3I005
TME 2–4 : Projet Exploration / Exploitation

Ariana Carnielli
Yasmine Ikhelif

Table des matières

1 Introduction

Ce mini-projet s'intéresse à la problématique de l'exploitation *vs* exploration, qui consiste à choisir, parmi une quantité de ressources limitées, combien de ces ressources vont être utilisées pour explorer le problème, afin d'avoir plus de chances de trouver la meilleure solution possible, et combien seront utilisées pour exploiter cette meilleure solution trouvée. Si beaucoup de ressources sont dépensées pour l'exploration, on aura plus de chances de trouver une solution proche de l'optimale, mais moins de ressources disponibles pour son exploitation. De l'autre côté, arrêter l'exploration trop tôt peut conduire au choix d'une action sous-optimale pendant la phase d'exploitation, ce qui conduit à un gain plus petit à long terme.

Dans ce mini-projet, on illustre la problématique de l'exploitation *vs* exploration dans trois situations. La première, décrite dans la Section 2, s'intéresse à l'exemple des bandits-manchots. Il s'agit de considérer une machine à sous à N leviers, chacun ayant une probabilité de victoire différente inconnue du joueur. L'objectif est de maximiser le gain, ce qui nécessite un bon équilibre entre l'exploration des différents leviers afin d'avoir une bonne estimée de la probabilité

de gain de chacun et l'exploitation du meilleur levier. L'algorithme principal permettant cet équilibre est l'algorithme UCB, dont les détails sont donnés dans la Section 2.

La deuxième situation considérée, décrite dans la Section 3, est celle du jeu de morpion, où trois stratégies de jeu sont implémentées. La première consiste dans une stratégie purement aléatoire, la deuxième est une stratégie de Monte Carlo, qui explore les actions possibles de façon aléatoire et choisit la meilleure et la dernière, une stratégie de Monte Carlo Tree Search, qui utilise l'algorithme UCB pour optimiser l'exploration des actions possibles.

Finalement, ces mêmes algorithmes, ayant été codés de façon généraliste, sont appliqués à un jeu légèrement plus complexe que morpion, le jeu puissance 4. Les détails de l'implémentation et les résultats obtenus sont donnés dans la Section 4.

2 Bandits-manchots

2.1 Description

On considère une machine à sous avec N leviers, numérotés par les entiers de 0 à $N-1$. Chaque levier, lorsqu'il est actionné, peut donner une récompense de 0 ou 1 de façon aléatoire. On suppose que tous les leviers sont indépendants, que deux actionnements différents du même levier sont aussi indépendantes, et que la récompense du levier i suit une loi de Bernoulli de paramètre μ^i constant en temps.

On dispose de T coups pour jouer à la machine à sous. À chaque coup $t \in \{0,\dots,T-1\}$, on choisit une *action* $a_t \in \{0,\dots,N-1\}$, qui représente le levier choisi pour ce coup, et on accumule le gain obtenu avec ce levier, noté par r_t. Ainsi, r_t est une variable aléatoire suivant une loi de Bernoulli de paramètre μ^{a_t}. L'objectif est de maximiser le gain total G_T obtenu au bout de T coups, $G_T = \sum_{t=0}^{T-1} r_t$. Comme G_T est une variable aléatoire, on maximise son espérance, qui vaut

$$\mathbb{E}(G_T) = \mathbb{E}\left(\sum_{t=0}^{T-1} r_t\right) = \sum_{t=0}^{T-1} \mathbb{E}(r_t) = \sum_{t=0}^{T-1} \mu^{a_t}.$$

Ainsi, si les $\mu^0,\dots,\mu^{N-1}$ étaient connus, la stratégie maximisant son espérance serait de choisir

$$a_t = \underset{i\in\{0,\dots,N-1\}}{\operatorname{argmax}}\ \mu^i$$

pour tout $t \in \{0,\dots,T-1\}$. Si on note μ^* la valeur maximale de $\{\mu_0,\dots,\mu_{N-1}\}$ et G_T^* la variable aléatoire G_T avec la stratégie a_t définie ci-dessus, le maximum de l'espérance de G_T vaut $\mathbb{E}(G_T^*) = \sum_{t=0}^{T-1} \mu^* = T\mu^*$.

Comme les μ^i ne sont pas connus, la stratégie ci-dessus ne peut pas être utilisée en pratique. Pour une autre stratégie, on s'intéresse au regret L_T du joueur au bout de T coups, défini comme la différence entre l'espérance du gain obtenu avec la stratégie optimale ci-dessus et son gain réel, $L_T = \mathbb{E}(G_T^*) - G_T = T\mu^* - \sum_{t=0}^{T-1} r_t$. Cette définition est légèrement différente de celle donnée dans l'énoncé du projet car on utilise l'espérance de G_T^* au lieu d'un G_T^* aléatoire. Ce choix, arbitraire, a été fait pour assurer que, lorsque deux résultats identiques sont obtenus par deux joueurs, leur regret sera aussi identique.

