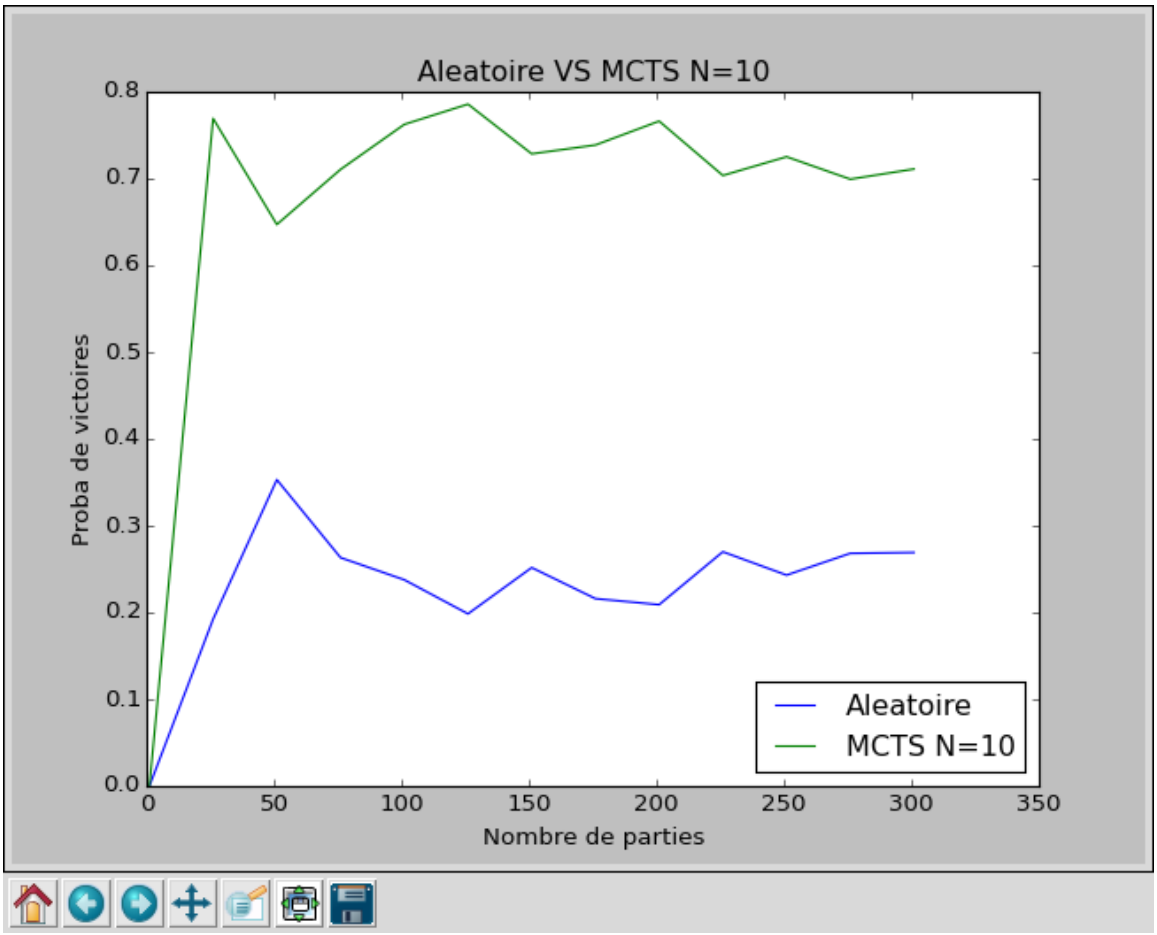
Pour implémenter l'algorithme UCT , nous avons eu besoin de créer une classe Node:

- Elle contient les informations sur le nombre de victoires et le nombre de visites du noeud afin de connaitre la probabilité de réussite du coup.
- Elle contient une variable contenant le coup joué qui a permis d'arriver à cet état . (un tuple)
- Elle contient également l'état du morpion à un instant t donné.
- Elle contient la liste des actions possibles à partir de l'état contenu dans l'objet Node avec la méthode de State : state.get_actions()
- Elle a une liste de ses enfants (children) afin de pouvoir exploiter l'information.
- Nous avons également défini une méthode UCTSelect(self) qui permet d'évaluer un coup en fonction de sa probabilité de gagner mais également du nombre de fois qu'il a parcouru le noeud.

