Exploration/Exploitation et Puissance 4

Mikael Slotboom 21301180
Dorin-Mihai Manea 21302798
Snkar Mam 21300810

12-10-2023

—

Statistique et Informatique (LU3IN005)

—

Nicolas Baskiotis

Table des matières

Introduction 4
Description du code 4
La description du code des parties 1 et 2 : 4
La description du code de la parte 3 : 5
Combinatoire du puissance 4 6
Approche aléatoire : 6
Approche Monte-Carlo : 7
Bandits-manchots 11
Algorithme aléatoire : 11
Algorithme greedy 12
Algorithme ε-greedy : 13
Algorithme UCB 14

Introduction:

L'exploration vs exploitation est un dilemme clé en intelligence artificielle. Il s'agit de déterminer si l'on doit s'appuyer sur ce que l'on sait déjà où explorer de nouvelles options. En publicité, cela peut signifier choisir entre montrer une annonce familière ou essayer quelque chose de nouveau. Dans les jeux comme le GO, il s'agit de choisir entre un coup bien connu et un coup risqué mais potentiellement révolutionnaire. Ce projet se concentre sur l'application de ce dilemme au jeu "Puissance 4" et Bandits-manchots. Il y a quatre parties : 1) Analyse du jeu puissance 4, 2) Utilisation de l'algorithme de Monte-Carlo, 3) Étude des approches d'exploration vs exploitation de Bandits-manchots, et 4) Exploration des algorithmes avancés pour les jeux.

Description du code

- connect_four_statistics/
  - part1_part2/
    - connect4.py
    - distribution_of_moves.py
    - main.py
    - monte_carlo_player.py
    - random_player.py
  - part3/
    - epsilon_greedy_pick.py
    - greedy_pick.py
    - levers.py
    - main.py
    - random_pick.py
    - ucb.py

Le dossier connect_four_statistics est le dossier principal qui contient deux sous-dossiers, part1_part2 et le sous-dossier part3.

La description du code des parties 1 et 2 :

Le **sous-dossier part1_part2** est le dossier qui contient tous les fichiers des part 1 (Combinatoire du puissance 4) et part 2 (Algorithme de Monte-Carlo).

Le fichier **main.py** dans le répertoire part1_part2 est le fichier depuis lequel on exécute le programme de deux premières parts du projet (Combinatoire du puissance 4 et Algorithme de Monte-Carlo).

Le fichier **connect4.py** est une classe modélise le jeu Connect4. Elle gère la logique du jeu, garde une trace de l'état du plateau de jeu, et vérifie les conditions de victoire.

Le fichier **random_player.py** représente une stratégie de joueur simple qui choisit au hasard une colonne pour jouer un jeton. C'est une IA basique pour le jeu Connect4.

Le fichier **distribution_of_moves.py** est une classe conçue pour l'analyse. Elle simule plusieurs jeux de Connect4 et recueille des statistiques sur les résultats, comme le nombre de mouvements qu'il a fallu pour gagner. Elle fournit ensuite une représentation visuelle de ces statistiques.

Le fichier **monte_carlo_player.py** est une stratégie de joueur plus avancée. Elle utilise des simulations de Monte Carlo pour prédire le résultat de différents mouvements et choisit le meilleur mouvement basé sur ces simulations. En essence, c'est une IA plus sophistiquée pour le jeu Connect4.

La description du code de la parte 3 :

Le fichier **levers.py** : Cette classe représente les machines à sous (bandits). Chaque levier a une probabilité associée qui dicte sa chance de donner une récompense.

Le fichier **random_pick.py** : Cette classe représente une stratégie naïve où un joueur choisit simplement un levier au hasard à chaque fois.

Le fichier **greedy_pick.py** : Cette classe représente une stratégie avide où un joueur explore chaque levier un certain nombre de fois, puis choisit toujours le meilleur levier par la suite.

Le fichier **epsilon_greedy_pick.py** : Cette classe représente la stratégie ε-greedy. Ici, le joueur choisit principalement le meilleur levier (basé sur les résultats précédents), mais essaie occasionnellement (avec une probabilité ε) un levier au hasard. Cela équilibre l'exploration et l'exploitation.

