Master 2 Intelligence Artificielle - Machine Learning
- Module : Apprentissage orienté agent -

Rapport

Projet apprentissage orienté agent

Cherti Mehdi

8 janvier 2014

Table des matières

Chapitre 1

Introduction

Dans ce projet, il s'agissait de faire une implémentation des différents algorithmes d'apprentissage orienté agent qu'on a appris dans le cours. Dans le rapport, je présenterai ce qui a été implémenté ainsi que les résultats des expérimentations.

Chapitre 2

Un seul état, un seul agent

Dans cette partie du cours, on a étudié un ensemble d'algorithmes qui s'appliquaient aux problèmes avec un seul état et un seul agent. Les algorithmes qui ont été implémentés sont : Greedy, Greedy-epsilon, Softmax, UCB.

2.1 Résultats

Les algorithmes ont été testés sur le problème du Bandit multi-armé.Dans les expérimentations, un ensemble de 500 problèmes de bandits a été généré, chaque problème a 10 actions, la moyenne de chaque action a été sélectionnée aléatoirement entre 0 et 100. la variance de chaque action a été fixée à 10.

Dans 2.1, on peut voir pour chaque algorithme l'évolution du coût moyen par rapport à l'itération. le coût moyen est la moyenne obtenue dans tous les problèmes de bandits générés à chaque itération.

Dans 2.2, on peut voir pour chaque algorithme l'évolution du pourcentage de sélection de la meilleure action par rapport à l'itération. Le pourcentage de sélection de la meilleure action est le nombre de fois où on sélectionne la meilleure action parmi tous les problèmes de bandits générés à chaque itération.

On peut voir dans 2.1 et 2.2 que UCB surpasse les autres algorithmes et que comme prévu, Greedy a le gain moyen le plus petit. On peut également observer l'effet de ϵ sur la convergence de Greedy Epsilon. Avec une grande valeur, le démarrage est plus lent parce qu'il ne fait qu'explorer mais avec le temps il trouve des bonnes approximations des vraies valeurs alors qu'avec ϵ petit, il ne fait qu'explorer et donc le gain moyen ne change pas énormément avec le temps. Il est à noter que UCB a été implémenté avec un C qui décroit avec le temps.

