3I005 : Statistique et Informatique

Projet :

Exploration/Exploitation

Par :
Angelo ORTIZ
Celina HANOUTI

Licence d'Informatique
Année 2018/2019

Table des matières

Table des figures

Introduction

En intelligence artificielle, plus précisément en *Machine Learning*, on est souvent amené, pour un ensemble de choix possibles, à trouver un bon compromis entre l'exploration et l'exploitation. L'exploration consiste à obtenir d'avantage d'informations dans le but de prendre de meilleures décisions dans le futur. L'exploitation, quant à elle, consiste à agir de manière optimale en fonction de ce qui est déjà connu. Ce dilemme constitue un problème largement étudié dans le domaine de l'apprentissage par renforcement. En effet, beaucoup d'algorithmes ont été proposés pour résoudre un certain nombre de ces problèmes.

Dans ce projet, nous allons étudier et analyser un certain nombre d'algorithmes basés sur le dilemme de *l'exploration vs exploitation* à travers des IAs de jeu. Dans un premier temps, nous allons effectuer une évaluation expérimentale d'algorithmes classiques dans un cadre simplifié. La deuxième partie est consacrée à l'implémentation d'un algorithme de Monte Carlo et enfin, dans les deux dernières parties, nous allons étudier des algorithmes plus avancés ainsi qu'un jeu de stratégie combinatoire.

Première partie

Bandits-manchots

1 Description du problème

On considère le problème d'apprentissage par renforcement suivant. Un agent est confronté à plusieurs reprises à un choix parmi N différentes actions dont la récompense moyenne, notée μ^i pour l'action i, est inconnue de l'agent. Nous désignons l'action sélectionnée sur le pas de temps t par a_t, la récompense correspondante par r_t et le nombre de fois où l'action a a été choisie jusqu'au temps T par $N_T(a)$.

L'objectif de l'agent est de maximiser la somme des récompenses obtenues après la période d'apprentissage, i.e. les T premières parties. Autrement dit, l'agent doit identifier l'action au rendement le plus elevé i^*= $\text{argmax}_{i \in 1,\dots,N} \mu^i$ qu'il jouera continument par la suite. Cependant, les valeurs μ^i étant inconnues, il est donc contraint d'estimer ces récompenses moyennes, notée $\hat{\mu}^i$ pour l'action i, au moyen desdites parties d'apprentissage. Ces estimations sont ainsi données par :

$$\hat{\mu}^i = \frac{1}{N_T(a)} \sum_{t=1}^{T} r_t \mathbb{1}_{a_t = a}$$

2 Algorithmes

Si de nombreux algorithmes ont été proposés pour résoudre ce dilemme, on se contentera ici d'en citer que quelques-uns :

2.1 Algorithme aléatoire

L'algorithme aléatoire se contente de choisir une action uniformément au hasard et il effectue ainsi uniquement de l'exploration. De ce fait, nous l'utilisons comme baseline pour les tests présentés par la suite. On peut donc déjà présager que cet algorithme sera forcément le moins optimal.

2.2 Algorithme *greedy*

L'algorithme *greedy* est basé sur une politique de selection très simple qui consiste à sélectionner l'une des actions ayant la valeur estimée la plus élevée : a_t = $\text{argmax}_{i \in 1,\dots,N} \hat{\mu}_t^i$. Autrement dit, l'algorithme *greedy* ne fait qu'éxploiter les connaissances dont il dispose afin de maximiser la recompense à un instant t.

2.3 Algorithme ε-*greedy*

Une approche courante pour trouver un compromis exploitation/exploration est l'algorithme ε-*greedy* qui consiste à choisir à l'instant t, avec une probabilité $1 - \varepsilon$, l'action qui maximise le rendement moyen sur les estimations faites jusque-là et avec une probabilité ε, une action tirée d'une distribution uniforme. L'avantage de cet algorithme repose sur le fait que, au fur et mesure que le nombre d'étapes d'apprentissage augmente, chaque action sera choisie un très grand nombre de fois assurant ainsi que tous les $\hat{\mu}^i$ convergeront vers les μ^i.