Le fichier **ucb.py** : Cette classe représente la stratégie Upper Confidence Bound (UCB). Elle équilibre l'exploration et l'exploitation en fonction de la fréquence à laquelle les leviers ont été choisis et de leurs retours moyens.

Le fichier **main.py** : C'est le script principal. Il crée des instances des classes Levers et des différentes classes de stratégie (RandomPick, GreedyPick, EpsilonGreedyPick, et Ucb). Il joue ensuite chaque stratégie et imprime les résultats.

Combinatoire du puissance 4

Approche aléatoire :

La stratégie du *RandomPlayer* est lorsqu'il est au tour de ce joueur de joue:

1. Il examine le plateau de jeu pour identifier toutes les colonnes qui ont encore de la place pour accueillir un jeton. En d'autres termes, il recherche toutes les colonnes qui ne sont pas encore complètement remplies.
2. Parmi ces colonnes disponibles, il en choisit une au hasard et y place son jeton.
3. Il ne prend pas en compte la stratégie de jeu, les mouvements précédents, ni n'essaie de prévoir les mouvements futurs. Le choix est entièrement basé sur le hasard.

Cette approche est, comme son nom l'indique, complètement aléatoire. Elle ne cherche pas à optimiser les chances de gagner, ni à contrer les mouvements de l'adversaire. C'est la raison pour laquelle elle est considérée comme une IA basique pour le jeu Connect4. Elle est utile pour les tests ou pour servir de point de comparaison avec des stratégies plus avancées, comme celle basée sur la méthode de Monte Carlo.

Cette approche est lorsque deux joueurs aléatoires jouent, la distribution du nombre de coups avant une victoire est affichée sous forme d'histogramme.

Par exemple, lorsque les deux joueurs ont joué 1000 fois, le joueur 1 a gagné 569 fois tandis que le joueur 2 a gagné seulement 428 fois, ils ont eu 3 matchs nuls, comme dans la figure 1.

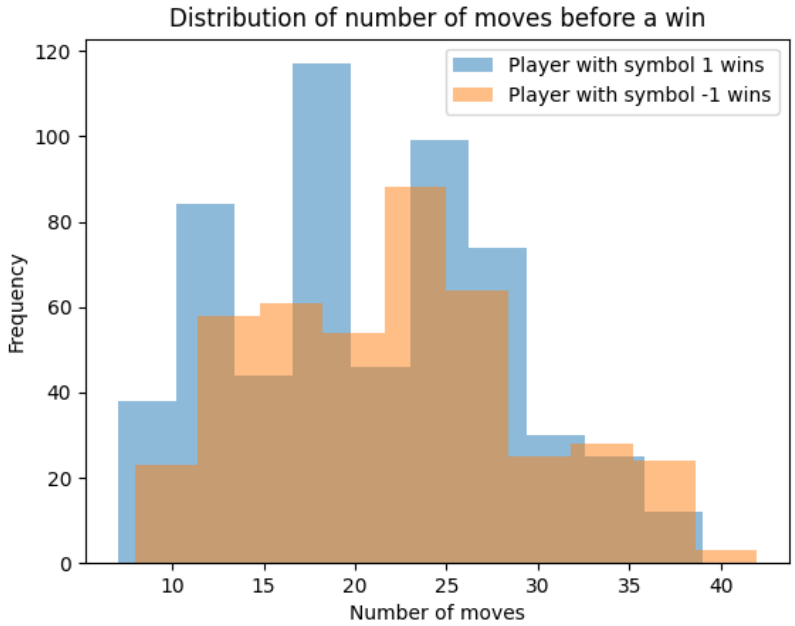

Figure 1 : La distribution des joueurs aléatoires (plateau 6x7)

La distribution ressemble à une distribution normale (loi normale), car elle est basée sur de nombreux essais aléatoires.

Concernant la différence entre le premier et le deuxième joueur, pour la plupart de temps on peut observer que le premier joueur ait en fait un léger avantage, car il commence toujours le joue.

Par rapport a la probabilité des matchs nuls, le pourcentage est toujours alentour 0.3 %. Parmi nos expériences, on a observé 2-4 matchs nuls sur 1000.

Théoriquement, le nombre maximal de parties différentes est égal à 7^{42} (dans un plateau de 6x7), car chaque colonne peut être choisie à chaque coup. Pour approximer ce nombre, on pourrait simuler un grand nombre de parties avec des joueurs aléatoires et compter combien de configurations uniques apparaissent.

