TME 10: RL

Le module `mdpgames` (fourni sur le site de l'UE) implémente une plateforme générique pour expérimenter des algorithmes de reinforcement learning sur des petits jeux. Ce tme est consacré à l'étude des algorithmes pour le dilemme Exploration/Exploitation et aux techniques de programmation dynamique.
L'architecture grossière de la plateforme est la suivante :

- Chaque jeu hérite de la classe `Game` : le jeu est en temps discret, à chaque tour une action est demandée, puis l'état suivant est calculé.
- Un contrôleur permet d'intéragir avec le jeu, c'est à lui que le jeu demande l'action. Une fois l'action effectuée, le résultat est répercuté au contrôleur.
- Le contrôleur peut être manuel (un clavier, classe `KeyboardController`) ou algorithmique.
- La classe `RL` encode un contrôleur spécialement pour le reinforcement learning. Elle inclue un objet de type `Politique` qui permet de représenter une politique.

Voici les méthodes importantes pour la classe `Politique` (la seule classe que vous allez utiliser) :

```
Politique(object):
  def __init__(self,name,controller)
  # Retourn l action associe a un etat state
  def get_action(self,state)
  # Appeler apres l execution de l action, avec etat de depart, action, etat d arrivee, recompense
  def update(self,state_src,action,state_dest,reward):
  # Appeler en fin de senarios avec tout le log
  def update_scenario(self,log):
  #Lance un episode
  def run(self,maxiter=1000,display=True)
  #Lance maxiter episodes de longueur maxiterrun
  def campagne(self,display=True,maxiter=1000,maxiterrun=100)
  #retourne les logs du run sous forme [(state,action,state_src,reward),...]
  def get_log(self)
```

Exploration vs Exploitation

Le problème de l'exploration vs exploitation peut se résumer au cas simple suivant : un ensemble $\mathcal{A}$ de n actions a_i sont possibles, chacune procure une récompense moyenne μ_i^* inconnue, comment identifier le plus rapidement l'action la plus profitable ? Vaut-il mieux faire confiance à l'estimation courante et choisir l'action avec la meilleure récompense (option greedy), ou alors chercher une autre action sous-exploré ?
On utilisera les notations suivantes : μ_i^t la moyenne des récompenses reçue pour l'action i au bout de t pas, T_i^t le nombre de fois où l'action a_i a été sélectionnée au temps t (on identifiera l'action et l'indice de l'action).
Nous allons utiliser le jeu pierre-papier-ciseau (ou chifumi) étendu à n choix comme cas d'application. Dans ce jeu, il n'y a qu'un seul état, les 2 joueurs sélectionnent simultanément un nombre n_1 et n_2 entre 1 et n. Le joueur 1 gagne (a une récompense de 1) ssi $(n_1 - n_2)\%n \leq n/2$, égalité (récompense 0) si $n_1 = n_2$, perd sinon (récompense -1). Nous allons supposé que le joueur 2 choisit à chaque tour au hasard le nombre selon une distribution fixe donné. Dans ce contexte, le but est bien de trouver le numéro que le joueur 2 joue le plus souvent (afin de jouer les numéros qui le contre).
Les algorithmes que l'on se propose d'étudier :

- uniforme : l'action est choisie uniformément au hasard
- Greedy (glouton) : $argmax_i\ \mu_i^t$
- ϵ-greedy : avec proba ϵ, uniformément sur $\mathcal{A}$; avec proba $1-\epsilon$, $argmax_i\ \mu_i^t$

- Roulette : probabilité de chaque action proportionnelle à la récompense moyenne estimée, $p(a_i) = \frac{\mu_i^t}{\sum_i \mu_i^t}$
- Softmax : $p(a_i) = \frac{e^{\mu_i^t/\tau}}{\sum_i e^{\mu_i^t/\tau}}$
- UCB : $argmax\ \mu_i^t + \sqrt{\frac{2\ logt}{T_i^t}}$

Il est courant d'initialiser les algorithmes en jouant d'abord au hasard les actions qui n'ont jamais été jouées jusqu'à ce que toutes les actions aient été testées au moins une fois.
Dans la suite, votre politique contrôlera le joueur 1. Le joueur 2 sera contrôlé par l'algorithme Roulette, sur une distribution que vous fixerez : en particulier, si la distribution est uniforme, cela équivaut à un tirage uniforme d'action ; si la distribution est déterministe (une seule proba non nulle), l'action sera déterministe.

Dans toute la suite, les récompenses estimées μ_i pour chaque action i seront stockées dans un dictionnaire `dist`, dont les clés sont les actions et les valeurs μ_i ; T_i également dans un dictionnaire `nbtimes`.
Coder les fonctions `greedy(dist,nbtimes), uniforme(dist,nbtimes), epsgreedy(dist,nbtimes)` (on passe `nbtimes` pour des raisons de compatibilité même s'il n'est pas utilisé).
Le bout de code suivant vous permet de tester votre politique (les dictionnaires sont mis à jour par le contrôleur directement, vous n'avez rien à coder de supplémentaire que vos fonctions de choix d'action) :

```
#Creation d un jeu
chifumi = Chifumi()
# Creation d un controleur lie au jeu
ctrl = ChifumiRL(chifumi)
# Creation d une politique d exploitation/exploration lie au controleur
politique = EvePolitique(ctrl)
# specifier la methode de selection de l adversaire
adv_dist = dict({1:0.1,2:0.2,3:0.5,4:0.1,5:0.1})
chifumi.adv = Random(dist = adv_dist,algo = roulette)

# specifier la methode de selection de la politique
politique.algo=uniforme #ou greedy, epsgreedy,...

# Jouer pendant 500 tours en affichant
politique.run(maxiter=500,display=True)
# recuperer les logs du jeu sous la forme [(c1,c2,win,reward)] avec win =1 si  joueur 1 gagne, -1 j
# et reward 0,0.5 ou 1
log = np.array(politique.get_log())
```