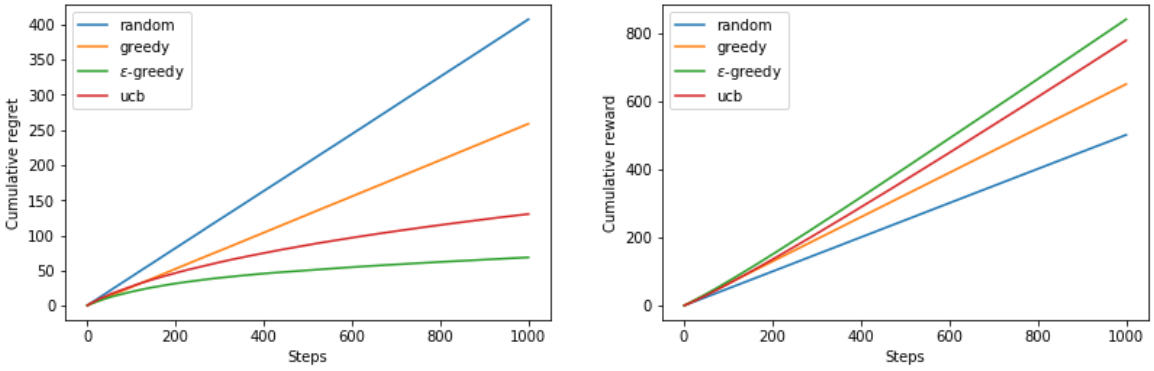
Par ailleurs, la valeur de ε a un fort impact sur l'apprentissage. C'est pour cela qu'on en a testé différentes valeurs (constantes) et même sous la forme d'une fonction décroissante du temps. On peut conjécturer la puissance de cet algorithme et sa capacité à explorer toutes les actions possibles et de prendre des décisions quasi proches de l'optimum.

2.4 Algorithme *Upper Confidence Bound*

L'algorithme UCB est une autre stratégie qui permet de trouver une balance entre l'exploration et l'exploitation. L'idée de l'algorithme consiste à se fier des estimations faites jusqu'alors à une borne supérieure de confiance. En effet, une incertitude existera toujours sur l'optimalité de ces estimations.

Les deux algorithmes définis précédemment choisissent toujours l'action qui semble meilleure à un instant t ou se contentent de choisir des actions aléatoirement sans distinction. L'algorithme UCB prend en revanche en compte l'optimalité des estimations mais également l'incertitude sur celles-ci. Il sélectionne l'action : $a_t = \mathrm{argmax}_{i \in 1,\ldots,N}(\hat{\mu}_t^i + \sqrt{\frac{2\log(t)}{N_T(i)}})$. L'incertitude est mesurée par le deuxième terme, il permet notamment de privilégier les actions le plus défavorisées par les choix de l'agent.

3 Mise en oeuvre et expérimentations



(a) Regret cumulé en fonction du nombre d'étapes d'apprentissage

(b) Récompense cumulée en fonction du nombre d'étapes d'apprentissage

FIGURE 1 – Performance des quatres algorithmes sur 10 leviers et 2000 exécutions

La figure 1 montre le gain et le regret cumulé en fonction de t pour les quatre algorithmes décrits dans la section précédente. Nous avons fait le choix de simuler avec une machine à 10 léviers, chacun d'entre eux suivant une loi de Bernoulli avec un paramètre choisi uniformément dans [0,1].

Comme dit précédemment, les algorithmes aléatoire et *greedy* correspondent respectivement aux comportements purement exploratoire et d'exploitation. Les courbes confirment ainsi nos hypothèses de départ, à savoir que ces comportements amèneraient à des résultats sous-optimaux, voire mauvais.