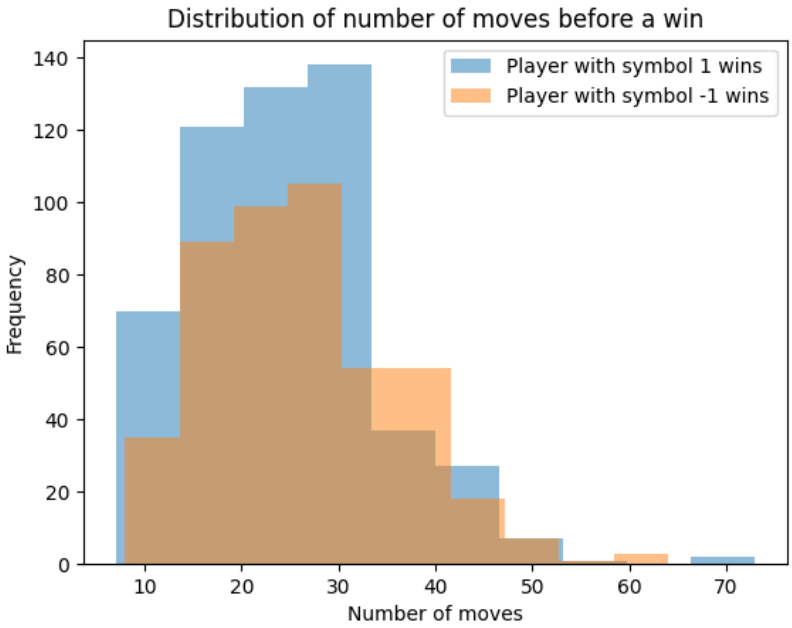

Figure 2 : La distribution des joueurs aléatoires (plateau 9x10)

De plus, si on augmente les dimensions du plateau, par exemple à 9x10, les chances de gagner pour jouer 1 augment aussi, tandis que la probabilité d'un match nul est réduite presque à 0. Ça se passe car il y a plus de façons pour les joueurs de gagner le jeu.

Dans ce cas-ci, la borne théorique pour le nombre maximal de parties différentes devient égale à 9^{90} (dans un plateau de 9x10), car chaque colonne peut être choisie à chaque coup.

Approche Monte-Carlo :

L'approche de Monte Carlo est une méthode de simulation utilisée pour estimer des probabilités complexes en se basant sur un grand nombre d'échantillonnages aléatoires.

Pour chaque coup possible sur le plateau de jeu actuel, le joueur *MonteCarloPlayer* simule un certain nombre de parties (défini par simulation_amount) en jouant ce coup. Pendant ces simulations, l'adversaire est représenté par un *RandomPlayer*. Cela signifie que, pour chaque simulation, après le coup initial de *MonteCarloPlayer*, l'adversaire (et le joueur lui-même pour ses coups suivants) joue au hasard. Les résultats de toutes les simulations pour un coup donné sont moyennés pour obtenir une estimation de la "qualité" de ce coup.

Une fois que toutes les simulations ont été jouées pour tous les coups possibles, le joueur *MonteCarloPlayer* choisit le coup avec la meilleure moyenne, c'est-à-dire celui qui, selon les simulations, a la plus grande probabilité de mener à une victoire.

Cette approche de Monte Carlo est plus avancée que la simple approche aléatoire, car elle tente d'évaluer les conséquences futures de chaque coup possible et choisit le coup qui semble offrir les meilleures chances de victoire à long terme. Cependant, elle est toujours basée sur de nombreuses simulations avec des joueurs aléatoires, donc elle n'est pas parfaitement précise.

On peut constater qu'en augmentant le paramètre *simulation_amount*, qui représente le nombre de simulations que le joueur effectuera pour chaque coup possible lorsqu'il décide dans quelle colonne jouer, l'approche Monte Carlo devient de mieux en mieux par rapport à l'approche complètement aléatoire.

