2023/2024

SORBONNE UNIVERSITÉ

Rapport du Mini-Projet 1 de Statistiques et informatique

Puissance 4 (Exploration/Exploitation)

- Groupe : 3
- ITCHIR Maissa
- 21121949

Introduction:

Les jeux de hasard sont caractérisés par un déroulement influence partiellement ou totalement par la chance .Parmi eux ,se trouve le célèbre jeu Puissance, objet d'étude de notre projet Cependant ,nous ne nous contentons pas de compter uniquement sur la chance nous explorons diverses stratégies pour augmenter nos chances de victoire de maniere intelligente.

Dans ce projet nous examinons 3 stratégies pour prendre les meilleures décisions dans le jeu Puissance 4 : un joueur aléatoire, un joueur utilisant la méthode de Monte Carlo et une approche basée sur les arbres d'exploration ainsi que la méthode UTC .L'objectif est d'évaluer leur efficacité dans l'obtention de taux de victoires significativement plus élevé .A travers ces différentes approches nous cherchons à démontrer la valeur de l'intelligence artificielle dans la prise de décisions même dans les jeux qui intègrent une composante aléatoire.

Les règles de jeu:

Puissance 4 est un jeu de strategie sui se joue sur une grille verticle de è colonnes et 6lignes.Deux joueurs s'affrontent ,l'un utilisant des jetons de couleur rouge et l'autre jaune.

- Les joueurs jouent tour à tour en plaçant un jeton de leur couleur dans l'une des colonnes
- le jeton tombe vers le bas de la colonne et occupe la première case vide en partant du bas
- le but du jeu est d'aligner quatre jetons de s propre couleur horizontalement, verticalement ou en diagonale
- le jeu se termine dès qu'il y ait de vainqueur , ce qui aboutit un match nul
- le joueur qui réussit à aligner quatre jetons remporte la partie

Description générale du code:

Mon projet se compose de deux dossiers. Le premier dossier 'combinatoire' contient un fichier regroupant toutes les fonctions relatives aux parties 1, 2 et 4, ainsi qu'une dernière fonction de simulation. Le second dossier, "bandits_manchots", contient un fichier où j'ai implémenté quatre algorithmes : aléatoire, greedy, epsilon-greedy et UCB. Ce dossier contient également un fichier "data.py" pour le traitement des données.

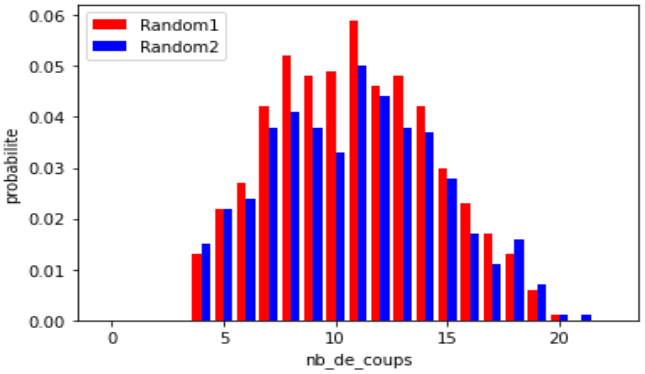
Partie 1:Combinatoire du puissance 4

Dans un premier temps, on implémente plusieurs classes qui définissent les joueurs et les mécanismes du jeu (Player, Game, RandomPlayer)

- La classe Player : représente un joueur, pour qu'un joueur puisse jouer une partie avec un autre joueur
- La classe Game : dans cette classe on implémente le moteur de jeu . Elle gere le plateau ,vérifie les conditions de victoire et gère le déroulement du jeu

