Travail pratique 2 : IFT-4102/7025, Hiver 2022
Apprentissage par renforcement

Université Laval, Département d'informatique et de génie logiciel

Date de remise: 27 mars 2022 à 23h59

Dans ce projet, vous allez implémenter l'algorithme de l'itération de la valeur et l'algorithme Q-learning. Vous allez d'abord tester vos agents sur le "Gridworld", puis les appliquer à un contrôleur de robot simulé ("Crawler") et à Pacman.

Comme dans le projet précédent, ce projet comprend un autograder permettant de tester vos solutions sur votre machine. Il peut être exécuté sur toutes les questions avec la commande:

```
python autograder.py
```

Il peut être exécuté pour une question particulière, telle que q2, avec:

```
python autograder.py -q q2
```

Ce projet a été développé par l'université Berkeley dans le cadre du cours CS188. Les questions et instructions qui suivent sont d'ailleurs essentiellement une traduction de la page web http://ai.berkeley.edu/reinforcement.html que vous pouvez aussi consulter.

Le code de ce projet se compose de plusieurs fichiers Python (reinforcement.zip), certains devront être lus et compris (au moins partiellement), et d'autres peuvent être ignorés.

Fichiers que vous allez modifier et remettre:

valueIterationAgents.py	Un agent d'itération de la valeur pour la résolution de MDP connus.
qlearningAgents.py	Agents Q-learning pour Gridworld, Crawler et Pacman.
analysis.py	Un fichier pour mettre vos réponses aux questions posées dans le projet.

Fichiers que vous pourriez vouloir regarder:

mdp.py	Définit les méthodes sur les MDP généraux.
learningAgents.py	Définit les classes de base ValueEstimationAgent et QLearningAgent.
util.py	Utilitaires.
gridworld.py	L'implémentation de Gridworld.
featureExtractors.py	Classes d'extraction de caractéristiques sur des paires (état,action).

Les fichiers de support que vous pouvez ignorer:

environment.py	Classe abstraite pour les environnements.
graphicsGridworldDisplay.py	Affichage graphique Gridworld.
graphicsUtils.py	Utilitaires pour les graphiques.
textGridworldDisplay.py	Plug-in pour l'interface texte Gridworld.
crawler.py	Le code du robot crawler.
graphicsCrawlerDisplay.py	Interface graphique pour le robot crawler.
autograder.py	Correcteur automatique
testParser.py	Analyse les fichiers de tests et de solutions pour l'autograder.
testClasses.py	Classes de tests générales.
`test_cases`	Répertoire contenant les cas de tests pour chaque question.
reinforcementTestClasses.py	Les classes de tests spécifiques au présent projet.

Évaluation et remise: Seulement les fichiers valueIterationAgents.py, qlearningAgents.py et analysis.py sont à remettre. Il ne faut pas modifier les autres fichiers car votre code sera exécuter avec les originaux lors de la correction. Un correcteur automatique est fourni, ce qui vous permettra de tester vos implémentations. Veuillez ne pas changer les noms des fonctions ou classes fournies dans le code, sinon le correcteur automatique ne fonctionnera plus.

Nous vérifierons votre code par rapport aux autres soumissions dans la classe pour la redondance logique. Si vous copiez le code de quelqu'un d'autre et le soumettez avec des modifications mineures, nous le saurons. Ces détecteurs de triche sont assez difficiles à tromper, alors n'essayez pas s'il vous plaît. Nous vous faisons confiance pour soumettre votre propre travail seulement. De plus, puisque ce projet Pacman a été utilisé dans divers cours dans différentes universités au cours des années, beaucoup d'étudiants l'ont fait, et certains d'entre eux ont possiblement rendu publiques leurs solutions. Merci de ne pas plagier ces solutions.