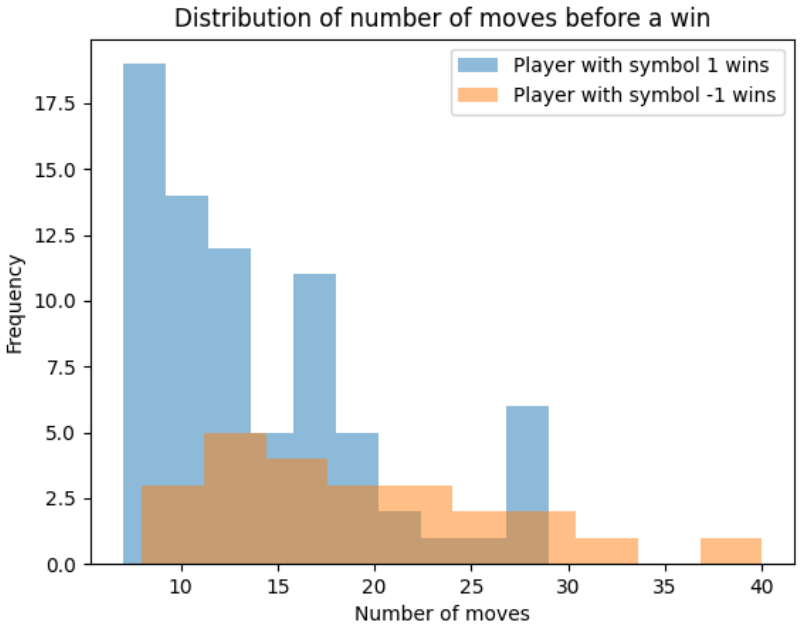

Figure 3 : Monte Carlo vs Random; jouer 1 a gagné 76 fois et jouer 2 a gagné 24 fois de 100 matchs
simulation_amount = 1

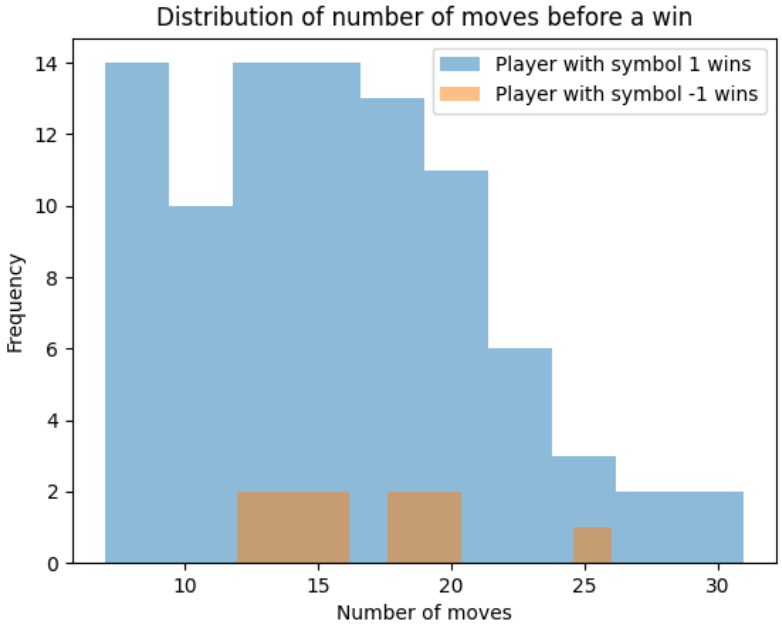

Figure 4 : Monte Carlo vs Random; jouer 1 a gagné 89 fois et jouer 2 a gagné 11 fois de 100 matchs
simulation_amount = 2