- La méthode add_children(…) dépend de la liste des actions possibles du noeud. Elle crée un nouveau noeud basé sur la liste des actions_possibles.

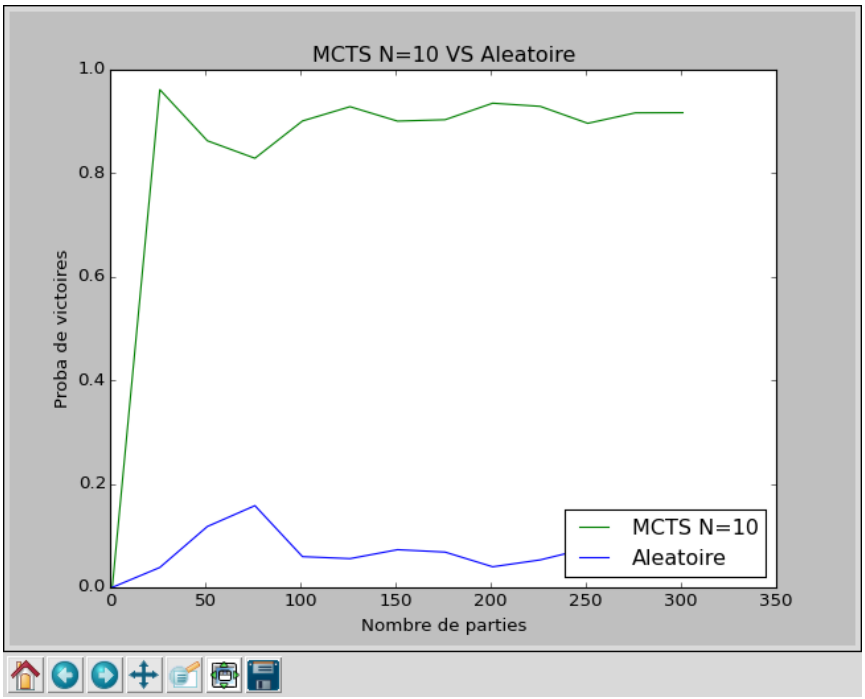
Une fois la classe Node implémentée , nous avons défini un nombre d'itération N (nbIteration dans le code) avant de jouer un coup sur le morpion. Ainsi , l'agent MCTS aurait une meilleure estimation des coups à jouer.

Prenons N = 10 avec le joueur 1 : l'agent Aléatoire et joueur 2 : l'agent MCTS (UCT).



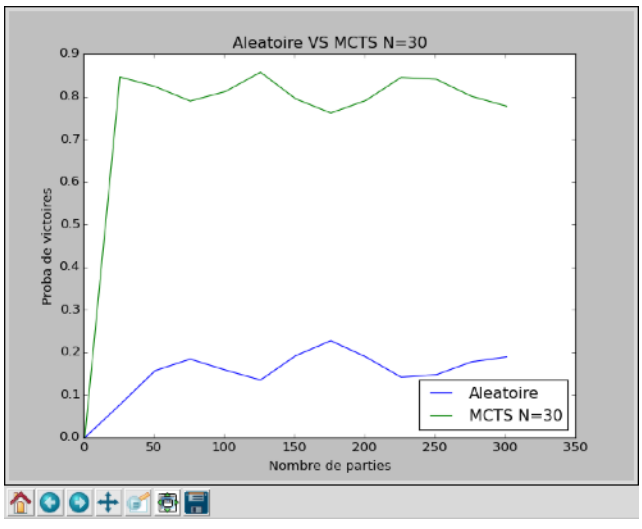
On constate que MCTS a une très grande chance de gagner malgré le départ en joueur 2, cela est dû au fait que l'agent aléatoire peut prendre la case au centre du morpion et peut également jouer de nombreux coups incohérents qui ne sont pas forcément pris en compte par UCT.