Nous sommes maintenant prêts à commencer! Après avoir téléchargé le code, décompressé, et en changeant dans le répertoire, vous devriez être capable de lancer le Gridworld en mode de contrôle manuel

```
python gridworld.py -m
```

Le point bleu est l'agent. Notez que lorsque vous appuyez vers le haut, l'agent ne se déplace réellement vers le nord que 80% du temps. Telle est la vie d'un agent du Gridworld! Vous pouvez contrôler de nombreux aspects de la simulation. Une liste complète des options est disponible en lançant:

```
python gridworld.py -h
```

L'agent par défaut se déplace de manière aléatoire:

```
python gridworld.py -g MazeGrid
```

Remarques: Le MDP Gridworld est tel que vous devez d'abord entrer dans un état pré-terminal (les doubles cases affichées dans l'interface graphique), puis effectuer l'action spéciale «quitter» pour que l'épisode se termine (dans l'état terminal appelé TERMINAL_STATE, qui n'est pas affiché dans l'interface graphique). Si vous exécutez un épisode manuellement, votre rendement total sera peut-être plus bas que ce que vous vous attendiez en raison du facteur de discompte (-d pour le modifier; 0,9 par défaut). Comme pour Pacman, les positions sont représentées par (x,y) les coordonnées cartésiennes et tous les tableaux sont indexés par [x][y], 'north' étant la direction d'augmentation en y, etc. Par défaut, la plupart des transitions recevront une récompense de zéro, bien que vous puissiez changer cela avec l'option de récompense (-r).

1 Itération de la valeur [6 points]

Écrivez un agent d'itération de la valeur en complétant le code dans ValueIterationAgents.py. Votre agent d'itération de la valeur est un planificateur hors ligne et non un agent d'apprentissage par renforcement. Au cours de sa phase de planification initiale, l'agent utilise l'algorithme de l'itération de la valeur pour un certain nombre d'itérations pour résoudre le MDP connu. Il peut ensuite suivre la politique résultante. Implémentez aussi les deux méthodes suivantes:

1. computeActionFromValues(state); calcule la meilleure action en fonction des valeurs obtenues par l'itération de la valeur (données par self.values).

2. computeQValueFromValues(state, action); renvoie la Q-valeur de la paire (state, action) en utilisant les valeurs données par self.values.

Ces quantités sont toutes affichées dans l'interface graphique: les valeurs sont les nombres dans les carrés, les Q-valeurs sont les nombres dans les quarts de carrés et la politique est donnée par les flèches sortantes de chaque carré.

Note importante: Utilisez la version "batch" de l'itération de la valeur où chaque vecteur V_k est calculé à partir d'un vecteur fixe V_{k-1} et non la version "en ligne" où chaque vecteur est mis à jour sur place. Cela signifie que lorsque la valeur d'un état est mise à jour dans l'itération k sur la base des valeurs de ses états successeurs, les valeurs des états successeurs utilisées dans ce calcul doivent être celles de l'itération k-1 (même si certains des états successeurs ont déjà été mis à jour lors de l'itération k).

Remarque: Assurez-vous de gérer le cas où un état ne dispose d'aucune action (réfléchissez à ce que cela signifie pour les récompenses futures).

Pour tester votre implémentation, lancez l'autograder:

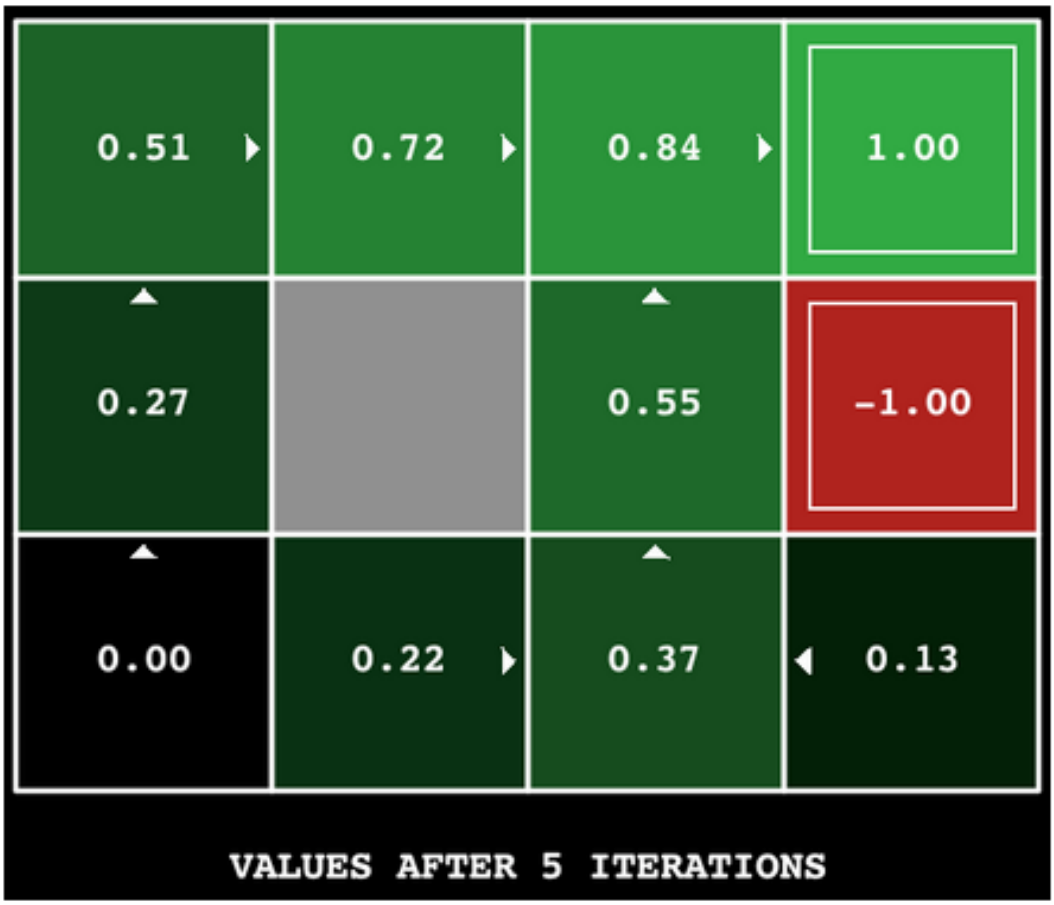
python autograder.py -q q1

La commande suivante charge votre agent, qui va calculer une politique et l'exécuter 10 fois. Appuyez sur une touche pour faire défiler les valeurs, les Q-valeurs et ensuite la simulation. Vous devriez constater que la valeur de l'état de départ (que vous pouvez lire à partir de l'interface graphique) et la récompense moyenne empirique (après les 10 tours d'exécution) sont assez proches.

python gridworld.py -a value -i 100 -k 10

Sur le gridworld par défaut, utiliser 5 itérations devrait vous donner cette sortie:

python gridworld.py -a value -i 5



2 Traversée du pont [1 point]

"BridgeGrid" est une grille avec un état terminal à faible récompense et un état terminal à récompense élevée séparés par un pont étroit. L'agent commence près de l'état de faible récompense. Avec le facteur de discompte par défaut de $0,9$ et le bruit par défaut de $0,2$ pour les actions, la stratégie optimale ne traverse pas le pont. Ne changez qu'un de ces deux paramètres de sorte que la stratégie optimale pousse l'agent à tenter de traverser le pont. Mettez votre réponse dans question2() de analysis.py. Le bruit fait référence à la fréquence à laquelle un agent se retrouve dans un état successeur non prévu lorsqu'il effectue une action. La valeur par défaut correspond à:

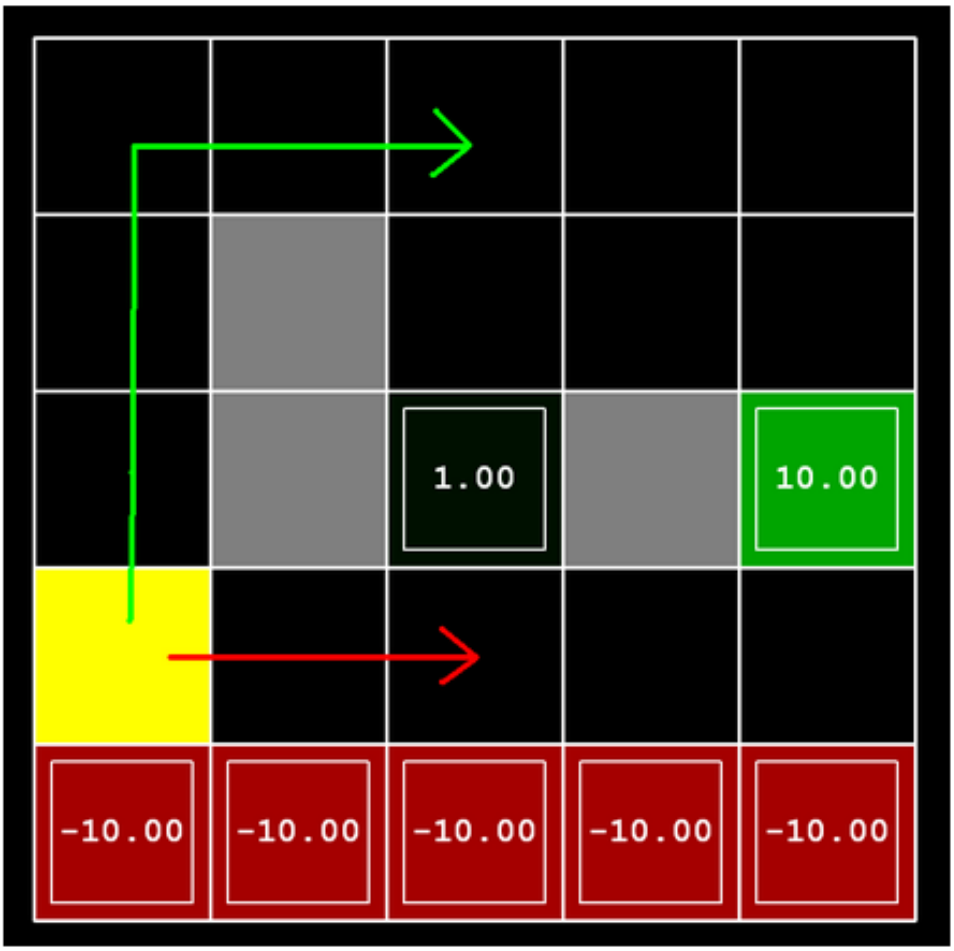
python gridworld.py -a value -i 100 -g BridgeGrid -d 0.9 -n 0.2

Nous vérifierons que vous n'avez modifié que l'un des paramètres donnés et qu'avec cette modification, un agent d'itération de la valeur correct devrait franchir le pont. Pour vérifier votre réponse, lancez le correcteur automatique:

python autograder.py -q q2

3 Politiques [5 points]

Considérez la grille "DiscountGrid", illustrée ci-dessous. Cette grille a deux états terminaux avec une récompense positive (dans la rangée du milieu), une sortie proche avec une récompense de $+1$ et une sortie lointaine avec une récompense de $+10$. La rangée inférieure de la grille comprend les états terminaux avec récompenses négatives (indiqués en rouge); chaque état dans cette région de "falaise" a une valeur de -10. L'état de départ est le carré jaune. Nous distinguons deux types de chemins: (1) les chemins qui "risquent la falaise" et se déplacent près de la rangée du bas de la grille; ces chemins sont plus courts mais risquent de générer une récompense négative importante et sont représentés par la flèche rouge dans la figure ci-dessous. (2) les chemins qui "évitent la falaise" et longent le bord supérieur de la grille. Ces chemins sont plus longs mais risquent moins d'engendrer des retombées négatives considérables. Ces chemins sont représentés par la flèche verte dans la figure ci-dessous.



Dans cette question, vous allez choisir les paramètres de discompte, de bruit et de récompense pour rester en vie (living reward) pour ce MDP afin de produire des politiques optimales de plusieurs types. Si un comportement n'est pas réalisable renvoyez le string 'NOT POSSIBLE' .

Voici les types de stratégies optimales que vous devez essayer de produire:

a) Préfère la sortie proche (+1), risquant la falaise (-10)

b) Préfère la sortie proche (+1), mais en évitant la falaise (-10)

c) Préfère la sortie lointaine (+10) au risque de la falaise (-10)

d) Préfère la sortie lointaine (+10), en évitant la falaise (-10)

e) Évite les deux sorties positives et la falaise (un épisode ne devrait donc jamais se terminer)

Les question3a() à question3e() doivent chacune renvoyer 3 éléments (discompte, bruit, living reward) dans analysis.py.

