



3I005 : STATISTIQUE ET INFORMATIQUE

Projet :  
*Exploration/Exploitation*

*Par :*  
*Angelo ORTIZ*  
*Celina HANOUTI*

Licence d'Informatique  
Année 2018/2019

# Table des matières

<b>Introduction</b>	<b>3</b>
<b>I Bandits-manchots</b>	<b>4</b>
1 Description du problème	4
2 Algorithmes	4
2.1 Algorithme Aléatoire . . . . .	4
2.2 Algorithme Greedy . . . . .	4
2.3 Algorithme $\varepsilon$ -Greedy . . . . .	5
2.4 Algorithme UCB . . . . .	5
3 Mise en oeuvre et expérimentations	6
<b>II Morpion et Monte Carlo</b>	<b>8</b>
4 Description du problème	8
5 Joueur Aléatoire	8
6 Joueur Monte Carlo	9
6.1 Implémentation . . . . .	9
6.2 Mise en oeuvre . . . . .	9
<b>III Monte Carlo Tree Search</b>	<b>11</b>
7 Description de l'algorithme	11
8 Implémentation	11
9 Mise en oeuvre	11
<b>Conclusion</b>	<b>12</b>

## Table des figures

1	Performance des quatres algorithmes sur 10 leviers et en moyennant les résultats sur 400 exécutions . . . . .	6
	(a) Regret cumulé en fonction du nombre d'étape d'apprentissage . . . . .	6
	(b) Recompenses cumulés en fonction du nombre d'étape d'apprentissage . . . . .	6
2	Performance des algorithmes Greedy et $\epsilon$ -Greedy . . . . .	7
	(a) Regret cumulé en fonction du nombre d'étape d'apprentissage . . . . .	7
	(b) Recompenses cumulés en fonction du nombre d'étape d'apprentissage . . . . .	7
3	Espérance de gain pour deux joueurs <i>Random</i> . . . . .	8
4	Comparaison entre un joueur <i>Monte Carlo</i> et trois joueurs : Un joueur <i>Random</i> , un joueur <i>UCT</i> un un joueur suivant la même stratégie <i>Monte Carlo</i> . . . . .	10
	(a) Espérance de gain du Joueur Monte Carlo avec un nombre d'échantillonnage égale à 100 et lorsqu'il entame les parties . . . . .	10
	(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 . . . . .	10

# 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 future. 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, Des algorithmes ont été proposés pour résoudre un certain nombre de problèmes qui illustrent des applications de ce dilemme.

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'implementation 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 suivant : un agent est confronté à plusieurs reprises à un choix parmi  $N$  différentes actions, chacune des  $N$  actions procure une récompense moyenne  $\mu^i$  inconnue par l'agent et nous désignons l'action sélectionnée sur le pas de temps  $t$  par  $a_t$ , la récompense correspondante par  $r_t$  et  $N_T(a)$  le nombre de fois où l'action  $a$  a été choisi jusqu'au temps  $T$ . L'objectif est de maximiser la somme des récompenses obtenue au bout des  $T$  premières parties, pour cela, l'agent doit identifier l'action au rendement le plus élevé  $i^* = \operatorname{argmax}_{i \in 1, \dots, N} \mu^i$ . Les valeurs  $\mu^i$  étant inconnues, il est donc nécessaire de faire des estimations qui doivent être le plus représentatives possible des valeurs  $\mu^i$ , notons  $\hat{\mu}^i$  ces estimations. L'estimation associée à une action est la récompense moyenne lorsque cette action est sélectionnée.

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

### 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 l'action uniformément au hasard, il effectue uniquement de l'exploitation et est utilisée 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 sélection très simple qui consiste à sélectionner l'une des actions ayant la valeur estimée la plus élevée :  $a_t = \operatorname{argmax}_{i \in 1, \dots, N} \hat{\mu}_t^i$ . Autrement dit, L'algorithme Greedy ne fait qu'exploiter les connaissances dont on dispose afin de maximiser la récompense à un instant  $t$ .

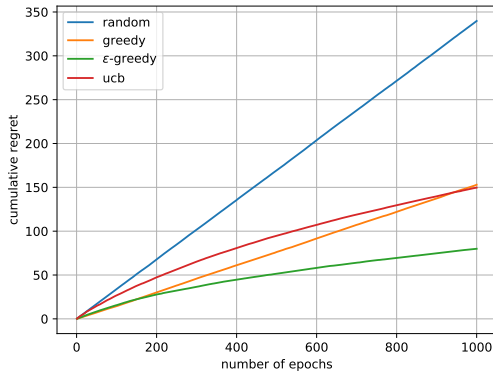
## 2.3 Algorithme $\varepsilon$ -Greedy

Une approche courante pour trouver un compromis exploitation/exploration est l'algorithme  $\varepsilon$ -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  $\varepsilon$ , une action uniformément au hasard. L'avantage de cet algorithme repose sur le fait que dans la mesure où le nombre d'étape d'apprentissage augmente, chaque action sera choisie un très grand nombre de fois assurant ainsi que tous les  $\hat{\mu}^i$  convergeront vers les  $\mu^i$ . De plus, la valeur de  $\varepsilon$  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 conjecturer la puissance de cet algorithme et sa capacité à explorer toutes les actions possibles et de prendre des décisions quasi proches de l'optimum.