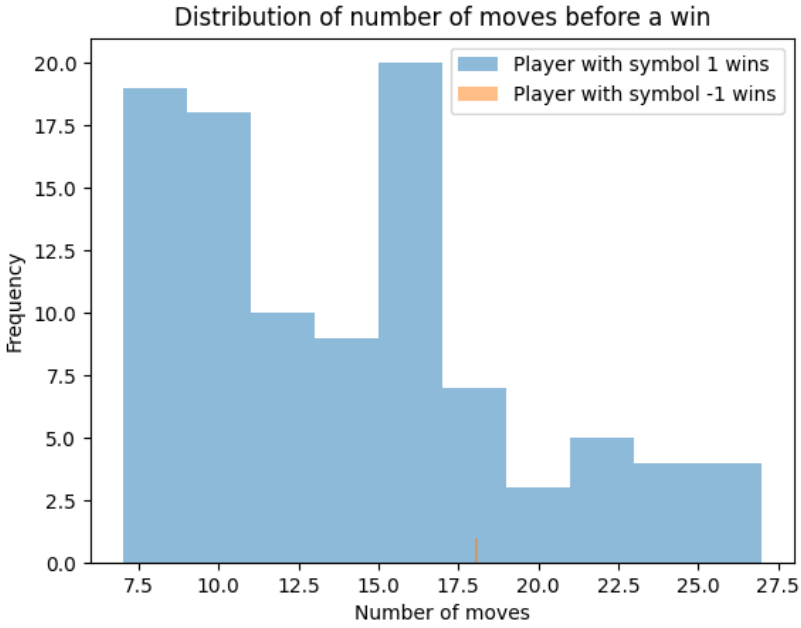

Figure 5 : Monte Carlo vs Random; jouer 1 a gagné 99 fois et jouer 2 a gagné 1 foi de 100 matchs
simulation_amount = 10

En mettant cette fois deux joueurs de type de Monte Carlo l'un contre l'autre, on observe que le premier joueur a encoure un avantage par rapport au jouer 2 lorsqu'ils ont la même valeur pour le paramètre *simulation_amount*. L'expérience a été répétée plusieurs foi, en changeant aussi *simulation_amount* pour bien valider l'argument.

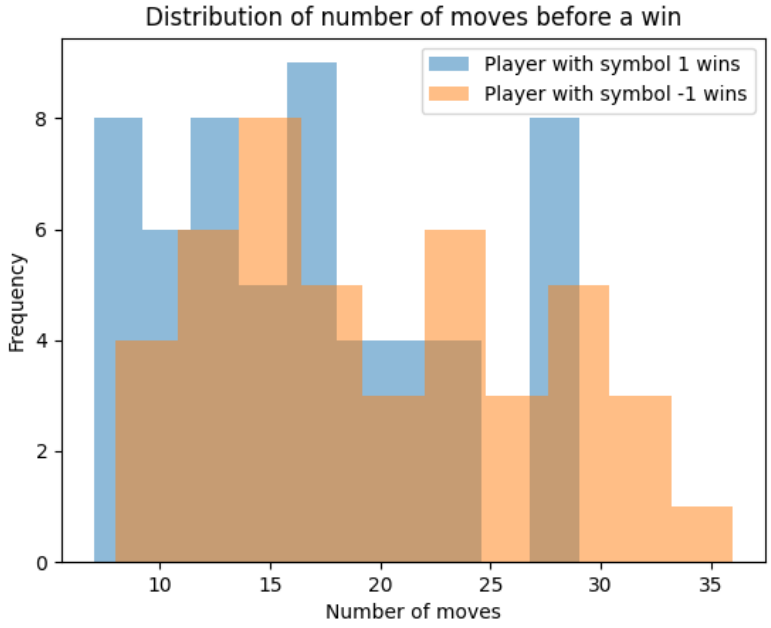

Figure 6 : Monte Carlo vs Monte Carlo; jouer 1 a gagné 56 fois et jouer 2 a gagné 44 foi de 100 matchs *simulation_amount* = 3

Par rapport à la valeur de *simulation_amount* , le jouer qui a la valeur la plut haute va gagner toujours peu import si il commence en premier ou non. Par exemple, on a fait exécuter le code en donnant 1 comme valeur de paramètre *simulation_amount* du premier jouer et 2 comme valeur de paramètre *simulation_amount* du deuxième jouer, on a remarqué que le jouer 2 a gagné le match même s'il a commencé la joue après le jouer 1. Ça a été le même cas pour les valeurs 3 et 5 pour joueur 1 et jouer 2 respectivement.