Pour vérifier vos réponses, lancez le correcteur automatique:

```
python autograder.py -q q3
```

4 Q-learning [4 points]

Notez que votre agent d'itération de la valeur n'apprend pas réellement par l'expérience. Au contraire, il résoud son modèle de MDP pour parvenir à une politique complète avant d'interagir avec un environnement réel. Lorsqu'il interagit avec l'environnement, il suit simplement la politique précalculée (il devient un agent réflexe). Cette distinction peut être subtile dans un environnement simulé tel qu'un Gridword, mais elle est très importante dans le monde réel, où le véritable MDP n'est pas disponible.

Vous allez maintenant coder un agent Q-learning, qui apprend à partir d'interactions avec l'environnement via sa méthode update(state, action, nextState, reward). Pour cette question, vous devez implémenter les méthodes update, computeValueFromQValues, getQValue et computeActionFromQValues dans le fichier qlearningAgents.py.

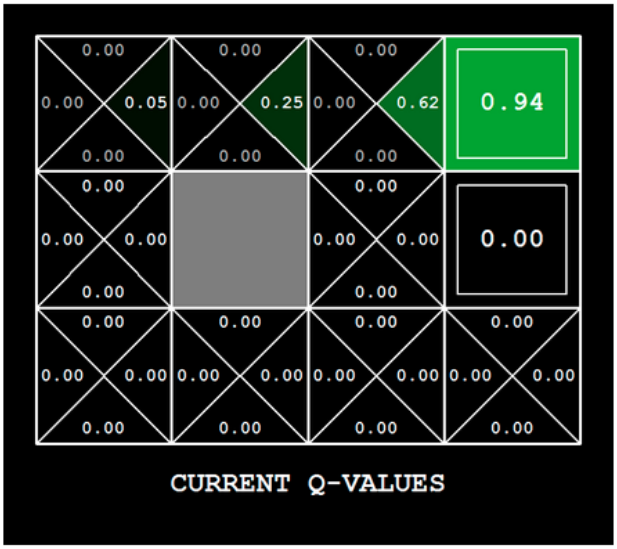
Remarque: Pour computeActionFromQValues, vous devez briser les égalités de manière aléatoire. La fonction random.choice() aidera. Dans un état particulier, les actions que votre agent n'a pas encore vues ont toujours une Q-valeur égale à zéro, et si toutes les actions que votre agent a vues auparavant ont une Q-valeur négative, une action pas encore vue peut être optimale.

Important: Assurez-vous que dans les fonctions computeValueFromQValues et computeActionFromQValues, vous n'accédez qu'aux Q-valeurs que via getQValue. Cette abstraction sera utile pour la question 8 lorsque vous redéfinirez getQValue pour utiliser des caractéristiques (features) des paires état-action plutôt que les paires état-action directement.

Avec la mise à jour du Q-learning en place, vous pouvez regarder votre agent apprendre sous contrôle manuel, à l'aide du clavier:

python gridworld.py -a q -k 5 -m

Rappelez-vous que -k contrôlera le nombre d'épisodes que votre agent va utiliser pour apprendre. Observez comment l'agent apprend sur l'état dans lequel il se trouvait précédemment et pas sur celui dans lequel il tombe. Astuce: pour aider au débuggage, vous pouvez désactiver le bruit en utilisant le paramètre –noise 0.0 (bien que cela rend évidemment le Q-learning moins intéressant). Si vous dirigez manuellement Pacman vers le nord, puis vers l'est le long du chemin optimal pour quatre épisodes, vous devriez voir les Q-valeurs suivantes:



Pour vérifier vos réponses, lancez le correcteur automatique:

python autograder.py -q q4

5 Exploration epsilon-greedy [3 points]

Complétez votre agent Q-learning en implémentant la sélection d'actions epsilon-greedy dans getAction, ce qui signifie que l'agent choisit des actions aléatoires une fraction epsilon du temps et qu'il suit ses meilleures actions selon les Q-valeurs actuelles le reste du temps. Notez que le choix d'une action aléatoire peut entraîner le choix de la meilleure action, c'est-à-dire que vous devez choisir une action aléatoire parmi toutes les actions légales.

python gridworld.py -a q -k 100

Vos Q-valeurs finales devraient ressembler à celles de votre agent d'itération de la valeur, surtout le long des chemins les plus empruntés. Toutefois, elles seront inférieures en raison des actions aléatoires et de la phase d'apprentissage initiale.