On est alors confronté à un problème du type exploitation *vs* exploration, où l'exploration consiste à tester les leviers afin d'estimer leurs paramètres μ^i et l'exploitation consiste à jouer le levier avec le plus grand paramètre μ^i estimé.

2.2 Implémentation du jeu

L'implémentation de la machine à sous a été faite en stockant les leviers d'une machine comme un tableau de taille N contenant les paramètres $\mu^0,\dots,\mu^{N-1}$ des N machines. La fonction `cree_machine` permet de créer un tel tableau avec des paramètres choisis de façon aléatoire uniforme dans $[0,1]^N$. Un coup du jeu est simulée par la fonction `jouer`, qui prend en

argument une machine et l'action / levier choisi et rend le gain correspondant au coup joué. Une stratégie pour le jeu est une fonction quelconque prenant en argument un tableau `mu` donnant la proportion de gains de chaque levier et un deuxième tableau `Na` contenant la quantité de coups jouées dans chaque levier, et qui renvoie un coup à jouer à la prochaine étape. Les T coups d'une partie sont simulés par la fonction `run`, qui prend en argument la machine, la stratégie et T et qui utilise les fonctions précédentes pour simuler les T coups, renvoyant à la fin les dernières versions des tableaux `mu` et `Na` et un tableau de taille T avec le gain cumulé à chaque $t \in \{0, \dots, T-1\}$. La fonction `run` utilise la fonction auxiliaire `mise_a_jour` qui met à jour les tableaux `mu` et `Na` en fonction du levier choisi et du résultat obtenu.

2.3 Stratégies

Dans ce projet, quatre stratégies ont été implémentées.

- **Algorithme aléatoire.** Dans cet algorithme, le levier à jouer dans le coup suivant est choisi de façon aléatoire uniforme parmi les N leviers disponibles. Il s'agit d'un algorithme qui fait uniquement une exploration des N leviers sans utiliser aucune information obtenue pour exploiter le meilleur levier observé.
- **Algorithme glouton.** L'idée de cet algorithme est de décomposer le jeu en deux phases bien distinctes, l'une avec uniquement de l'exploration et l'autre avec uniquement de l'exploitation. Dans la première, les N leviers sont joués chacun la même quantité de fois, passée en argument à la stratégie, afin d'estimer les paramètres μ^i de chaque levier par la proportion de gains obtenue expérimentalement sur ce levier stockée dans le tableau `mu`. Remarquons que cette estimée est justifiée par la loi des grands nombres, qui garantit que la proportion des gains obtenue expérimentalement avec le levier i converge vers le paramètre μ^i. Dans la deuxième phase, l'algorithme glouton exploite les informations obtenues en jouant toujours au levier correspondant au maximum du tableau `mu`.
- **Algorithme ε-glouton.** L'inconvénient de l'algorithme glouton est que le tableau `mu` contient uniquement des estimées de μ^i, et pas leurs vraies valeurs, qui sont inaccessibles. Ainsi, si ces estimées sont mauvaises, ce qui peut arriver de façon aléatoire, on risque de passer toute la phase d'exploration en jouant un levier qui n'est pas l'optimal. Pour y remédier, l'algorithme ε-glouton modifie l'algorithme glouton en introduisant un paramètre ε petit. À chaque coup, on a une probabilité ε de choisir un levier au hasard, de façon uniforme parmi les N leviers, et, avec une probabilité $1-\varepsilon$, on joue comme dans l'algorithme glouton. Cela garantit que, même après la fin de la première phase de l'algorithme glouton, on continue à avoir un peu d'exploration pour essayer d'améliorer les estimées de `mu` et ainsi éviter de rester bloqué sur une action sous-optimale.
- **Algorithme UCB.** L'algorithme ε-glouton permet de garder un peu d'exploration pendant la phase d'exploitation, mais cette exploration est faite de façon purement aléatoire. L'algorithme UCB cherche à faire une exploration de façon plus intelligente avec les informations disponibles. L'idée de cet algorithme est de choisir la prochaine action a_t par la formule

$$a_t = \underset{i \in \{0,\dots,N-1\}}{\operatorname{argmax}} \left(\hat{\mu}_t^i + \sqrt{\frac{2\log(t)}{N_i(t)}} \right),$$