En revanche, les deux autres algorithmes trouvent un équilibre entre les deux comportements au taux d'équilibre entre l'exploration et l'exploitation près ; c'est pourquoi les résultats sont plus intéressants. Cependant, le taux d'équilibre utilisé n'est pas le même dans les deux cas. En effet, l'algorithme ε-*greedy* ($\varepsilon = 0.04$) choisit des actions au hasard dans 4 % des cas, ce qui reste un taux assez bas. Ceci contraste avec la "méfiance" de l'algorithme UCB en ses connaissances, ce qui le fait garder un comportement plus exploratoire dans le temps. Nous pouvons voir ces différences dans les valeurs $N_T(a)$: pour UCB, l'écart entre elles reste réduit en raison de sa "méfiance", alors que pour ε-*greedy*, d'après son trait d'exploitation, il y a une (ou très peu de) valeur qui est beaucoup plus élevée que les autres.

Par ailleurs, le comportement exploratoire constant dans le temps de

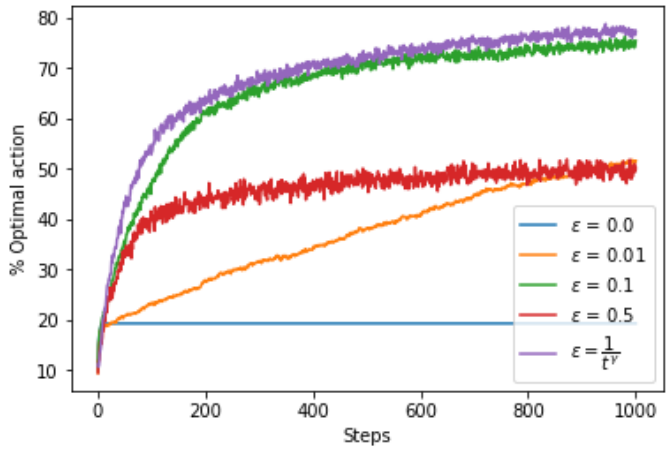


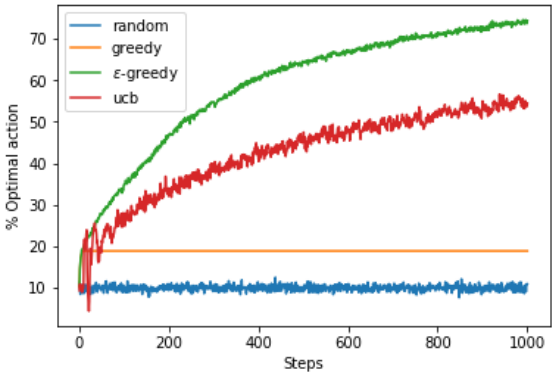
FIGURE 2 – Pourcentage de choix du meilleur levier par ε-*greedy*

l'algorithme ε-*greedy* reste un choix raisonnable au vu de la figure 2. Nous pouvons y apprécier que, malgré ses choix sous-optimaux dans les 200 premiers pas, il arrive à de résultats aussi bons, notamment pour $\varepsilon = 0.1$, que ceux de la méthode avec $\varepsilon = \frac{1}{t^{0.35}}$, i.e. décroissant dans le temps.

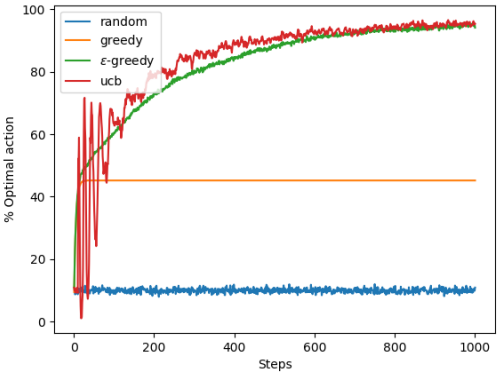
La figure 3 montre l'impact de l'écart entre la meilleure récompense de la machine et la deuxième meilleure sur la performance des algorithmes. Pour l'algorithme aléatoire, cela reste identique dans les quatre cas car il ne possède aucune dépendance. Pour l'algorithme *greedy*, nous savons qu'il choisit une première action de manière aléatoire (car les estimations sont toutes initialisées à 0), et la garde tout au long de l'apprentissage. C'est pourquoi, pour un écart donné, le nombre de choix optimaux reste constant, mais, lorsque nous considérons l'ensemble d'écarts, les valeurs obtenues ne sont pas dépendantes les unes des autres.