Vous pouvez simuler une variable aléatoire binaire avec une probabilité de succès p en utilisant util.flipCoin(p), qui renvoie True avec la probabilité p et False avec la probabilité 1-p.

Pour tester votre implémentation, lancez l'autograder:

python autograder.py -q q5

Sans code supplémentaire, vous devriez maintenant être en mesure de lancer le robot crawler Q-apprenant:

python crawler.py

Si cela ne fonctionne pas, vous avez probablement écrit du code trop spécifique au problème GridWorld et vous devriez le rendre plus général pour tous les MDP.

6 Traversée du pont revisitée [1 point]

Tout d'abord, entraînez un Q-apprenant avec des actions complètement aléatoires ($epsilon = 1$) en utilisant le taux d'apprentissage par défaut sur la grille BridgeGrid sans bruit et durant 50 épisodes et observez s'il trouve la stratégie optimale:

python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1

Essayez maintenant la même expérience avec un epsilon égal à 0. Existe-t-il un epsilon et un taux d'apprentissage pour lesquels il est très probable (supérieur à 99%) que la politique optimale soit apprise après 50 itérations? question6() dans analysis.py doit renvoyer soit un tuple de 2 éléments (epsilon, learning rate) ou la chaîne 'NOT POSSIBLE' s'il n'y en a pas. Epsilon est contrôlé par -e et le taux d'apprentissage par -l.

Pour tester votre réponse, lancez l'autograder:

python autograder.py -q q6

7 Q-learning et Pacman [1 point]

Il est temps de jouer à Pacman! Pacman jouera en deux phases. Au cours de la première phase d'entraînement, Pacman commencera par apprendre les valeurs des paires (états,actions). Parce qu'il faut beaucoup de temps pour apprendre des Q-valeurs précises, même pour de petites grilles, les parties d'entraînement de Pacman fonctionnent par défaut en mode silencieux, sans affichage graphique. Une fois l'entraînement terminé, Pacman entrera en mode test. Lors des tests, les self.epsilon et self.alpha seront fixés à 0, ce qui mettra fin à l'apprentissage et à l'exploration, afin de permettre à Pacman d'exploiter sa politique apprise. Par défaut, les parties de test sont affichées dans l'interface graphique. Sans aucun changement sur votre code, vous devriez pouvoir exécuter le Q-learning pour Pacman sur de très petites grilles comme suit:

python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid

Notez que PacmanQAgent est déjà défini pour vous en termes du QLearningAgent que vous avez déjà implémenté. PacmanQAgent n'est différent que par le fait qu'il dispose de paramètres d'apprentissage par défaut plus efficaces pour le jeux de Pacman ($epsilon = 0.05, alpha = 0.2,$ $gamma = 0.8$). Vous recevrez tout le crédit pour cette question si la commande ci-dessus fonctionne sans exception et si votre agent gagne au moins 80% du temps. L'autograder lancera 100 parties de test après les 2000 parties d'entraînement.

Aide: si votre QLearningAgent fonctionne pour gridworld.py et crawler.py mais ne semble pas apprendre une bonne politique pour Pacman sur smallGrid, cela peut être dû au fait que vos getAction et / ou computeActionFromQValues ne considèrent pas correctement les actions non vues jusqu'à maintenant. En particulier, comme les actions non vues jusqu'à maintenant ont par définition une Q-valeur égale à zéro, si toutes les actions observées ont des Q-valeurs négatives, une action non vue jusqu'à maintenant peut être optimale. Méfiez-vous de la fonction argmax d'util.Counter!

Pour tester votre code, lancez l'autograder:

python autograder.py -q q7

Remarque: Si vous souhaitez expérimenter avec des paramètres d'apprentissage différents, vous pouvez utiliser l'option -a , par exemple -a epsilon=0.1,alpha=0.3,gamma=0.7 . Ces valeurs seront alors accessibles en tant que self.epsilon, self.gamma et self.alpha pour l'agent.