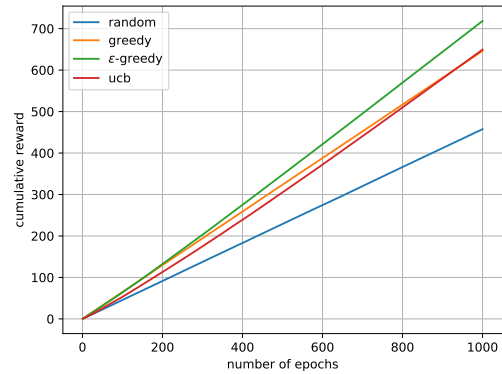
## 2.4 Algorithme UCB

L'algorithme UCB est une autre stratégie qui permet de trouver une balance entre l'exploration et l'exploitation. L'idée de cet algorithme consiste à se fier à une borne supérieure de confiance, en effet, une incertitude existera toujours sur l'optimalité des estimations faites jusque là. 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 compte l'optimalité des estimations mais également l'incertitude sur celles-ci. Il sélectionne l'action :  $a_t = \operatorname{argmax}_{i \in 1, \dots, N} (\hat{\mu}_t^i + \sqrt{\frac{2 \log(t)}{N_T(i)}})$ . L'incertitude est mesurée par le deuxième terme, il permet notamment de privilégier les actions le plus défavorisées dans le choix de l'agent.

### 3 Mise en oeuvre et expérimentations



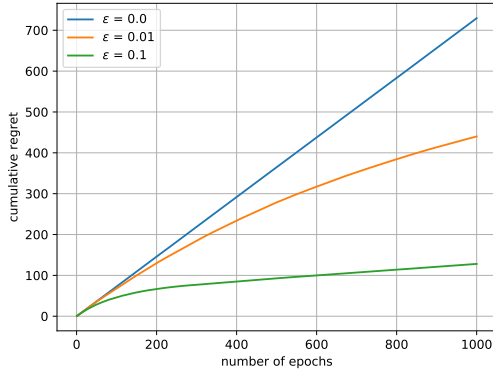
(a) Regret cumulé en fonction du nombre d'étape d'apprentissage



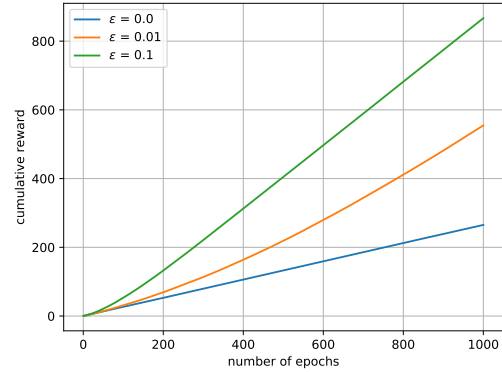
(b) Recompenses cumulés en fonction du nombre d'étape d'apprentissage

FIGURE 1 – Performance des quatres algorithmes sur 10 leviers et en moyennant les résultats sur 400 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 leviers, chacun suit une loi de Bernoulli avec un paramètre choisi uniformément dans  $[0,1]$ . TODO : commenter les courbes ci-dessus. Il me semble qu'on est sensé remarquer une amelioration si on change la distribution de gain. En effet, si on a des gains qui suivent une distribution normale plutôt qu'une distribution uniforme aléatoire, on remarquera une legere amelioration des performance pour chaque algorithme. Cela est logique, car l'action optimale est mieux "séparé" des autres actions (ici il serait plus judicieux de favoriser l'exploitation plutôt que l'exploration).