Coder les autres fonctions et expérimenter. Tracer en particulier le score cumulé en fonction du temps pour chaque algorithme, également les courbes du nombre de fois où chaque numéro est joué. Tester sur différentes distributions pour l'adversaire (distribution uniforme, choix déterministe, ...). Quelle(s) caractéristique(s) sur la distribution de l'adversaire permettent de contrôler la difficulté de la tâche ?

Programmation dynamique

La classe `BlockWorld` représente un jeu de labyrinthe. La récompense dans le jeu du labyrinthe est définie de la manière suivante : +100 si c'est la case de la récompense, −100 si la case d'arrivée est un trou, -1 sinon. L'état est représenté par le couple (i, j) indiquant les coordonnées de l'agent dans la grille ($(0, 0)$ en haut à gauche).
Dans la suite, un MDP sera modélisé par un dictionnaire de dictionnaires tel que `mdp[state][action][state']` représente la probabilité de transition de `state` vers `state'` en prenant l'action `action`. On supposera que :

- `mdp.keys()` permet d'avoir tous les états du mdp ;
- `mdp[state].keys()` donne l'ensemble des actions possibles dans un état ;
- `mdp[state][action].keys()[0]` dans le cas déterministe donne l'état d'arrivée en choisissant l'action `action` à partir de l'état `state`. (`mdp[state][action] == {state':1.}`, le dictionnaire associé à une action est composée d'une seule clé, l'état d'arrivée). d'arrivée).

Coder l'algorithme de policy iteration. Pour cela, compléter la classe `PolicyIteration`. Elle contient déjà :

- le modèle du mdp (`self.mdp`)
- un dictionnaire représentant la politique `self.pi` (clé : état, valeur : action)
- un dictionnaire représentant la fonction de valeur V_π `self.v` (clé : état, valeur : réel)
- une méthode `self.get_action(state)` qui renvoie l'action selon la politique courante pour l'état `state` (en fait `self.pi[state]`)
- une méthode `self.get_next_state(state,action)` qui renvoie l'état d'arrivée correspondant au choix de `action` pour l'état `state`
- une méthode `self.get_reward(state,action,state')` (en fait au niveau du contrôleur, qui existe également au niveau de la politique) qui permet d'obtenir la récompense pour un triplet `state,action,state'`
- une méthode `self.optim(maxiter,gamma)` qui appelle `maxiter` fois la méthode `self.next_step(gamma)`, qui appelle successivement `next_v` (évaluation de politique) et `next_pi` (amélioration de politique).

Il ne vous reste qu'à compléter les méthodes `self.next_v` et `self.next_pi` qui réalisent une itération d'évaluation de politique (opérateur de Bellman $(T^\pi V)(s_t) = r(s_t, \pi(s_t)) + \gamma V(s_{t+1}))$ et une d'amélioration de politique ($\pi(s_t) = argmax_a\ r(s_t, a) + \gamma V_\pi(s_{t+1})$), s_{t+1} étant l'état suivant s_t selon la politique π en choisissant donc l'action $\pi(s_t)$.
Afin de tester, vous pouvez utiliser la méthode `self.campagne(display=True,maxiter,maxiterrun)` qui lance `maxiter` parties de `maxiterrun` pas. Visualiser les fonctions de valeurs apprises, expérimenter selon différentes valeurs de γ.

QLearning et Monte Carlo

La classe `Interactive` propose un squelette pour implémenter soit un algo de QLearning, soit de Monte-Carlo, en offline ou online/off-policy ou on-policy. Elle dispose en particulier :

- d'un dictionnaire `nbtimes` tel que `self.nbtimes[state][action]` dénote le nombre de fois où l'action a été explorée à partir de l'état
- d'un dictionnaire `self.q` tel que `self.q[state][action]` représente la Q-value du couple (état,action)
- d'un champ `self.choice` qui permet de fixer l'algorithme d'exploration/exploitation pour le choix des actions en fonctions des récompenses et du nombre de fois où l'action a été jouée, utilisée dans la méthode `get_action` (du type greedy, ϵ-greedy, UCB, cf section précédente)
- d'une méthode `update(state_src,action,state_dest,reward` qui est appelée à la fin de chaque pas en cours de jeu

- d'une méthode `update_scenario(log)` appelée à la fin de chaque partie, `log` contenant la liste des quadruplets (état,action,état d'arivée, récompense) de la partie.

Compléter `update` en implémentant un algorithme de `QLearning`. Compléter `update_scenario` en implémentant un algorithme de Monte-Carlo.