où $\hat{\mu}_t^i$ est l'estimée de μ^i disponible à l'instant t et $N_i(t)$ est la quantité de fois que le levier i a été actionné jusqu'à l'instant t. Le terme $\sqrt{\frac{2\log(t)}{N_i(t)}}$ est celui responsable pour l'exploration ; sans ce terme, cette formule est exactement celle utilisée dans la phase d'exploitation de l'algorithme glouton. La division par $N_i(t)$ indique que, plus on a joué sur le levier i, plus notre estimée $\hat{\mu}_t^i$ est proche de μ^i. Le terme en $\log(t)$ garantit qu'on ne néglige pas complètement un levier : même si son estimée de $\hat{\mu}_t^i$ est trop petit, s'il a été trop peu joué, le terme $\sqrt{\frac{2\log(t)}{N_i(t)}}$

augmente au cours du temps et il sera éventuellement rejoué, afin d'améliorer encore l'estimée de $\hat{\mu}_t^i$. Comme la croissance de $\log(t)$ est assez lente, ces explorations de leviers avec une estimée petite n'arrivent pas très souvent pour ne pas trop nuire à l'exploration. Remarquons aussi que, comme il y a une division par $N_i(t)$, il est nécessaire de joueur chaque levier au moins une fois avant de pouvoir appliquer cet algorithme, ce qui impose une phase d'exploration pendant au moins les N premiers coups.

Ces quatre stratégies ont été implémentées comme quatre fonctions, `algo_alea`, `algo_glouton`, `algo_glouton_e` et `algo_UCB`, prenant toutes en argument les tableaux `mu` et `Na` comme décrit dans la Section 2.2. L'algorithme glouton prend aussi en argument facultatif la quantité de fois que chaque levier est joué pendant la phase d'exploitation et l'algorithme ε-glouton prend en argument facultatif la valeur de ε.

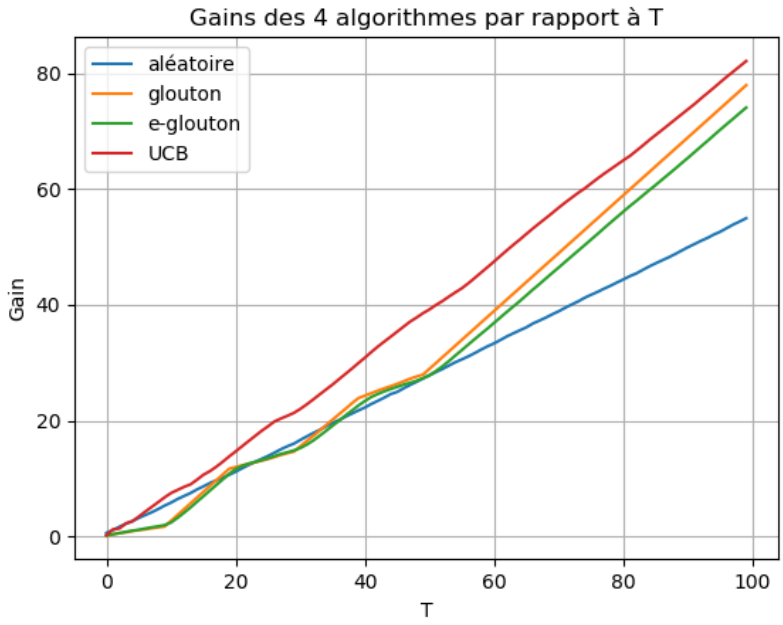
2.4 Résultats

La Figure 1 montre le gain et le regret en fonction de t pour les quatre algorithmes décrits dans la Section 2.3. Pour cette simulation, on a choisi une machine à $N = 5$ leviers, le paramètre de la loi de Bernoulli de chaque levier ayant été choisi de façon aléatoire uniforme dans $[0,1]$. On a fait une simulation avec 100 parties, chacune avec $T = 100$ coups. On a choisi une phase d'exploration de 10 coups par levier (donc 50 coups au total) pur l'algorithme glouton et $\varepsilon = 0.1$ pour l'algorithme ε-glouton. Les courbes sont les gains et regret moyens obtenus.