Pour les deux autres algorithmes, comme ils possèdent un trait exploratoire, l'écart dans la distribution des récompenses a un effet remarquable. D'un côté, l'algorithme ε-*greedy* améliore ses performances car, dès la reconnaissance du meilleur levier, il le choisit majoritairement. D'autre côté, pour l'algorithme UCB, il y a un pourcentage d'écart optimal qui se trouve entre 25 % et 50 % : tout comme pour l'algorithme ε-*greedy*, son trait d'exploitation fait qu'il améliore ses performances, mais, comme déjà mentionné, il effectue plus d'exploration que ce dernier et il continue à ce faire même lorsqu'il a trouvé le meilleur levier. Dans ce cas-là, nous disons que l'algorithme cherche à améliorer ses connaissances déjà optimales.

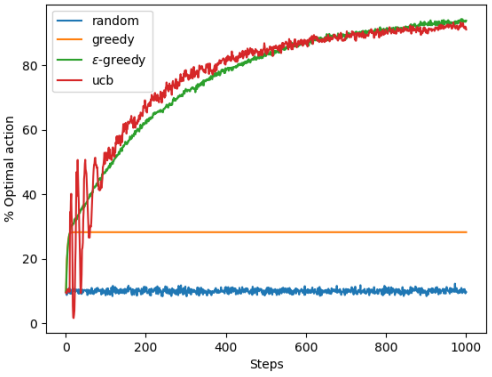
Pour la dernière expérience sur la machine à N-leviers, nous avons fait varier le nombre de leviers. Pour ce faire, nous avons fixé la longueur de la période d'apprentissage à 1000 pas et effectué 500 exécutions avec



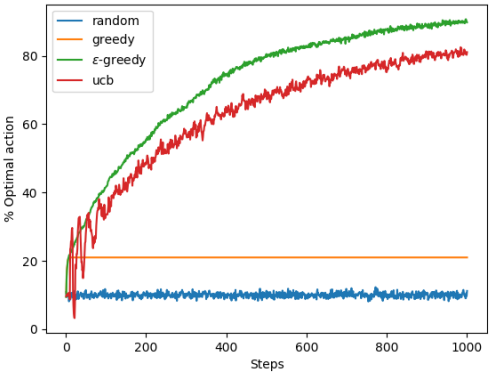
(a) Sans écart fixé



(b) Écart de 25 %



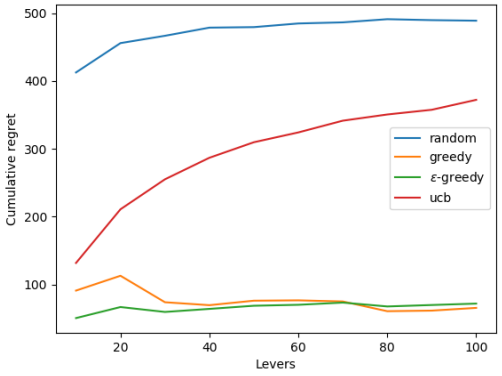
(c) Écart de 50 %



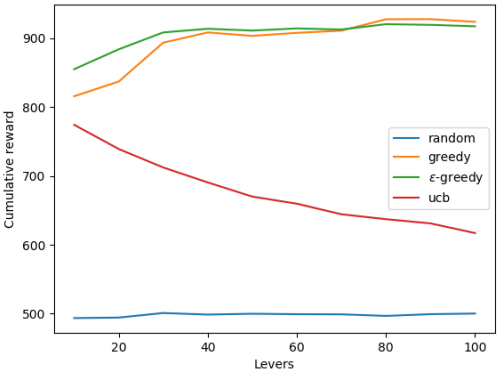
(d) Écart de 75 %

FIGURE 3 – Performance des quatres algorithmes sur 1000 pas d'apprentissage et 2000 exécutions pour différentes proportions entre les deux meilleures récompenses moyennes