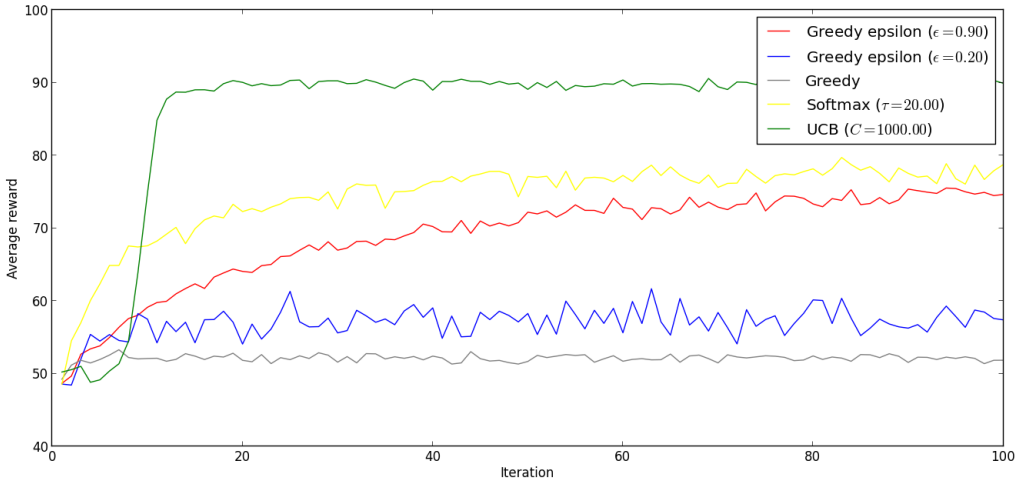

FIGURE 2.1 – Gain moyen en fonction de l'itération pour chaque algorithme

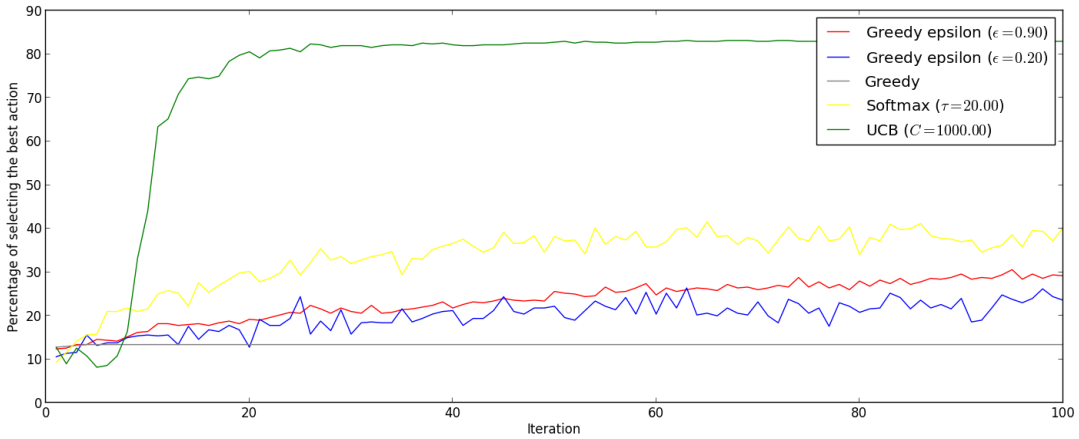

FIGURE 2.2 – Pourcentage de sélection de la meilleure action en fonction de l'itération pour chaque algorithme

Chapitre 3

Plusieurs états, un seul agent

Dans cette partie du cours, on a étudié le modèle MDP et on va vu plusieurs algorithmes qui permettaient d'évaluer une politique et l'améliorer. Les algorithmes qui ont été implémentés sont :

Pour l'évaluation de la politique :

- Bellman (résolution directe du système linéaire d'équations)
- Programmation dynamique
- Monte Carlo
- Différence temporelle

Pour l'amélioration de la politique :

- Alternance entre Monte Carlo pour l'évaluation et amélioration de la politique avec la méthode gloutonne
- Q-Learning

3.1 Résultats

Les algorithmes ont été testés sur un jeu de labyrinthe où l'objectif est d'atteindre un emplacement bien déterminé tout en minimisant le nombre de mouvements. Pour réaliser cela, l'agent aura un gain de 1 s'il atteint l'emplacement gagnant mais un gain de -0.1 à chaque mouvement.

Le labyrinthe avec lequel les algorithmes ont été testés est le suivant :

```
############
#   #     G#
#       #  #
# ####   # #
#     #    #
#  #### #  #
#   #      #
#      #####
#   #      #
#   ###### #
#     S    #
############
```

les # sont les murs, S est l'état initial, G est l'état gagnant.

3.1.1 Evaluation

Pour mesurer la pertinence de l'évaluation des différents algorithmes, chaque algorithme a été comparé avec l'algorithme "brute" qui résout le système d'équations linéaires de Bellman directement. La politique aléatoire uniforme a été utilisée. la distance entre les solutions trouvées par deux algorithmes est la distance euclidienne entre les vecteurs des états trouvés par ces algorithmes. En lançant l'algorithme de programmation dynamique, une distance de 0.4 a été trouvée avec les solutions réelles. Pour MonteCarlo, une adaptation a été faite pour qu'il puisse prendre en compte les paramètres du taux d'apprentissage et γ et donc pour qu'il puisse être comparable avec les autres algorithmes. γ a été fixé à 0.8.

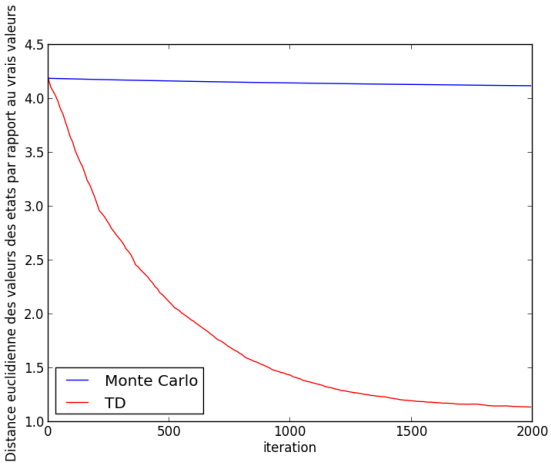

FIGURE 3.1 – Evolution de la distance des valeurs des états par rapport aux vraies valeurs pour TD et MC

Dans 3.1, on peut voir l'évolution des valeurs des états par rapport aux vraies valeurs pour TD et MonteCarlo (MC). Ils ont été tous les deux appliqués avec un $\alpha = 0.001$, étant donné qu'avec une valeur plus grande les deux algorithmes divergeaient. On peut voir dans 3.1 que MC se stabilise à un certain niveau alors que TD continue de décroître jusqu'à atteindre un nombre très petit d'erreurs par rapport aux vraies valeurs.

3.1.2 Solution optimale

Pour la recherche de la solution optimale, j'ai comparé MonteCarlo (alternance entre MonteCarlo pour l'évaluation et amélioration de la politique en sélectionnant la politique gloutonne) et QLearning. Les deux algorithmes ont été lancés avec 200 épisodes chacun. Pour QLearning, j'ai utilisé les valeurs suivantes : $\alpha = 0.9$, $\epsilon = 0.8$.