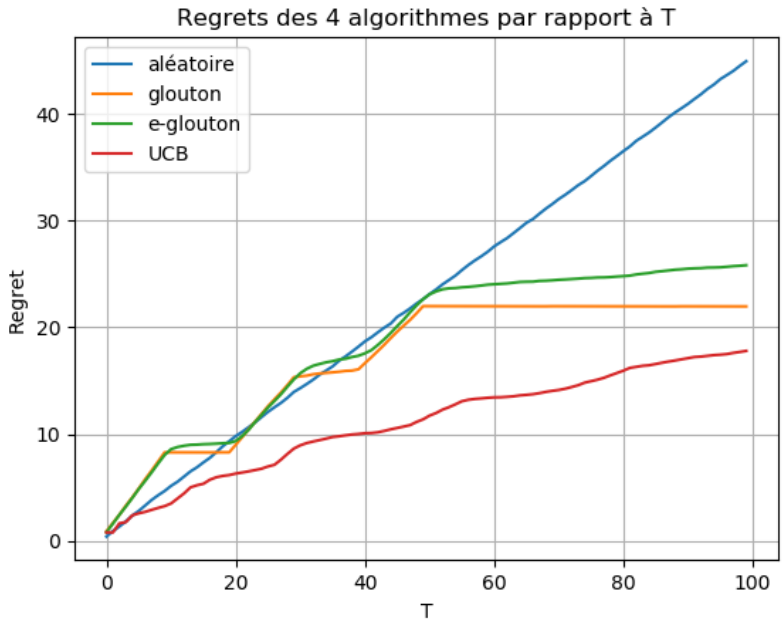
On remarque que, dans cette simulation, l'algorithme UCB donne le gain maximal (et donc le regret minimal) parmi les quatre. On y voit aussi clairement que, jusqu'à $t = 50$, les algorithmes glouton et ε-glouton sont en phase d'exploration, la croissance de leur gain et de leur regret dépend ainsi de quel levier ils choisissent à chaque instant. À la fin de cette phase, l'algorithme glouton a trouvé le bon levier et donc son regret reste constant. Comme l'algorithme ε-glouton garde un peu d'exploration, son regret croit lentement après cette phase. L'algorithme UCB a un regret qui croit à chaque fois plus lentement, et de façon plus lente que les autres, lui donnant le plus petit regret à la fin.

La Figure 2 montre les histogrammes des regrets au temps $T = 100$ après 10000 parties pour chacun des 4 algorithmes, avec les mêmes paramètres que pour la Figure 1. On y remarque que l'algorithme UCB donne toujours les plus petits regrets, qui sont concentrés entre 15 et 24. Le comportement des algorithmes glouton et ε-glouton sont similaires, mais avec un regret toujours plus grand pour le ε-glouton car, avec une phase d'exploration de 50 coups, l'algorithme glouton a beaucoup de chances de trouver le meilleur levier parmi les 5, et donc l'exploration faite par l'algorithme ε-glouton après les premiers 50 coups ne lui fait qu'augmenter son regret. Comme attendu, le regret de l'algorithme aléatoire est le pire des quatre.

On peut aussi comparer la qualité des estimées des μ^i obtenus par les quatre algorithmes dans les simulations de la Figure 1, données dans la Table 1. On y remarque que l'algorithme aléatoire donne les meilleurs estimées, car cet algorithme ne fait que de l'exploration. L'algorithme UCB se concentre sur les leviers avec les plus grandes probabilités de gain, et ainsi ses estimées de μ^1 et μ^4 sont meilleurs que les autres. L'algorithme glouton fait une estimée avec 10 coups pour chaque levier et ensuite 50 coups supplémentaires pour le meilleur levier estimé, ce qui lui donne une bonne estimée de μ^1 mais une estimée plus mauvaise des autres μ^i. L'algorithme ε-glouton améliore les estimées par rapport à l'algorithme glouton.



(a)



(b)

FIGURE 1 – (a) Gain et (b) regret en fonction de t pour chacun des quatre algorithmes de la Section 2.3.