  - **`__init__(self, num_line, num_column):`** un constructeur qui initialise le jeu en définissant le nombre de lignes et de colonnes du plateau , il crée également le tableau de jeu
  - **`is completed(self):`** une méthode qui vérifie si le plateau est entièrement rempli
  - **`reset(self, players) :`** une méthode qui réinitialiser le plateau de jeu , elle est utilisée pour commencer une nouvelle partie
  - **`get_win_board(self):`** une méthode qui fournit une liste des configurations gagnantes potentielles que le jeu vérifie pour déterminer s'il y'a une victoire après chaque coup.
  - **`has_won(self) :`** une méthode qui parcourt les configurations gagnantes possibles et vérifie si l'une d'entre elles est réalisée sur le plateau actuel
  - **`play(self, x, player: Player):`** cette méthode permet à un joueur de jouer un coup en plaçant sa pièce dans la colonne spécifiée x , elle vérifie si la colonne est valide et place la pièce si c'est le cas.
  - **`get_available_columns(self):`** une méthode qui retourne une liste de colonnes disponibles ou un joueur peut placer sa pièce . Cela permet au joueur humain de prendre des décisions informées
  - **is_finished(self):** cette méthode vérifie si le jeu est terminé , s'il ya un vainqueure ou si le plateau est complet
  - **run(self, *players: Player, showChessBoard=True, showResult=True, restart=True):** Cette méthode gère le déroulement d'une partie;elle appelle les méthodes appropriées pour faire jouer chaque joueur.

Cela nous permet de faire une simulation de parties entre deux joueurs agissent de manière aléatoire dans le but d'analyser la distribution du nombre de coups nécessaires avant qu'un joueur ne remporte la partie .En effectuant un total de 1000 partie on aura le graphique suivant:



Pour un échantillon de 1000 parties, on observe les probabilités de victoire suivantes pour deux joueurs aléatoires:

En 5 coups : les probabilités de victoire pour les joueurs 1 et 2 sont toutes les deux faibles à environ 0.02. Cela suggère que dans un petit échantillon de parties il est rare qu'un joueur gagne en 5 coups

En 10 coups : Les probabilités de victoire augmentent pour les deux joueurs avec une légère avance pour le joueur 1 .Cependant ,les deux probabilités restent proches à environ 0.04 chacune

En coups 15 : Les probabilités de victoire diminuent pour les deux joueurs

En coups 20 : les probabilités de victoire continuent de diminuer et sont maintenant inférieures à 0.01 pour les deux joueurs

D'apres les resultats precedents , on déduit que le jeu est dépendant du hasard lorsque les deux joueurs jouent d'une manière aléatoire, le tirage aléatoire peut favoriser parfois l'un ou l'autre des joueurs l'autre et d' un autre côté il faut noter que ces résultats sont basées sur un échantillon précis et les probabilités changent si on choisit un autre ensemble de parties .Cela signifie que ce jeu avec des joueurs aléatoires repose fortement sur la chance.

Réponse de la question 4 : la distribution qu'on observe suit une loi poisson.On peut interpréter le nombre de coups nécessaires pour gagner comme un événement 'rare', car gagner en peu de coups moins probable que de gagner en plus de coups avec λ variant en fonction du nombre de coups.

Réponse de la question 5 :

```
Random1 won [0, 0, 0, 0, 12, 21, 26, 48, 34, 36, 45, 52, 62, 35, 44, 27, 19, 18, 8, 12, 4, 1, 0] in total 504 times
Random2 won [0, 0, 0, 0, 15, 24, 27, 38, 45, 42, 47, 61, 38, 39, 35, 28, 18, 19, 8, 5, 6, 1, 0] in total 496 times
there are  0 ties during the games, and the probability is 0.0
```

A partir de là, On peut calculer la probabilité de la partie nulle en utilisant :Nombres de parties nulles / Nombre de parties totales = 1 / 1000 = 0.001

Partie 2 : Algorithme de Monte-Carlo

Notre stratégie précédente (joueur aléatoire vs joueur aléatoire), les joueurs jouent de manière totalement aléatoire sans aucune considération stratégique , donc les résultats de jeu dépendent entièrement du hasard . Avec la stratégie Monte Carlo ,le joueur Monte Carlo

utilise des simulations pour évaluer les différentes colonnes disponibles à chaque tour; il simule plusieurs jeux aléatoires a partir de chaque etat du jeu pour estimer les probabilités de victoire associées à chaque colonne et enfin il prend des décisions informées basées sur ces estimations de probabilité (une prise de décision plus intelligente).