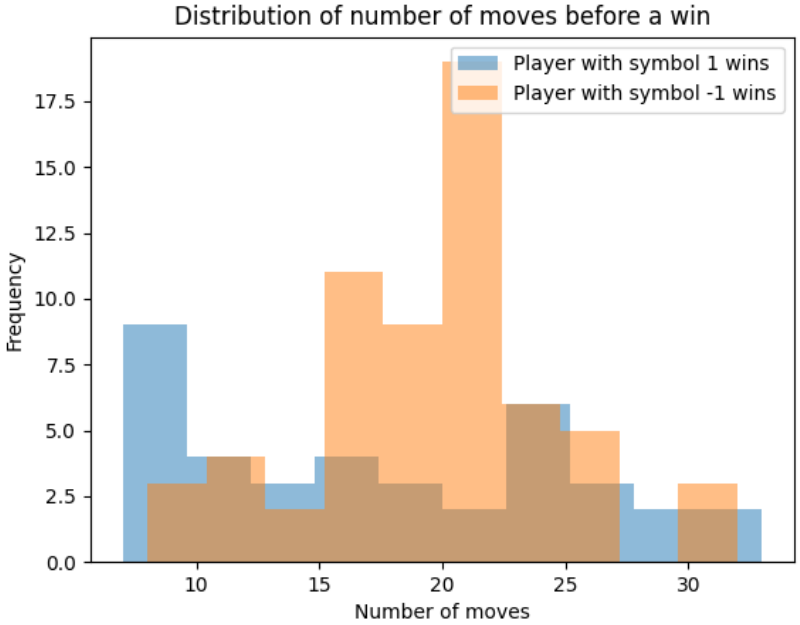

Figure 7 : Monte Carlo vs Monte Carlo; jouer 1 a gagné 57 fois et jouer 2 a gagné 43 foi de 100 matchs
simulation_amount –> 1 vs. 2

Bandits-manchots

Nous noterons par la suite :

$N_T(a)$ - le nombre de fois où l'action (le levier) *a* à été choisi jusqu'au temps *T*.

$\hat{\mu}_T^a = \frac{1}{N_T(a)} \sum_{t=1}^{T} r_t \mathbf{1}_{a_t=a}$ - la récompense moyenne estimée pour l'action/levier *a* à partir des essais du joueur.

La liste des μ^i, qui sont des valeurs réelles entre 0 et 1, symbolise l'ensemble des leviers à travers les paramètres de la loi de Bernoulli liée à chaque levier.

Algorithme aléatoire :

L'algorithme aléatoire est la stratégie la plus simple parmi les quatre approches présentées dans le code. Dans cette stratégie, à chaque tour, un levier est choisi au hasard parmi tous les leviers disponibles, sans tenir compte des résultats précédents.

Fonctionnement:

- Initialisation (*__init__*) :

Lors de l'initialisation, l'algorithme prend deux arguments : le nombre total de fois que le joueur souhaite jouer (*total_play_times*) et les leviers disponibles (*levers*).
Il détermine également le nombre total de leviers disponibles en utilisant la méthode *get_levers* de la classe *Levers*.

- Choix du levier (*_choice*) :

Cette méthode génère un indice aléatoire qui détermine quel levier sera choisi parmi tous les leviers disponibles. Pour cela, elle utilise la fonction *randint* de la bibliothèque *random* pour obtenir un nombre entier aléatoire entre 0 et le nombre total de leviers moins un.

- Simulation du jeu (*play*) :

Cette méthode simule le choix aléatoire des leviers pour le nombre total de fois spécifié (*total_play_times*).
Pour chaque choix, l'algorithme :
  - Utilise la méthode *_choice* pour déterminer quel levier jouer.
  - Met à jour le compteur de ce levier spécifique pour garder une trace du nombre de fois où chaque levier a été joué.

- Utilise la méthode *pick_lever* de la classe *Levers* pour simuler le jeu de ce levier et obtenir le résultat (récompense ou non).
- Ajoute le résultat au gain total.

À la fin de la simulation, la méthode retourne un tuple contenant deux éléments : la liste des compteurs pour chaque levier (indiquant combien de fois chaque levier a été joué) et le gain total accumulé pendant toute la simulation.

Bien que cette méthode soit simple à comprendre et à mettre en œuvre, elle n'est généralement pas la plus efficace pour maximiser les récompenses dans le problème du bandit manchot.

Algorithme greedy

L'algorithme greedy est une stratégie qui se base sur une phase d'exploration initiale pour estimer la valeur de chaque levier, puis il exploite cette connaissance en choisissant toujours le levier avec la meilleure valeur estimée.