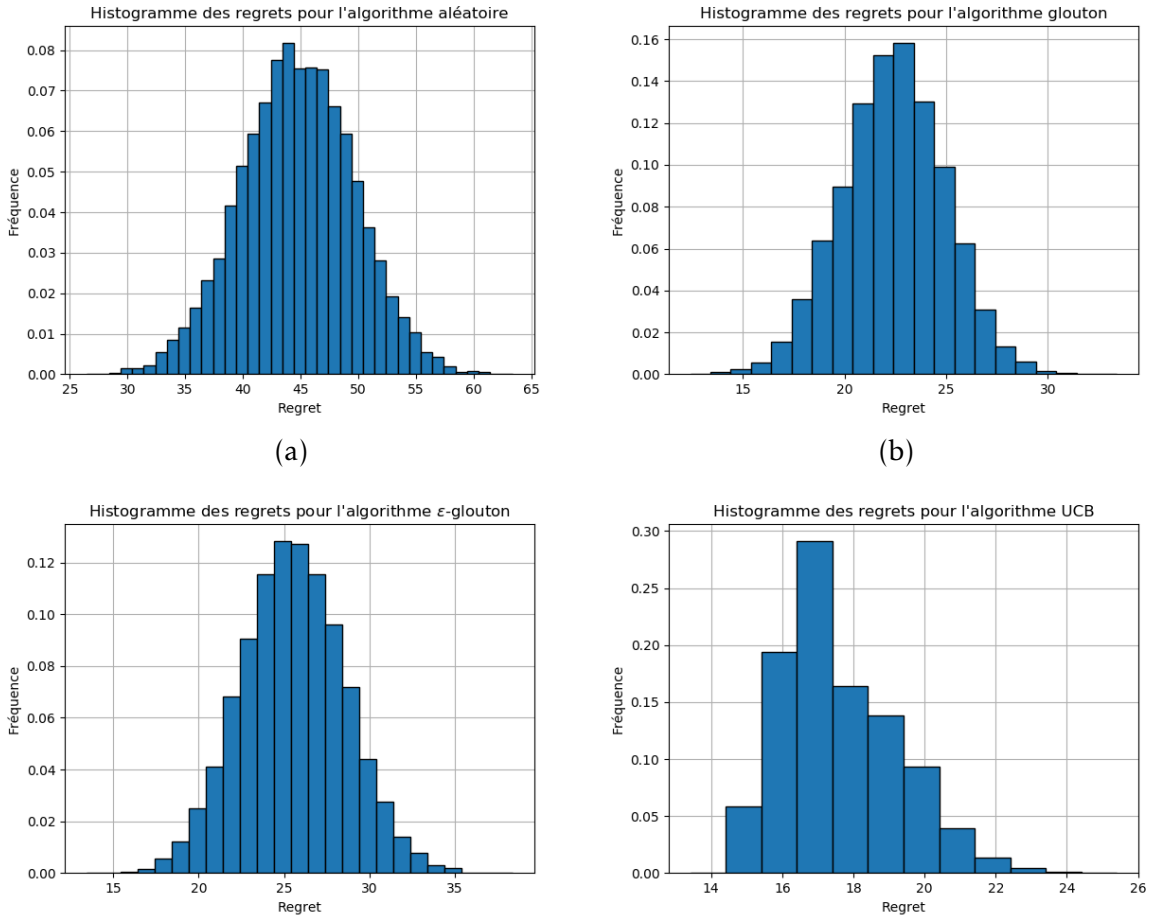


FIGURE 2 – Histogramme des regrets au temps $T = 100$ après 10000 parties pour les algorithmes (a) aléatoire, (b) glouton, (c) ε-glouton et (d) UCB.

	μ^0	μ^1	μ^2	μ^3	μ^4
Valeurs réelles	0.0225	0.9499	0.3808	0.1185	0.8149
Algorithme aléatoire	0.0205	0.9502	0.3688	0.1236	0.8139
Algorithme glouton	0.0270	0.9420	0.4070	0.1340	0.7833
Algorithme ε-glouton	0.0212	0.9396	0.3772	0.1136	0.8131
Algorithme UCB	0.0193	0.9420	0.3225	0.0995	0.7998

TABLE 1 – Valeurs réelles des paramètres μ^i et valeurs estimées par chacun des algorithmes.

3 Morpion

3.1 Description et implémentation fournie

Dans cette partie, on s'intéresse à des algorithmes pour joueur au morpion. Une implémentation de ce jeu a été fournie, basée sur quatre classes : une classe abstraite `State` permettant de représenter l'état générique d'un jeu de plateau à deux joueurs, une classe Jeu simulant un jeu générique, une classe `MorpionState` qui hérite de `State` et représente l'état d'un jeu de morpion, et une classe abstraite `Agent` permettant de représenter le comportement d'un agent. Un agent doit être capable, à partir de l'état courant du jeu, de renvoyer l'action qu'il choisit, c'est-à-dire la case qu'il marque avec son symbole, ce qui est fait par une méthode `get_action`

prenant en argument l'état courant `state` et renvoyant l'action choisie.

3.2 Stratégies

Ce projet implémente trois stratégies différentes pour joueur au morpion : une stratégie aléatoire, une stratégie de Monte Carlo et une stratégie Monte Carlo Tree Search.

- **Stratégie aléatoire.** La stratégie aléatoire, comme pour le problème des bandits-manchots de la Section 2, n'utilise aucune information sur l'état du jeu outre les actions possibles et fait un choix aléatoire uniforme d'une de celles-ci. Si une stratégie aléatoire n'est pas intéressante par soi-même, elle sera utilisée comme une partie des deux autres stratégies.
- **Stratégie Monte Carlo.** Les algorithmes de ce type consistent à remplacer une exploration exhaustive de l'espace de possibilités par un parcours d'un échantillonnage aléatoire uniforme de cet espace afin d'obtenir une approximation de quelle action est optimale sans tout calculer. Dans notre cas, à un état donné, pour chaque action possible, la stratégie de Monte Carlo simule plusieurs parties aléatoires (à l'aide de la stratégie aléatoire précédente) et calcule le taux de victoire de chaque action, choisissant celle avec le plus grand taux.
- **Stratégie Monte Carlo Tree Search (MCTS).** La stratégie de Monte Carlo précédente consiste à regarde uniquement les actions possibles à partir de l'état courant et simuler des joueurs aléatoires ensuite. La stratégie Monte Carlo Tree Search s'intéresse plutôt à l'arbre de toutes les possibilités du jeu à partir de l'état courant et jusqu'à toutes les fins possibles du jeu. Cet algorithme cherche à parcourir cet arbre de façon intelligente, en choisissant en priorité les branches qui ont plus de chances de conduire à une victoire. Plus précisément, il commence par explorer toutes les actions possibles à partir de l'état actuel au moins une fois et utilise l'algorithme UCB décrit dans la Section 2.3 pour choisir quelle branche explorer en priorité après, continuant ainsi jusqu'à un nombre prédéterminé d'explorations.