Implémentation:

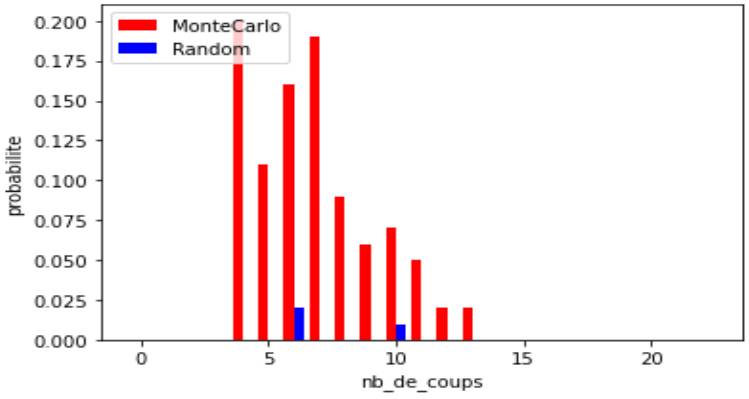
`-__init__(self, signal=1, name="untitled",simulation_times=50):`
le constructeur de la classe initialise un joueur Monte Carlo avec un signal , un nom et un nombre de simulations à effectuer

`-division_array(self, array_mem, array_denom):` Cette méthode prend en entrée deux listes array_mem et array_denom. Elle renvoie une liste où chaque élément est le résultat de la division de l'élément correspondant de array_mem par l'élément correspondant de array_denom. Si le dénominateur est zéro, elle gère ce cas spécifique.

`-getOneStep(self, game):` Cette méthode permet de prendre une décision lors d'un tour de jeu Elle prend en entrée l'objet game qui représente l'état actuel du jeu. Le joueur Monte Carlo va simuler plusieurs parties à partir de l'état actuel et estimer les probabilités de victoire associées à chaque colonne disponible. Il choisira ensuite la colonne avec la probabilité de victoire la plus élevée.

Lors de la simulation entre un joueur aléatoire et un joueur Monte Carlo, On observe que ce dernier à démontrer une performance notablement supérieure par rapport au joueur aléatoire; pour les parties qui se sont conclues en un petit nombre de coups (en coups 5 par exemple) le Monte Carlo montre une probabilité de victoire significative (0.1) tandis que le joueur aléatoire n'a pas réussi à remporter une victoire dans ces situations. Même lorsque la partie s'est prolongée (en coups 7) la probabilité de victoire du joueur aléatoire reste faible (0.025) en comparaison avec le Monte Carlo qui maintient une probabilité de victoire plus élevée(0.15).

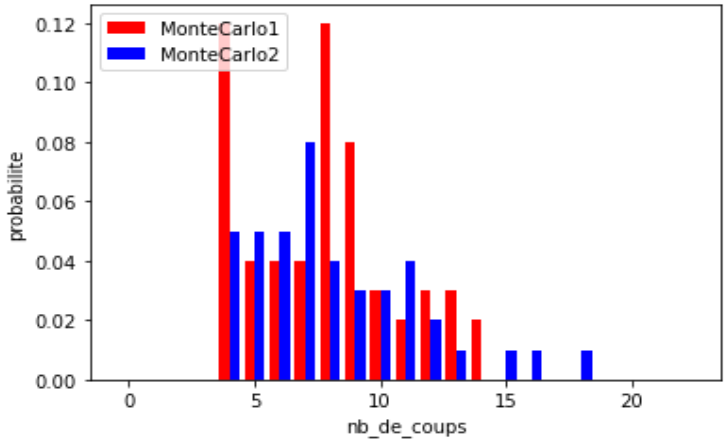
Ces résultats indiquent clairement l'efficacité de l'algorithme Monte Carlo dans la prise de décision par rapport à une approche purement aléatoire.(l'importance de l'intelligence artificielle basée sur la simulation pour des jeux de stratégie comme Puissance 4).




```
MonteCarlo won [0, 0, 0, 0, 20, 11, 16, 19, 9, 6, 7, 5, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0] in total 97 times
Random won [0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] in total 3 times
there are  0 ties during the games, and the probability is 0.0
```