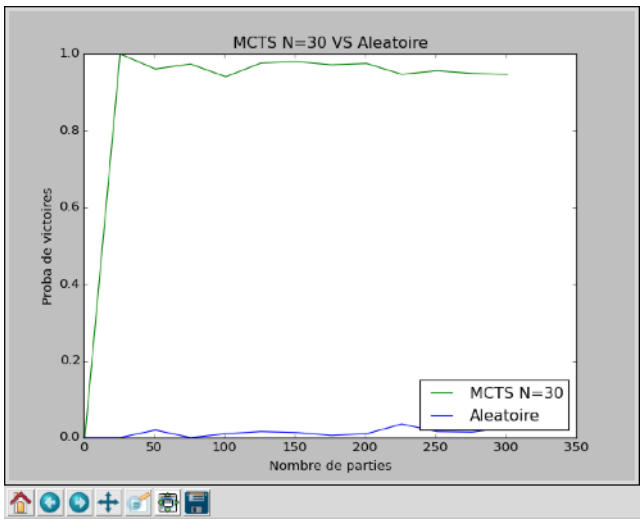
Lorsqu'il est joueur 1 , on a un contraste assez important :



Lorsqu'on effectue les simulations , il prendra toujours la case centrale et il augmentera fortement ses chances de gagner car elle est optimale. Il lui arrive cependant de perdre ou de faire des égalités lorsque les coups de l'agent aléatoire sont incohérents.

En faisant varier le nombre d'itérations à N = 30 , on aura une meilleur approximation des probabilités de victoire mais il ne gagne pas avec une probabilité de 1.



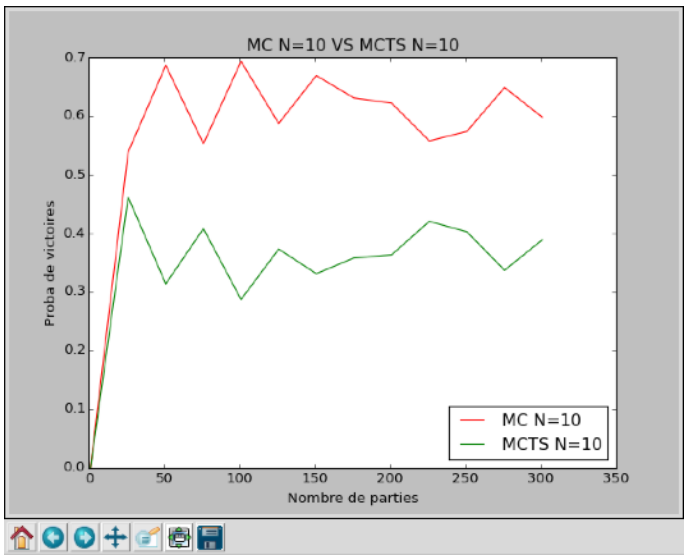


Comparons le maintenant avec le joueur de Monte Carlo défini dans l'exercice 2:

Faisons varier le nombre d'itération avant de jouer un coup sur le véritable morpion :

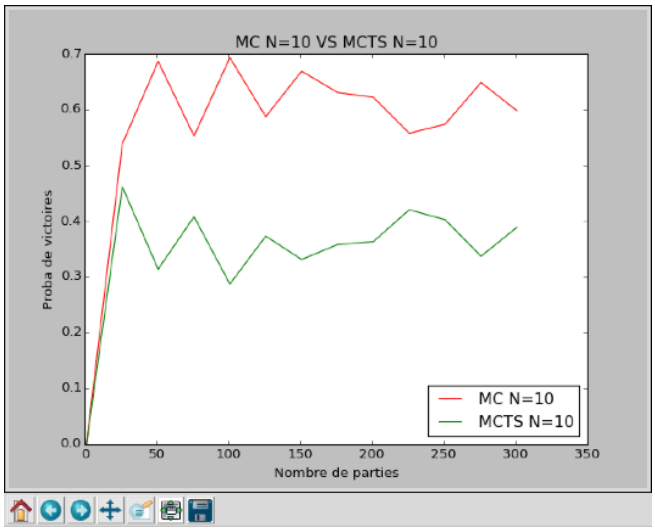
N = 10

On a alors la courbe suivante lorsque MCTS est en 2ème position:

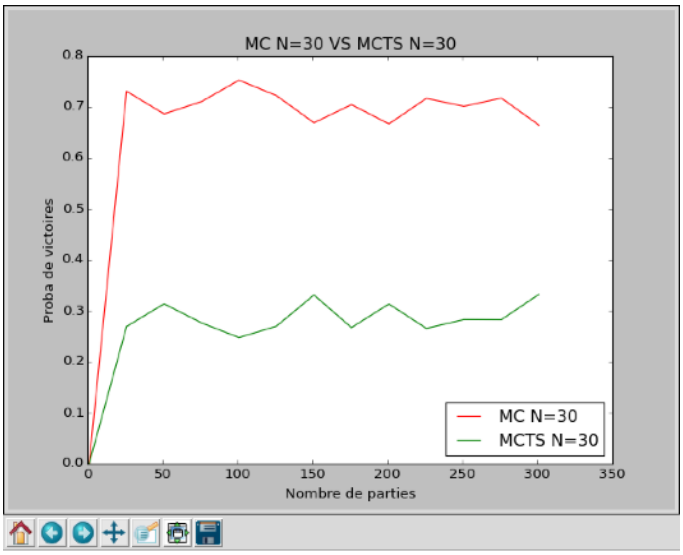


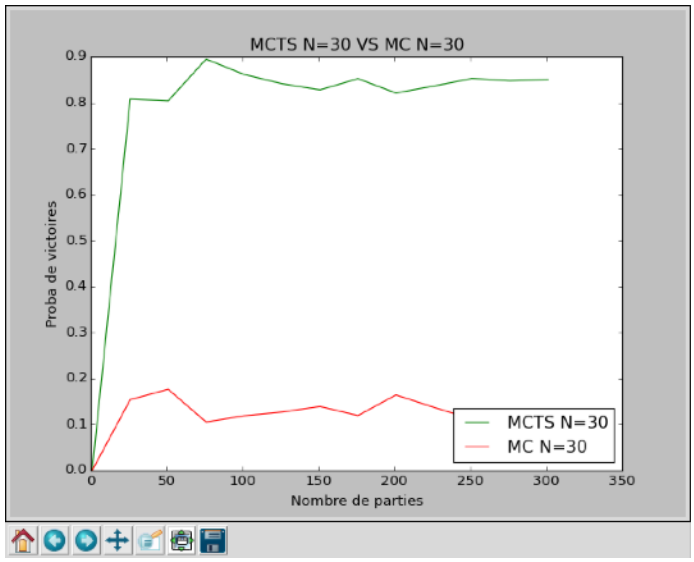
On constate que le joueur de Monte Carlo pose plus de problème au joueur UCT , surtout si il est désavantagé par son départ en 2ème position. (Le joueur de Monte Carlo prendra en général la case centrale)

On voit que la tendance s'inverse lorsque UCT est le 1er joueur pour la même raison que Monte Carlo.



En mettant N = 30 , on constate que les écarts augmentent drastiquement mais les résultats seront les même:

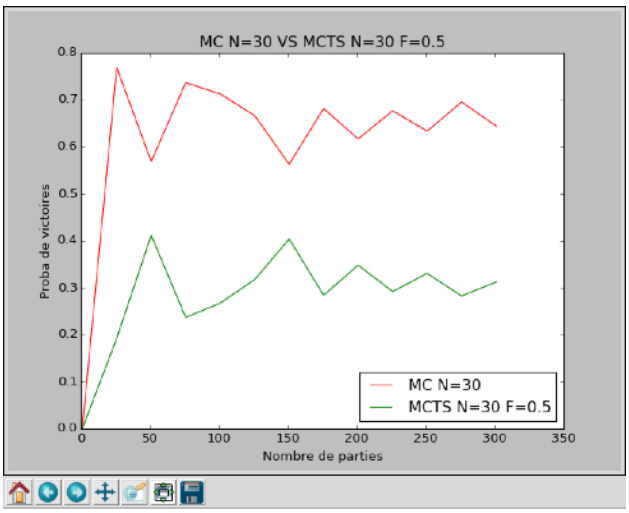




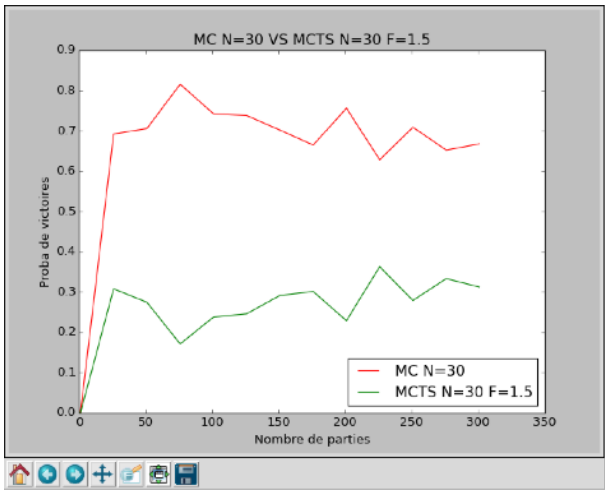
On constate que les nombres d'itération dédiées à l'exploration permettent de renforcer les gains du joueur 1 de façon générale et que UCT perd en général en tant que joueur 2.