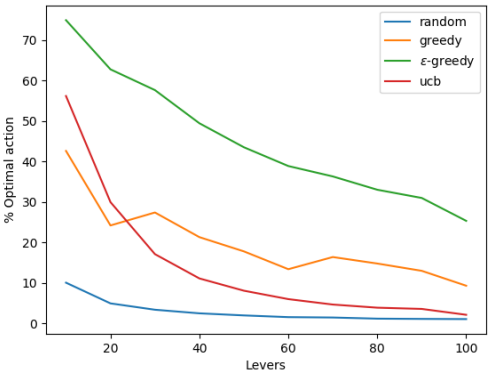
des distributions de récompenses aléatoires. Au fur et à mesure que le nombre de leviers augmente, les agents ont moins de chances de retrouver le levier optimal avec un nombre de pas d'apprentissage fixé. Ceci explique ainsi les courbes sur la figure 4-c. En effet, caractère exploratoire ou pas, il y a plus de leviers à tester et ça veut dire plus de pas nécessaires.



(a) Regret cumulé en fonction du nombre d'étapes d'apprentissage



(b) Récompense cumulée en fonction du nombre d'étapes d'apprentissage



(c) Nombre de choix optimaux en fonction du nombre d'étapes d'apprentissage

FIGURE 4 – Performance des quatres algorithmes sur 1000 pas d'apprentissage et 500 exécutions pour différents nombres de leviers

Deuxième partie

Morpion et Monte Carlo

4 Description du problème

Dans cette partie, on considère le célèbre jeu du Tic Tac Toe qui se joue sur une grille 3×3. Les joueurs alternent tour à tour en plaçant un «X» ou un «O» sur une case vide de la grille. Le premier joueur à aligner trois symboles gagne. Nous allons considérer deux types de joueurs, des joueurs *Random* qui se contenteront de choisir leurs prochains coups à au hasard et des joueurs qui, eux, utiliseront une méthode de *Monte Carlo*. L'implémentation de ce jeu est basée sur quatre classes : `State` qui représente l'état du jeu, une classe `Agent` qui elle, représente le comportement d'un agent. La méthode `get_action` permet à l'agent de choisir le prochain coup à jouer.

5 Joueur Aléatoire

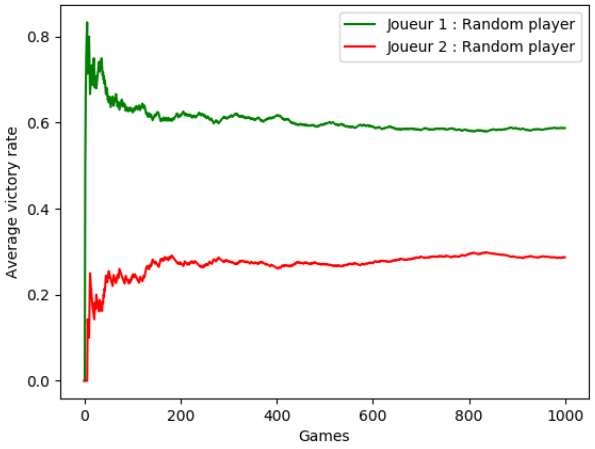


FIGURE 5 – Espérance de gain pour deux joueurs *Random*

La stratégie aléatoire consiste à choisir aléatoirement une action parmi les actions possibles, autrement dit, un joueur aléatoire choisira une action en exploitant uniquement les informations du jeu. On considère une partie comme étant une expérience à deux issues : victoire ou défaite du premier joueur. On note X la variable aléatoire qui dénote la victoire du

Joueur 1 ; en cas de victoire, on donne à X la valeur 1, sinon elle prend la valeur 0.