Fonctionnement:

- Initialisation (*__init__*):

Lors de l'initialisation, l'algorithme prend trois arguments : le nombre total de fois que le joueur souhaite jouer (*total_play_times*), les leviers disponibles (*levers*) et un facteur d'avidité (*greed*) qui détermine combien de fois chaque levier sera essayé pendant la phase d'exploration.
Il détermine également le nombre total de leviers disponibles en utilisant la méthode *get_levers* de la classe *Levers*.

- Exploration des leviers (*explore*):

Cette méthode parcourt chaque levier et calcule une valeur moyenne basée sur le nombre de fois que ce levier donne une récompense pendant la phase d'exploration.
Elle utilise la méthode *_explore_lever* pour obtenir la valeur moyenne pour chaque levier.

- Exploration d'un levier spécifique *(_explore_lever*):

Cette méthode simule le jeu d'un levier spécifique un certain nombre de fois (déterminé par le facteur d'avidité divisé par le nombre total de leviers).
Elle totalise les récompenses obtenues pour ce levier pendant cette phase d'exploration et calcule la valeur moyenne.
Cette valeur moyenne est ensuite retournée.

L'action choisie est :

$$a_t = \text{argmax}_{i \in \{1,\dots,N\}} \hat{\mu}_t^i$$

- Simulation du jeu (*play*):

Après la phase d'exploration, cette méthode identifie le levier ayant la meilleure valeur moyenne.

Pour le reste des tours (c'est-à-dire *total_play_times* moins le nombre total de tours d'exploration), l'algorithme choisit toujours ce levier "optimal".
Le résultat de chaque choix est ajouté au gain total.

Bien que cette stratégie puisse être efficace si la phase d'exploration est suffisamment longue pour fournir des estimations précises, elle présente un risque : si la phase d'exploration n'identifie pas correctement le meilleur levier, l'algorithme continuera d'exploiter un levier sous-optimal pendant la phase d'exploitation.

Algorithme ε-greedy :

L'algorithme ε-greedy est une stratégie qui cherche à équilibrer l'exploration et l'exploitation. Au lieu de s'en tenir strictement à l'action qui semble actuellement la meilleure (comme dans l'approche avide), l'algorithme ε-greedy choisit occasionnellement une action au hasard. Cette approche permet d'éviter de s'enfermer trop tôt dans une action qui semble optimale, mais qui pourrait ne pas l'être en réalité.

Fonctionnement:

- Initialisation (*__init__*)*:*

Lors de l'initialisation, l'algorithme prend plusieurs arguments, dont le nombre total de jeux (*total_play_times*), les leviers disponibles (levers), un facteur d'avidité (*greed*), la valeur de ε (epsilon) et un indicateur pour savoir si une exploration préliminaire doit être effectuée (*preliminary_exploration*).
La valeur de ε détermine la probabilité de choisir un levier au hasard à chaque étape.

- Exploration initiale (*_explore*):

Si preliminary_exploration est activé, l'algorithme commence par une phase d'exploration pour estimer la valeur de chaque levier. Cette phase utilise la stratégie greedy.

- Choix au hasard (*_pick_random_choice*):

Cette méthode est utilisée lorsque l'algorithme décide d'explorer (c'est-à-dire de choisir un levier au hasard). Elle choisit un levier, joue avec et met à jour le gain total et les statistiques pour ce levier.

- Meilleur choix (*_pick_best_choice*):

Cette méthode est utilisée lorsque l'algorithme décide d'exploiter (c'est-à-dire de choisir le levier qui a le meilleur rendement moyen jusqu'à présent). Elle choisit le levier avec le meilleur rendement moyen, joue avec et met à jour le gain total et les statistiques pour ce levier.

- Simulation du jeu (*play*):

Si l'exploration préliminaire est activée, elle est effectuée en premier.
Pour chaque tour, l'algorithme génère un nombre aléatoire entre 0 et 1. Si ce nombre est inférieur ou égal à ε, il choisit un levier au hasard. Sinon, il choisit le levier avec le meilleur rendement moyen.
À la fin, il retourne le nombre de fois que chaque levier a été joué et le gain total.

Cette stratégie est souvent utilisée dans les problèmes d'apprentissage par renforcement car elle permet d'éviter de se concentrer trop tôt sur une action qui semble être la meilleure, tout en continuant à capitaliser sur les connaissances acquises.