3.3 Implémentations

Dans cette partie, on présente de façon plus détaillée l'implémentation des stratégies Monte Carlo et Monte Carlo Tree Search. L'implémentation de l'algorithme aléatoire n'est pas décrite car elle est triviale.

3.3.1 Monte Carlo

L'algorithme de Monte Carlo a été implémenté en créant une classe `AgentMC` héritant de la classe `Agent`. Elle implémente la méthode `get_action` en récupérant la liste d'actions possibles à partir de l'état courant et construisant deux dictionnaires indexés par les actions, l'un avec le nombre total de victoires pour chaque action et l'autre avec le nombre total de fois où cette action a été jouée. Pour garantir une exploration minimale, chaque action est jouée au moins une fois, avant de commencer une boucle où, à chaque tour, une action est choisie au hasard de façon uniforme, le prochain état du jeu est calculé, et ensuite cet état est joué par deux stratégies aléatoires afin de déterminer si cette action conduit à une victoire ou une défaite. À la fin une quantité prédéterminée de tours de boucle, la méthode renvoie l'action avec la plus grande proportion de victoires. La quantité de tours de boucle est fixée comme `n` fois le nombre d'actions possibles, où `n` est passé en argument facultatif à la classe.

3.3.2 Monte Carlo Tree Search

Comme la stratégie MCTS utilise un parcours d'arbre, on commence d'abord par implémenter une structure d'arbre adaptée à nos besoins. Cette structure se base sur la classe `Noeud`, qui stocke l'état du jeu sur ce nœud, son nœud parent (ou `None` si c'est la racine), un dictionnaire avec ses nœuds fils, indexés par les actions qui conduisent à chaque fils, et les quantités de

défaites et de parties jouées à partir de ce nœud. Cela veut dire qu'un nœud stocke la quantité de *défaites* du joueur dont c'est le tour à ce nœud.

La classe `Noeud`, outre son constructeur, contient deux fonctions. La première, `maj`, permet de mettre à jour le nombre de défaites de ce nœud à partir d'un résultat d'une partie et propage cette information de façon récursive à son parent jusqu'à la racine. La deuxième, `choix_ucb`, choisit le prochain nœud à simuler à partir des principes de l'algorithme UCB. Si l'état est un état terminal, cette fonction renvoie le nœud lui-même. Sinon, si son tableau avec ses enfants n'a pas encore été initialisé, cette fonction l'initialise. Elle parcourt ensuite le tableau d'enfants et, si elle trouve un enfant qui n'a pas encore été joué, la fonction lui retourne. Sinon, elle utilise l'algorithme UCB pour choisir un de ses enfants et fait un appel récursif à partir de cet enfant.

La classe principale de la stratégie MCTS est la classe `AgentMCTS`, qui hérite de `Agent` et implémente la fonction `get_action`. Elle initialise d'abord un nœud `racine` lié à l'état courant, crée ses fils directs et joue une fois avec la stratégie aléatoire à partir de chacun d'eux. Ensuite, elle fait une boucle avec une quantité prédéterminée de tours. À chaque tour, la fonction `choix_ucb` est appelée sur la racine afin de déterminer un nœud terminal ou qui n'a pas encore été joué. Un jeu est simulé à partir de ce nœud et le résultat est propagé vers ses ascendants à l'aide de la fonction `maj`. À la fin de cette boucle, la fonction regarde la quantité de défaites des fils de la racine et choisit celui avec le plus grand taux de défaites, ce qui correspond au plus grand taux de victoires pour le joueur de la racine. La quantité de tours de boucle est fixée comme `n` fois le nombre d'actions possibles, où `n` est passé en argument facultatif à la classe.

3.4 Résultats

Avant de présenter les résultats, rappelons que, dans le morpion, le premier joueur a un avantage naturel. Par exemple, lorsque le jeu se joue jusqu'à ce que la grille soit remplie, le premier joueur aura joué 5 coups, alors que le deuxième n'en aura joué que 4. En plus, le premier joueur a à sa disposition, au début, les 8 façons possibles de gagner (3 lignes, 3 colonnes et les deux diagonales), mais, après son premier coup, le deuxième joueur aura moins de façons possibles de gagner (4 si le premier joueur commence par le centre, 5 s'il commence par l'un des quatre coins ou 6 s'il commence dans l'une des quatre autres cases).