Il est évident alors que X suit une loi de Bernoulli. Sur la figure ci-dessus, on peut déduire visuellement le paramètre p. En effet, en moyenne, le Joueur 1 a une probabilité $\mathbb{P}(X{=}1) \simeq 0.58$ de gagner. On en déduit que X suit une loi de Bernoulli de paramètre $p \simeq 0.58$ et de variance $\mathbb{V}(X){=} p(1-p) \simeq 0.24$.

6 Joueur Monte Carlo

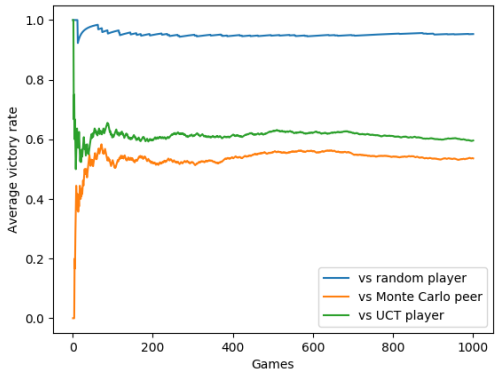
L'idée générale d'une simulation de $Monte\ Carlo$ consiste à jouer un certain nombre de parties avec des choix aléatoires puis d'utiliser les résultats de ces parties pour calculer une bonne action à jouer. Lorsqu'un joueur gagne une de ces parties aléatoires, il devra privilégier les actions qui l'ont mené à cette victoire, dans l'espoir de choisir un coup gagnant lors des prochaines parties. Une stratégie $Monte\ Carlo$ échantillonne aléatoirement l'espace de toutes les actions possibles et fait une estimation de l'action qui semble la plus optimale.

6.1 Implémentation

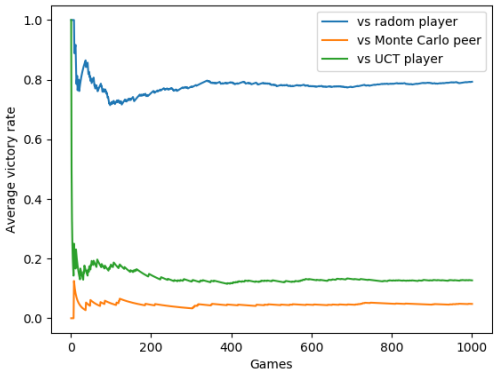
Pour implémenter cet algorithme nous avons créer une classe `MonteCarloPlayer` qui hérite de `Agent` et qui implémente la méthode `get_action` dans laquelle on recupère d'abord toutes les actions possibles. Ensuite, pour N itérations, on choisit une action aléatoirement et on recupère l'état du jeu engendré qui sera simulé jusqu'à terminaison par deux joueurs aléatoires. Enfin, on met à jour la récompense associée à cette action suivant le résultat de la simulation. La méthode renvoie l'action avec la plus grande récompense.

6.2 Mise en oeuvre

On peut observer sur la figure 6-a que le joueur basé sur la méthode $Monte\ Carlo$ surpasse largement le joueur $Random$ avec plus de 95% de parties gagnées. Cependant, il existe des situations où gagner est impossible pour le joueur $Monte\ Carlo$, malgré le fait que sa politique de jeu soit plus intelligente que celle du joueur $Random$. Par exemple, la défaite est inévitable si c'est le joueur $Random$ qui entame la partie et qu'il marque la case centrale en premier. Dans ce cas, beaucoup de voies seront bloquées dans la grille et donc, peu importe la stratégie du joueur, dans le meilleur cas, il fera match nul. Par ailleurs, $Monte\ Carlo$ gagne entre 50% et 70%