Algorithme UCB

L'algorithme UCB est une stratégie avancée pour le problème du bandit manchot. Plutôt que de se baser uniquement sur les récompenses moyennes observées, UCB prend également en compte le niveau d'incertitude ou la confiance autour de ces moyennes. L'idée est d'explorer de manière optimale en donnant la priorité aux leviers qui ont soit un potentiel de gain élevé soit une grande incertitude.

Fonctionnement:

- Initialisation *(__init__)*:

Lors de l'initialisation, l'algorithme prend en compte le nombre total de jeux (*total_play_times*) et les leviers disponibles (*levers*).
Il initialise également des listes pour suivre le retour moyen de chaque levier et le nombre de fois que chaque levier a été joué.

- Jeu d'un levier (*_play_lever*):

Cette méthode joue un levier spécifié, obtient le résultat (récompense ou non), et met à jour le gain total ainsi que les statistiques pour ce levier spécifique.

- Mise à jour des moyennes et des compteurs (*_calculate_new_average_and_amount*) :

Après avoir joué un levier, cette méthode est utilisée pour mettre à jour la moyenne des retours pour ce levier et le nombre de fois qu'il a été joué.

- Calcul du levier à jouer *(_calculate_choice*):

Cette méthode est le cœur de l'algorithme UCB. Pour chaque levier, elle calcule un poids basé sur le retour moyen de ce levier et un terme d'exploration. Ce terme d'exploration est basé sur le nombre total de jeux jusqu'à présent et le nombre de fois que le levier a été joué.
Le levier avec le poids le plus élevé (combinaison de la moyenne observée et du terme d'exploration) est choisi pour être joué.

L’action choisie est :

$$a_t = \text{argmax}_{i \in \{1,\dots,N\}} \left(\hat{\mu}_t^i + \sqrt{\frac{2\log(t)}{N_t(i)}} \right)$$

- Simulation du jeu (*play*):

Pour chaque tour, l'algorithme utilise la méthode *_calculate_choice* pour déterminer quel levier jouer, puis joue ce levier et met à jour les statistiques.

À la fin, il retourne le nombre de fois que chaque levier a été joué et le gain total.

UCB est souvent considéré comme une méthode efficace pour le problème du bandit manchot car elle permet d'explorer de manière intelligente tout en exploitant les connaissances acquises.

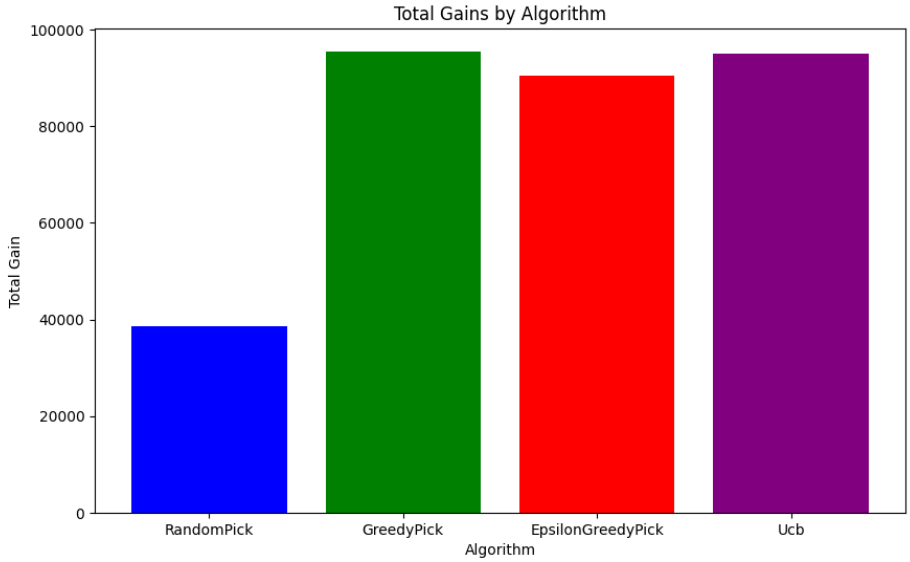

Figure 8 : La comparaison des quatre algorithmes.
greed = 1000, ε = 0.1, *levers* = 100, *premilinary_exploration* = False
total_play_times = 100000, *amount_of_levers* = 20

Pour le paramètre *premilinary_exploration* défini sur False, le graphique semble toujours similaire, sans grande variation dans les résultats.