Lors de la simulation entre un joueur Monte Carlo et un autre joueur Monte Carlo ,les deux joueurs utilisent la même stratégie. Le résultat de la partie dépendra des estimations de probabilités générées par chaque joueur à chaque tour. le joueur qui fait des meilleures estimations aura une meilleure chance de gagner, mais le résultat finale reste déterminé par

le hasard des simulations (Cela signifie que même si les deux joueurs utilisent la même stratégie, il peut y avoir des résultats différents en raison de la nature probabiliste de l'algorithme de Monte Carlo) .




```
MonteCarlo1 won [0, 0, 0, 0, 12, 4, 4, 4, 12, 8, 3, 2, 3, 3, 2, 0, 0, 0, 0, 0, 0, 0, 0] in total 57 times
MonteCarlo2 won [0, 0, 0, 0, 5, 5, 5, 8, 4, 3, 3, 4, 2, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0] in total 43 times
there are  0 ties during the games, and the probability is 0.0
```

Partie 3:Bandits-manchots

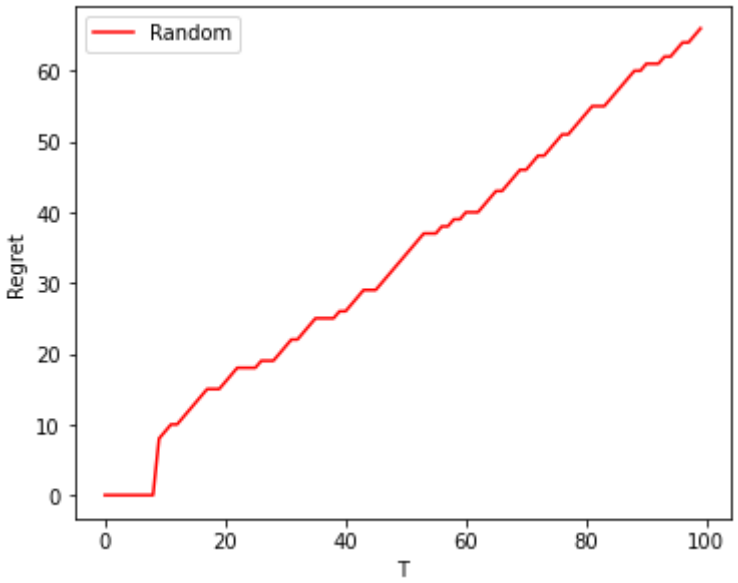
Probleme Exploration/Exploitation

Dans le jeu de machine à sous, appelé "Bandits Manchots", le joueur est confronté à plusieurs leviers, chacun offrant une récompense aléatoire. L'objectif du joueur est de maximiser son gain total, mais il ne dispose pas d'informations préalables sur lequel de ces leviers est le plus rentable. Ainsi, il doit trouver un équilibre subtil entre l'exploration de nouvelles actions et l'exploitation de celles qui semblent être lucratives jusqu'à présent. Les quatre algorithmes que nous allons examiner visent à résoudre ce défi en combinant habilement ces deux stratégies, exploration et exploitation, afin d'optimiser les gains du joueur.

Algorithme Aléatoire:

Dans l'algorithme aléatoire,à chaque étape ,une action (levier) est sélectionné de manière totalement aléatoire parmi toutes les actions possibles .Cela signifie que chaque levier a une chance égale d'être choisi.L'algorithme aléatoire se concentre entièrement sur l'exploration et ne prend pas en compte les résultats des actions passées.