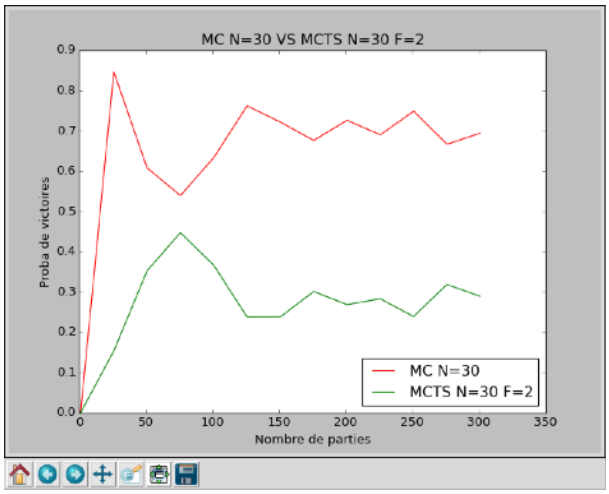
Nous allons maintenant voir si le facteur multiplicatif dans UCT permet d'augmenter les chances de gagner de l'algorithme UCT (joueur 2) tout en fixant un N afin de pouvoir comparer les performances de UCT.

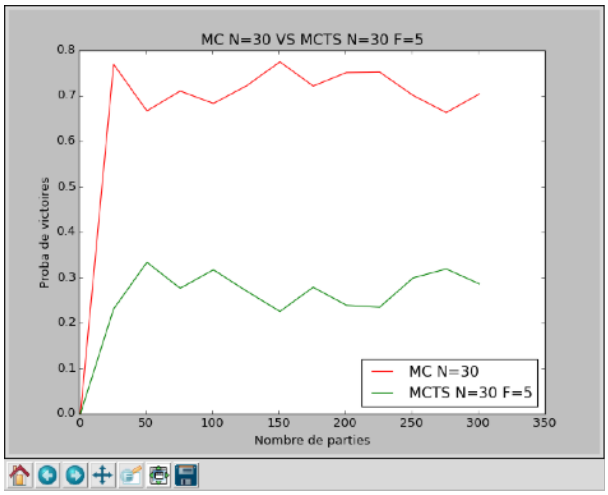
Pour N = 30 et F le facteur multiplicatif fixé à 0.5 , on voit que les gains sont plus intéressants pour UCT. Cela sous entendrait que nous devrions mettre le priorité sur l'exploitation et moins sur l'exploration de façon générale afin de maximiser les chances de réussite de UCT en tant que joueur 2.



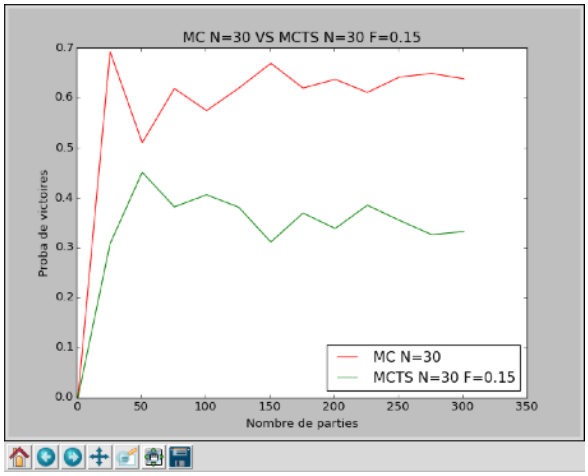
Pour des facteurs multiplicatifs élevés , F = {1.5 , 2 , 5} , nous obtenons de moins bon résultats par rapport à F = 0.5.







En prenant des F plus bas tel que F = 0.15 , on constate que les probabilités de gagner de UCT sont plus élevés :



On peut donc conclure que UCT fonctionnera mieux en mettant la priorité sur l'exploitation des probabilités estimées et en prenant légèrement en compte le coefficient de véracité.