(a) Espérance de gain du Joueur Monte Carlo avec un nombre d'échantillonnage égale à 100 et lorsqu'il entame les parties



(b) Espérance de gain du Joueur Monte Carlo avec un nombre d'échantillonnage égale à 100 et lorsque c'est le joueur adverse qui entame les partie

FIGURE 6 – Comparaison entre un joueur $Monte\ Carlo$ et trois joueurs : Un joueur $Random$, un joueur UCT un un joueur suivant la même stratégie $Monte\ Carlo$

des parties jouées contre un joueur UCT et un autre joueur $Monte\ Carlo$ avec le même nombre d'échantillonnage.

La figure 6-b illustre très bien l'avantage donné au joueur qui entame la partie étant donnée que l'espérance de gain du joueur $Monte\ Carlo$ diminue considérablement face aux deux joueurs $Monte\ Carlo$ et UCT et diminue légèrement face au joueur $Random$. Cela peut s'expliquer, comme précédemment, par le fait qu'il est très difficile pour un joueur de gagner lorsqu'il n'entame pas la partie malgré le fait que sa stratégie de jeu soit plus optimale que celle de son adversaire. Une hiérarchie entre les trois stratégies peut donc être déduite de ces courbes, cependant, ces résultats expérimentaux sont toutefois dépendants d'autres paramètres comme le nombre d'échantillonnages et le facteur d'exploration de l'algorithme UCT.

Troisième partie

Monte Carlo Tree Search

7 Description de l'algorithme

L'algorithme Monte Carlo Tree Search est un algorithme de recherche heuristique qui explore intelligemment l'arbre des actions possibles à partir de l'état courant du jeu et ce, jusqu'à ce que toutes les configurations du jeu soient explorées. Cet algorithme choisit en priorité les branches qui sont le plus susceptibles de mener à une victoire. Il maintient par ailleurs un compromis entre l'exploitation et exploration en utilisant l'algorithme UCB pour choisir la prochaine branche à explorer en priorité.

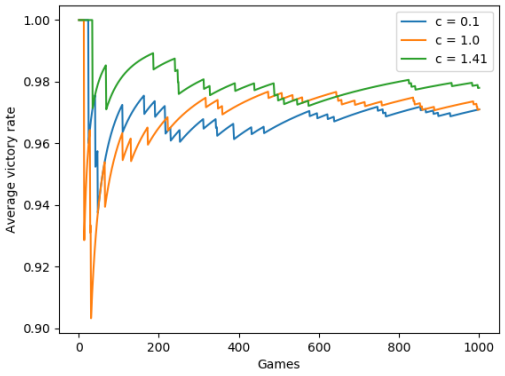
8 Implémentation

Nous avons implémenté cet algorithme à l'aide de trois classes, à savoir `State_Move`, `UCTree` et `UCTPlayer`. La première représente un couple état-coup, i.e. un nœud dans l'arbre de décision. Par ailleurs, `UCTree` est l'implémentation de cet algorithme à proprement parler car elle contient toutes les fonctionnalités nécessaires à son utilisation. Enfin, `UCPlayer` correspond au joueur `UCT`. En effet, il hérite de `Agent` et implémente la méthode `get_action` dans laquelle il crée un arbre enraciné à l'état courant du jeu et puis, pour N itérations, il simule une partie entre deux joueurs aléatoires et utilise la séquence de coups obtenue pour mettre à jour son arbre de décision en stockant le premier couple état-coup pas encore présent dans l'arbre. La méthode renvoie l'action avec la plus grande récompense.