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



(b) Recompenses cumulés en fonction du nombre d'étape d'apprentissage

FIGURE 2 – Performance des algorithmes Greedy et  $\epsilon$ -Greedy

La figure 2 illustre une comparaison entre la méthode *Greedy* et deux autres méthodes  $\epsilon$ -*Greedy* ( $\epsilon=0.01$  et  $\epsilon=0.1$ ). On remarque que le regret cumulé et le nombre de récompenses augmentent avec le nombre de pas d'apprentissage, on peut observer également qu'au tout début, l'algorithme *Greedy* s'améliore en apprentissage au même rythme que les deux autres algorithmes puis on voit une diminution en performance. En effet, la méthode gloutonne effectue uniquement de l'exploitation, autrement dit, elle se contente de choisir l'action la plus prometteuse à un instant donné sans explorer d'autres actions qui pourrait l'être davantage, les échantillonnages initiaux qu'elle a effectué se sont avérés sous optimaux par la suite ce qui explique son déclin de performance. La méthode  $\epsilon$ -*Greedy* avec  $\epsilon=0.1$  a exploré bien plus que la méthode  $\epsilon=0.01$  ce qui explique que la plupart du temps elle identifie l'action optimale assez rapidement. La méthode  $\epsilon$ -*Greedy* avec  $\epsilon=0.01$  s'améliore en apprentissage plus lentement car elle exploite plus mais à un moment donnée, nous sommes sensés observer un gain de performance de tel sorte qu'elle donne de meilleurs résultats que  $\epsilon$ -*Greedy* avec  $\epsilon=0.1$ .



## 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  $3 \times 3$ . 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 jouer aléatoirement 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

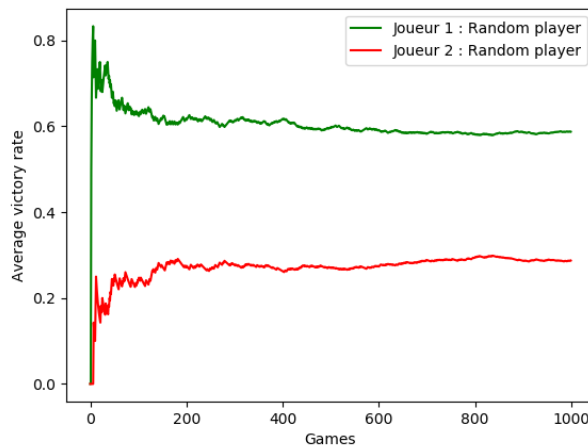


FIGURE 3 – Espérance de gain pour deux joueurs *Random*

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 Joueur 1. 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 et éviter les actions qui ont mené son adversaire à la défaite. 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.

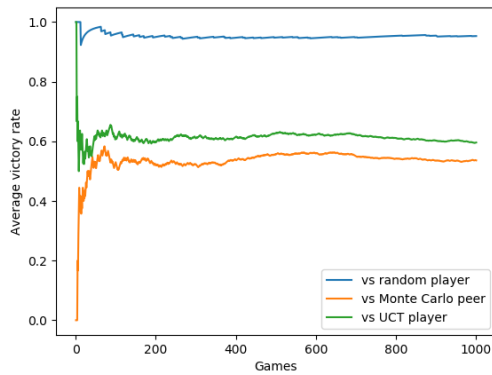
### 6.1 Implémentation

Pour implémenter cet algorithme nous avons créé une classe `MonteCarloAgent` qui hérite de `Agent` et qui implémente la méthode `get_action` dans laquelle on récupère d'abord toutes les actions possibles ensuite, pour  $N$  itérations, on choisit une action aléatoirement, on récupère l'état du jeu qui sera joué après cela par deux joueurs aléatoires, enfin, on met à jour la récompense associée à cet action selon le cas où elle a conduit à une victoire ou une défaite. La méthode renvoie l'action avec la plus grande récompense.

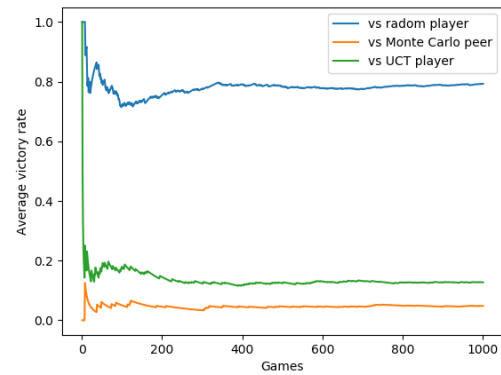
### 6.2 Mise en oeuvre

On peut observer sur la figure 4-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. *Monte Carlo* gagne entre 50% et 70% des parties jouées contre les joueurs *UCT* et un autre joueur *Monte Carlo* avec le même nombre d'échantillonnage. La figure 4-b illustre très

bien l'avantage donné au joueur qui entame la partie étant donnée que son espérance de gain 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 quasiment impossible de gagner pour un joueur 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'échantillonnage et le facteur d'exploration de l'algorithme *UCT*.



(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 parties

FIGURE 4 – Comparaison entre un joueur *Monte Carlo* et trois joueurs : Un joueur *Random*, un joueur *UCT* un un joueur suivant la même stratégie *Monte Carlo*

## **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**

## **9 Mise en oeuvre**

# Conclusion

Dans ce projet, nous avons mis en évidence le dilemme de l'exploitation *vs* exploration à travers divers exemples [à compléter]