La Figure 3 montre l'évolution des proportions des victoires des joueurs 1 et 2 et de matchs nuls en fonction du nombre N de parties jouées pour deux joueurs aléatoires. Elle illustre l'avantage donné par le jeu au premier joueur, qui a environ deux fois plus de chances de victoire que le deuxième joueur. Ces moyennes deviennent évidemment plus précises lorsque le nombre de parties jouées augmente.

La Table 2 présente les proportions de victoires des joueurs 1 et 2 et de matchs nuls au bout de 1000 parties pour chaque combinaison possible des stratégies pour les deux joueurs. Pour les joueurs Monte Carlo et MCTS, le paramètre `n` déterminant la quantité de tours de boucle dans son implémentation a été fixé à 20.

La première ligne de la Table 2 met en évidence l'avantage du premier joueur et confirme les résultats montrés dans la Figure 3. Elle montre aussi la hiérarchie entre les stratégies aléatoire, Monte Carlo et MCTS. Ainsi, même en tant que premier joueur, la stratégie aléatoire perd beaucoup de Monte Carlo et presque à chaque fois de MCTS, qui a un taux de 88% de victoires malgré le désavantage d'être le second joueur. En premier joueur, la stratégie de Monte Carlo gagne presque à chaque fois de l'aléatoire et a un bon taux de victoires contre elle-même, mais, contre MCTS, le plus probable est d'avoir un match nul, avec aussi une grande probabilité pour que MCTS gagne, même en second joueur. Lorsque MCTS est le premier joueur, il gagne presque systématiquement contre l'aléatoire, avec un taux très faible de match nuls et aucune défaite. Son taux de victoires contre Monte Carlo est lui aussi très grand. Quand deux joueurs MCTS jouent l'un contre l'autre, la probabilité de match nul est assez élevée, et, lorsqu'il n'y a pas match nul, c'est presque toujours le premier joueur qui l'emporte. Cela se rapproche de la stratégie optimale pour morpion, dans laquelle on a un match nul à coup sûr. Il est possible

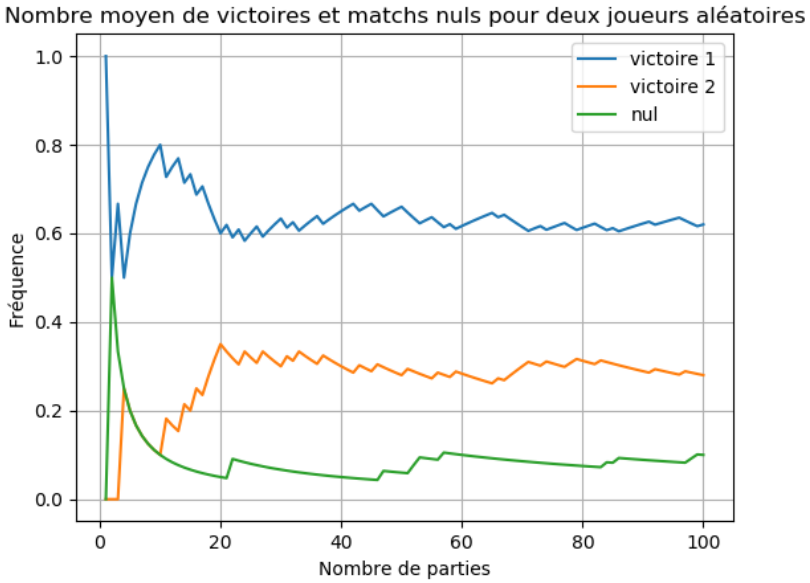


FIGURE 3 – Évolution des proportions de victoires des joueurs 1 et 2 et de matchs nuls entre deux joueurs aléatoires.

Joueur 1	Joueur 2	Victoire du joueur 1	Match nul	Victoire du joueur 2
Aléatoire	Aléatoire	57.2%	11.7%	31.1%
Aléatoire	Monte Carlo	14.5%	6.3%	79.2%
Aléatoire	MCTS	1.7%	10.3%	88.0%
Monte Carlo	Aléatoire	94.8%	3.8%	1.4%
Monte Carlo	Monte Carlo	64.1%	15.7%	20.2%
Monte Carlo	MCTS	13.7%	45.8%	40.5%
MCTS	Aléatoire	99.3%	0.7%	0.0%
MCTS	Monte Carlo	83.9%	15.8%	0.3%
MCTS	MCTS	23.2%	76.2%	0.6%

TABLE 2 – Proportions de victoires des joueurs 1 et 2 et de matchs nuls pour les différentes combinaisons possibles de stratégies dans le jeu de morpion.

de se rapprocher de cette situation en augmentant le paramètre n de MCTS déterminant le nombre de tours de boucle, mais cela rend aussi ce joueur plus lent. Avec $n = 20$, l'ensemble des simulations de la Table 2 a pris environ 30min pour s'exécuter, dont environ 20min pour le remplissage de ses trois dernières lignes.