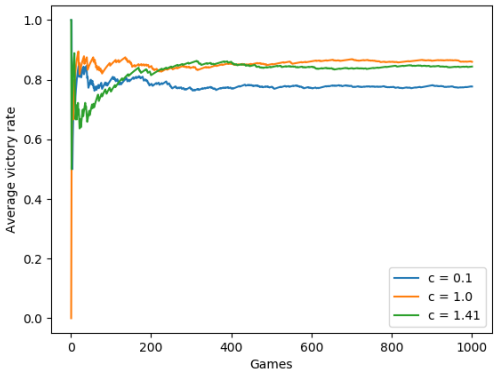
9 Mise en oeuvre

Dans la figure 7, de la même manière que dans la section 6, le joueur UCT est largement plus intelligent que le joueur aléatoire et ses résultats sont atténués lorsqu'il affronte le joueur Monte Carlo.

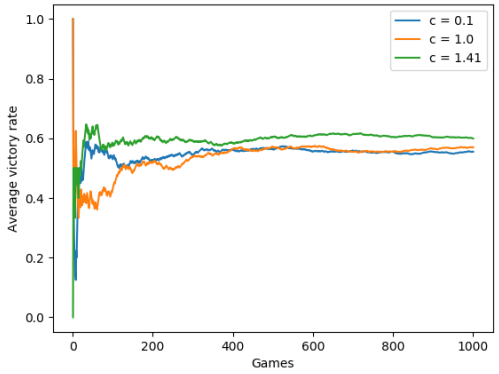
Par ailleurs, nous y voyons aussi que la répercussion du paramètre c modulant l'équilibre entre l'exploration et l'exploitation sur les performances de l'algorithme UCT est moindre. En effet, le joueur UCT effectue plus d'exploitation en raison de sa formule qui privilégie les meilleurs coups à différence du joueur $Monte\ Carlo$ qui explore beaucoup plus de résultats possibles.



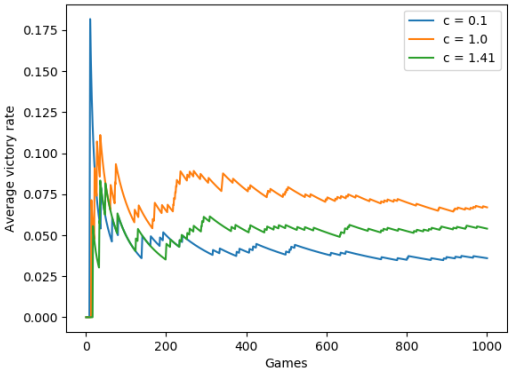
(a) Espérance de gain contre le joueur *Random* lorsqu'il entame les parties



(b) Espérance de gain contre le joueur *Random* lorsque ce dernier entame les parties



(c) Espérance de gain contre le joueur *Monte Carlo* lorsqu'il entame les parties



(d) Espérance de gain contre le joueur *Monte Carlo* lorsque ce dernier entame les parties

FIGURE 7 – Comparaison entre un joueur *UCT*, avec un nombre d'échantillonnages égal à 100, et les joueurs *Random* et *Monte Carlo*

C'est ce manque d'exploration qui explique les mauvais résultats du joueur UCT lorsqu'il joue en deuxième vis-à-vis du joueur *Monte Carlo*.

Conclusion

Dans ce projet, nous avons mis en évidence le dilemme de l'exploitation vs exploration à travers divers exemples, en l'occurrence deux jeux, où l'équilibre entre elles joue un rôle très important.

Au vu de nos expériences, nous pouvons confirmer que le paramètre modulant le choix entre les deux est dépendant du problème.

En ce qui concerne la machine à N leviers, nous avons constaté qu'un paramètre constant dans le temps peut être une solution acceptable, étant donné la simplicité de son implémentation, lorsque le besoin d'exploration est moindre.

La deuxième partie nous a servi pour vérifier expérimentalement une application de la loi de Bernoulli et son impact sur des comportements complexes dont elle est une brique de base.

Finalement, ce projet nous a permis de réaliser que, dans l'incertain, les estimations sont de la plus haute importance et c'est au moyen des probabilités que l'on détermine leur fiabilité.