En appliquant l'algorithme aléatoire avec N égal à 10 leviers et 100 itérations, on observe que le regret augmente de manière linéaire au fil du temps. Cela est dû au fait que cet algorithme ne tient pas compte des récompenses obtenues précédemment. Par conséquent ;le regret continue d'augmenter progressivement a mesure que le temps

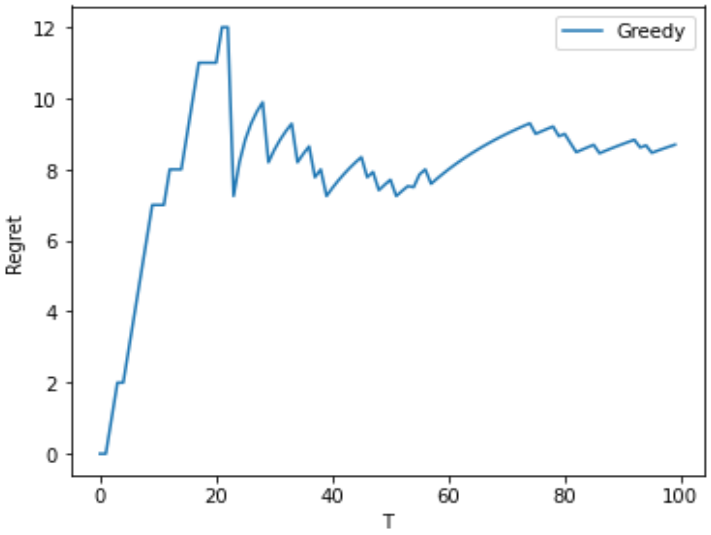


Algorithme Greedy:

L'algorithme Greedy vise à maximiser les gains en choisissant l'action qui semble être la plus rentable. Il prend en compte les résultats des actions précédentes. Au début, pendant un certain nombre d'itérations, il se concentre sur l'exploration. Chaque levier est activé un nombre de fois afin de collecter des données sur leur rendement potentiel. Ensuite, pour

A chaque levier, l'algorithme calcule une estimation du rendement moyen en prenant la moyenne des récompenses obtenues lors de cette phase exploratoire. Enfin, après cette phase d'exploration initiale, l'algorithme se tourne vers l'exploitation. Il choisit toujours le levier dont le rendement estimé est le plus élevé, espérant ainsi maximiser les gains.

En appliquant l'algorithme Greedy avec N égal à 10 leviers et 100 itérations, on observe que le regret augmente rapidement jusqu'à atteindre 15. Ensuite il diminue légèrement pour se stabiliser à 8. Cela indique que l'algorithme a initialement choisi un levier qui semblait être le meilleur mais a ensuite réalisé que ce choix n'était pas optimal. Il a ensuite ajusté sa sélection pour minimiser les pertes ce qui a conduit à une diminution du regret. Finalement;le regret se stabilise à 8 ce qui suggère que l'algorithme a trouvé un bon équilibre entre exploration et exploitation.

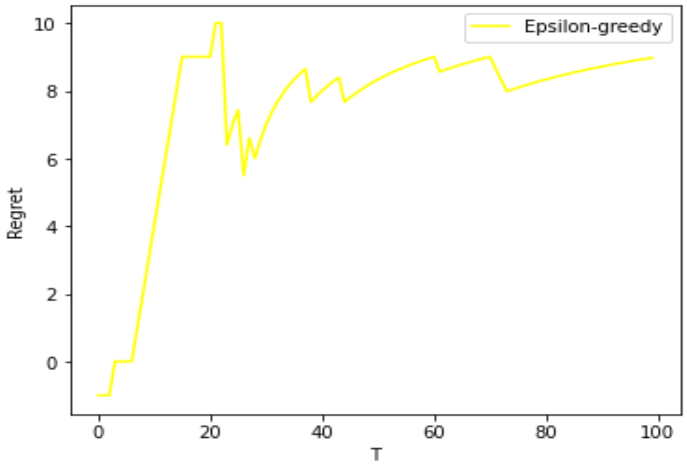


Algorithme ϵ-Greedy:

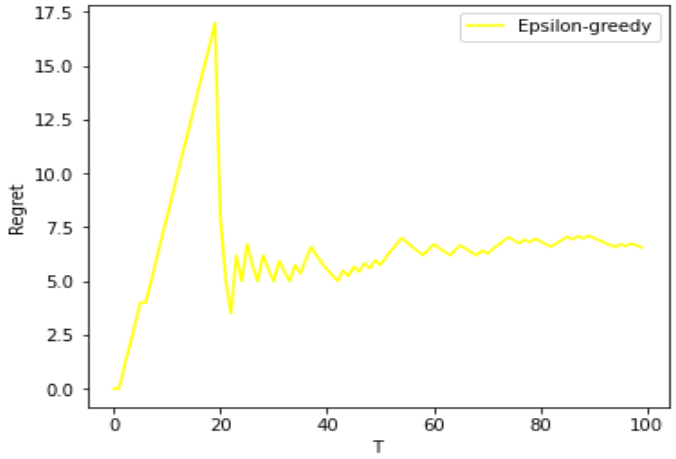
Au début, l'algorithme se concentre sur l'exploration en activant chaque levier un certain nombre de fois pour collecter des données. À chaque itération, il prend une décision en sélectionnant une action parmi toutes les actions possibles. Ensuite, il explore avec une probabilité ϵ. Cela signifie qu'il choisit une action au hasard parmi toutes les actions disponibles. En parallèle, il applique l'algorithme Greedy avec une probabilité 1−ϵ. Cela implique qu'il opte pour l'action qui a montré les meilleurs résultats jusqu'à présent en termes de rendement estimé. Ainsi, l'algorithme ϵ-greedy combine de manière astucieuse l'exploration et l'exploitation pour maximiser les gains.

En appliquant l'algorithme Greedy avec N égal à 10 leviers et 100 itérations, avec un epsilon de 0.1, on note une évolution particulière de regret .initialement ,le regret augmente rapidement jusqu'à atteindre la valeur de 10 lorsque t=20.Cela indique que l'algorithme a commencé par explorer les leviers ,ce qui lui permis de découvrir rapidement les récompenses associées à certains d'entre eux. Ensuite le regret diminue à 8 ce qui signifie que l'algorithme a commencé à exploiter le levier qu'il estimait être le plus rentable. A t=40 et t=60 on observe une légère augmentation du regret ce qui indique que l'algorithme a réévalué ses choix à ces moments-la.

Avec une valeur d'epsilon de 0.1 ,l'algorithme a une probabilité de 10 d'explorer et une probabilité de 90 d'exploiter à chaque itération.(pour un epsilon égal 0.1 on aura le graphique suivant)



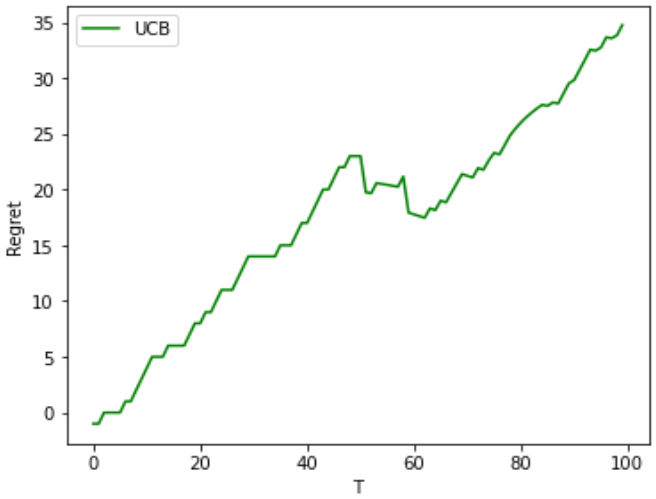
En augmentant la valeur d'epsilon à 0.9 par exemple, l'algorithme aura une probabilité plus élevée d'explorer plutôt que d'exploiter à chaque itération , il aura plus de chances de choisir aléatoirement un levier à chaque itération.Cela signifie qu'il explorera plus souvent et pourra découvrir de nouveaux leviers qui pourraient potentiellement être très rentables.(une exploration plus longue avant de se stabiliser dans l'exploitation de leviers spécifiques, et le regret initiale pourrait être plus élevé)