Une amélioration envisageable du code serait de réutiliser l'arbre d'un coup à l'autre. En effet, à chaque fois que l'on demande son action à un joueur MCTS, il construit à partir de zéro l'arbre du jeu à partir de l'état actuel. Une fois son coup choisi et retourné, cet arbre n'est plus réutilisé, et un nouvel arbre est construit à partir de zéro à la prochaine fois qu'on lui demande une action. Pour garder de l'information entre un coup et l'autre, on pourrait stocker l'arbre comme variable d'instance pour le joueur et le mettre à jour à chaque coup joué. Pour cela, il faut prendre en compte le coup qu'il a joué et aussi le coup joué par l'autre joueur.

4 Puissance 4

4.1 Description et implémentation

Les algorithmes pour jouer au morpion de la Section 3 ont été appliqués au jeu Puissance 4. Ce jeu se joue sur une grille à 6 lignes et 7 colonnes et l'objectif d'un joueur est d'aligner 4 de ses pièces dans l'horizontale, verticale ou diagonale. Les joueurs ne peuvent pas placer leurs pièces de façon arbitraire : à son tour, un joueur choisit une colonne et y place sa pièce obligatoirement dans la ligne la plus en bas possible.

Les stratégies implémentées pour le morpion marchent sans aucune modification pour le jeu Puissance 4, car elles se basent sur les méthodes de la classe abstraite `State`. Ainsi, pour implémenter ce jeu, il suffit d'implémenter une classe héritant de `State`, que l'on a appelée `Puissance4State`. Dans cette classe, il suffit d'implémenter les fonctions `get_actions`, qui donnent les actions possibles d'un joueur à son tour, `win`, qui détermine si l'un des deux joueurs a gagné le jeu et, si oui, lequel, et `stop`, qui détermine si le jeu est terminé, soit par la victoire d'un des deux joueurs ou par le remplissage complet de la grille.

4.2 Résultats

Comme pour le jeu de morpion, le jeu Puissance 4 donne un avantage au premier joueur, ce qui a été découvert en 1988 de façon indépendante et presque simultanée par James Dow Allen et Victor Allis, mais cet avantage est moins significative que pour le jeu de morpion. Nous avons simulé 500 parties pour chaque combinaison possible de stratégies pour les deux joueurs d'un jeu de Puissance 4, les proportions de victoires des joueurs 1 et 2 et de matchs nuls étant représentés dans la Table 3.

Joueur 1	Joueur 2	Victoire du joueur 1	Match nul	Victoire du joueur 2
Aléatoire	Aléatoire	49.4%	5.0%	45.6%
Aléatoire	Monte Carlo	5.6%	0.4%	94.0%
Aléatoire	MCTS	0.0%	0.2%	99.8%
Monte Carlo	Aléatoire	95.8%	0.2%	4.0%
Monte Carlo	Monte Carlo	49.0%	4.6%	46.4%
Monte Carlo	MCTS	7.2%	4.6%	88.2%
MCTS	Aléatoire	100.0%	0.0%	0.0%
MCTS	Monte Carlo	94.0%	3.6%	2.4%
MCTS	MCTS	43.8%	17.4%	38.8%

Table 3 – Proportions de victoires des joueurs 1 et 2 et de matchs nuls pour les différentes combinaisons possibles de stratégies dans le jeu Puissance 4.

On remarque, dans le match entre deux joueurs aléatoires, l'avantage du premier joueur, qui est cependant beaucoup moins importante que dans le cas du morpion. Contre les stratégies Monte Carlo ou MCTS, le joueur aléatoire perd presque à chaque fois, peu importe s'il est le premier ou deuxième joueur. Lorsque les deux joueurs suivent la stratégie Monte Carlo, les résultats ne sont pas très différents que dans le cas de deux joueurs aléatoires. Encore une fois, le joueur MCTS est celui avec la meilleure stratégie, gagnant presque toutes les parties contre les autres joueurs. Le cas entre deux joueurs MCTS est aussi celui avec le plus d'occurrences de matchs nuls.

5 Conclusion

Ce rapport a permis, dans la Section 2, de mettre en évidence la problématique de l'exploitation *vs* exploration à travers l'exemple des bandits-manchots, qui illustre bien l'intérêt d'un

algorithme équilibrant exploration et exploitation comme l'algorithme UCB. On a également pu remarquer, dans les Sections 3 et 4, que l'utilisation de cet algorithme dans la stratégie Monte Carlo Tree Search permet d'obtenir des joueurs très performants.