Dans 3.2, on peut voir les politiques optimales trouvées par MC et QLearning. Dans 3.2, pour chaque case on lui affecte la meilleure action trouvée par l'algorithme. Ainsi, U, D, R, L correspondent respectivement à haut, bas, droite,

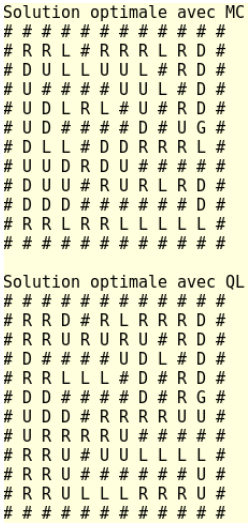

FIGURE 3.2 – Politique optimale avec MC (en haut) et QLearning (en bas)

gauche. G est la case gagnante. On peut voir que QLearning a clairement donné de meilleurs résultats que Monte Carlo (pour le même nombre d'épisodes).

Chapitre 4

Plusieurs états, plusieurs agents

Dans cette partie du cours, on a étudié des algorithmes qui s'appliquent à des problèmes avec plusieurs états et plusieurs agents. On a notamment étudié l'utilisation d'une méthode de type Monte Carlo nommée Monte Carlo search qui pouvait s'appliquer à des jeux comme Go.

J'ai implémenté un moteur du jeu de Go pour tester Monte Carlo search. Dans le moteur de Go implémenté,les scores de chaque joueur sont calculés en comptant le nombre d'intersections dans les territoires plus le nombre de pions que le joueur a sur le plateau. La partie s'arrête quand les deux joueurs décident de ne plus jouer. La règle du Ko a été respectée pour éviter les boucles infinies avec les joueurs non humains (MonteCarlo Search par exemple).

Le moteur du jeu (4.1) est assez flexible, chaque joueur peut être implémenté par une classe et il est possible donc de faire des duels entre différents joueurs assez facilement. Une classe pour un joueur humain a été implémentée également pour les tests. La règle du Ko a été respectée en utilisant le hashage de Zobrist (pour ne pas recalculer les hash à chaque modification des états) sur les états pour détecter les cycles de taille 3.

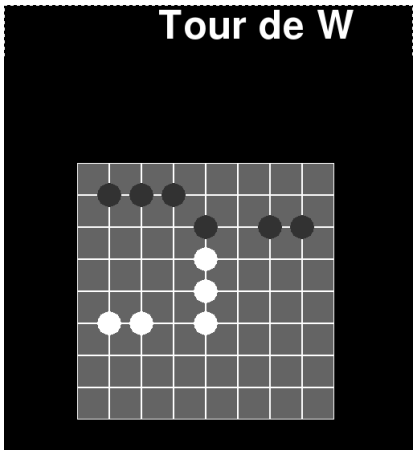

FIGURE 4.1 – Moteur de go et visualisation graphique

4.1 Monte Carlo Search

Plusieurs types de joueurs ont été implémentés. RandomPlayer permet de sélectionner à chaque fois, parmi les cases disponibles, une case au hasard. DeterministicPlayer permet de jouer des cases précises selon une liste qu'il reçoit en entrée. UserInterfacePlayer est le joueur humain. Finalement, MonteCarloPlayer est l'implémentation de Monte Carlo Search. MonteCarloPlayer commence par parcourir tous les états possibles jusqu'à une profondeur donnée ensuite il estime les valeurs des états de l'horizon en utilisant une ou plusieurs simulations aléatoires du jeu jusqu'à sa fin.

MonteCarlo Search a été testé contre lui même et contre le joueur random dans des plateaux de 4x4. Pour ne pas favoriser le joueur qui commence la partie, on joue des parties avec le joueur blanc qui commence la partie et d'autres avec le joueur noir. Dans 4.2, on peut voir les résultats des duels. Plusieurs duels sont joués et le gains moyens sont affichés à la fin pour chaque type de duel. On peut observer dans 4.2 que l'amélioration du nombre de simulations et de la profondeur ont amélioré les scores moyens. Des expérimentations avec des paramètres de valeurs plus grands n'ont pas pu être faits parce que l'exécution était assez lente.

```
10 jeux joués pour chaque cas, 5 avec le premier joueur qui commence, 5 avec le deuxieme qui commence
Deux Monte Carlo (profondeur=1, couleur=W) pour le premier, (profondeur=2, couleur=B) pour le deuxieme, nbr_simulations=1 pour les deux
{'B': 8.4, 'W': 7.5}
Monte Carlo (couleur=B, profondeur=1, nbr_simulations=1) et Random(couleur=W)
{'B': 7.9, 'W': 7.7}
Monte Carlo (couleur=B, profondeur=2, nbr_simulations=2) et Random(couleur=W)
{'B': 8.3, 'W': 7.5}
```

FIGURE 4.2 – Duels entre Monte Carlo Search Player et Random Player