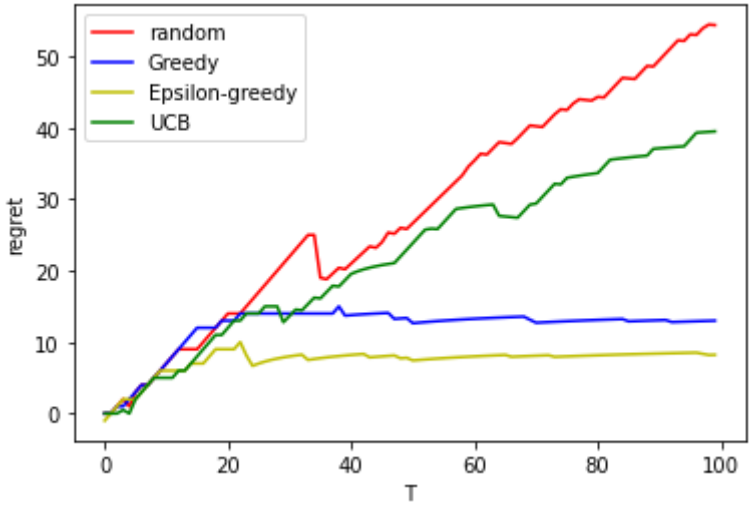
Algorithme UCB:

L'algorithme UCB vise à équilibrer l'exploration(découverte de nouvelles actions) et l'exploitation(choix des actions qui semblent être les meilleures jusqu'à présent), il utilise des bornes supérieures de confiance pour estimer la valeur potentielle de chaque action. A chaque itération l'algorithme calcule un score pour chaque action; ce score est basé sur la moyenne estimée de la récompense de l'action plus une mesure de l'incertitude associée à cette estimation. L'action avec le score le plus élevé(la fonction getArgMax calcule le score pour chaque action en utilisant la formule UCB) est choisie**.**

En appliquant l'algorithme UCB avec N égal à 10 leviers et 100 itérations, pour t=20 on a regret qui vaut 5 ,a ce stade l'algorithme a déjà commencé à faire des choix qui semblent être plus rentables que les choix initiaux. A t= 40 le regret augmente à 15 donc l'algorithme a pris une décision qui s'est avérée moins rentable et à t =100 le regret augmente encore à 35,cela indique que l'algorithme peut continuer à prendre des décisions qui n'étaient pas optimales.



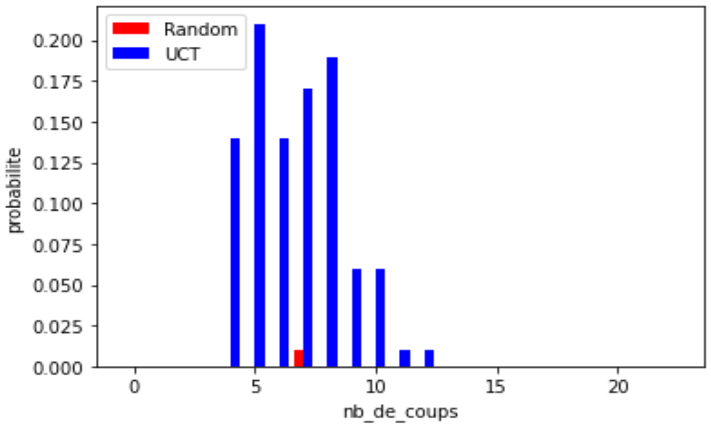
Enfin , dans le problème des bandits manchots ,l'algorithme UCB est le meilleur choix en raison de sa capacité à équilibrer l'exploration et l'exploitation de manière efficace.



Partie 4:Arbre d'exploration et UCT

L'algorithme UCT est une technique d'exploration utilisée pour prendre des décisions dans l'arbre de jeu.Contrairement à une exploration complètement aléatoire, il cherche à prioriser les actions qui semblent les plus prometteuses en termes de victoire tout en continuant à explorer d'autres options . Il utilise le principe de l'algorithme UCB à chaque embranchement permettant ainsi de trouver un équilibre entre exploration et exploitation.

En appliquant l'algorithme UCT contre un joueur aléatoire on observe que la probabilité de victoire du joueur aléatoire en fonction de nombre de coups est extrêmement faible et n'apparaît qu') une seule reprise .Cela prouve l'efficacité de l'algorithme UCT dans le contexte du Puissance 4


Random
UCT
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
probabilite
0
5
10
15
20
nb_de_coups