Remarque: Alors qu'un total de 2010 parties seront jouées, les 2000 premières parties ne seront pas affichées à cause de l'option -x 2000, qui désigne les 2000 premières parties pour l'entraînement (pas de sortie). Ainsi, vous ne verrez que Pacman jouer les 10 dernières parties. Le nombre de parties d'entraînement peut également être transmis à votre agent avec l'option numTraining . Si vous souhaitez regarder 10 parties d'entraînement pour voir ce qui se passe, utilisez la commande:

python pacman.py -p PacmanQAgent -n 10 -l smallGrid -a numTraining=10

Pendant l'entraînement, vous verrez à tous les 100 matchs des statistiques sur les résultats des parties. Epsilon étant positif pendant l'entraînement, Pacman jouera plutôt mal même après avoir appris une bonne politique: c'est parce qu'il fait de temps en temps un déplacement aléatoire qui lui fera percuter un fantôme. Il devrait nécessiter entre 1 000 et 1 400 matchs avant que les récompenses moyennes de Pacman pour un segment de 100 épisodes ne deviennent positives, ce qui indique qu'il a commencé à gagner plus de parties qu'il en perd. À la fin de l'entraînement, cette moyenne de récompenses devrait rester positive et être assez élevée (entre 100 et 350).

Assurez-vous de bien comprendre ce qui se passe ici: l'état du MDP correspond à la configuration exacte du jeu à laquelle Pacman fait face, avec les transitions qui décrivent toute une série de modifications apportées à cet état. Les configurations de jeu intermédiaires dans lesquelles

Pacman a été déplacé mais les fantômes n'ont pas répondu ne sont pas des états du MDP, mais sont intégrées aux transitions.

Une fois que Pacman aura terminé son entraînement, il devrait gagner de manière très fiable dans les parties de test (au moins 90% du temps), car il exploite maintenant sa politique apprise. Cependant, vous pourrez constater que l'entraînement du même agent sur le mediumGrid ne fonctionne pas bien. Dans notre implémentation, les récompenses moyennes de Pacman restent négatives tout au long de l'entraînement. Au moment des tests, Pacman joue plutôt mal, perdant probablement tous ses matchs. L'entraînement prendra également beaucoup de temps malgré son inefficacité. Pacman ne parvient pas à gagner sur des layouts plus larges car chaque configuration est un état distinct avec ses propres Q-valeurs. Il n'a aucun moyen de généraliser et de comprendre par exemple que percuter un fantôme est mauvais pour toutes les positions. Cette approche ne scale donc pas très bien à de larges espaces d'états. Le Q-learning approximatif vise à remédier à ces problèmes.

8 Q-learning approximatif [3 points]

Implémentez un agent Q-learning approximatif qui attribue des poids aux caractéristiques (features) des états, de nombreux états pouvant partager les mêmes caractéristiques. Ecrivez votre code dans la classe ApproximateQAgent du fichier qlearningAgents.py.

La Q-fonction approximative prend la forme suivante:

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a) w_i,$$

où chaque poids w_i est associé à une certaine caractéristique $f_i(s,a)$. Les caractéristiques $f_i(s,a)$ (qui sont des fonctions de paires (état, action)) sont fournies pour vous dans featureExtractors.py. Ces caractéristiques peuvent être obtenues sous la forme d'un util.Counter avec self.featExtractor.getFeatures(state,action). Vos poids devraient aussi similairement être contenus dans un objet de type util.Counter.

La mise à jour des poids s'opère de la manière suivante:

$$difference = (r + \gamma \max_{a'} Q(s',a')) - Q(s,a)$$
$$w_i \longleftarrow w_i + \alpha \cdot (difference) \cdot f_i(s,a),$$

où r est la récompense obtenue et $difference$ est le même terme que dans le Q-learning standard.

Par défaut, ApproximateQAgent utilise l'IdentityExtractor, qui attribue un feature à chaque paire (state,action). Avec cet extracteur de features, votre agent Q-learning approximatif devrait fonctionner de la même manière que PacmanQAgent. Vous pouvez tester cela avec la commande suivante:

python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid

Important: La classe ApproximateQAgent est une sous-classe de QLearningAgent. Par conséquent, elle hérite de plusieurs méthodes telles que getAction. Assurez-vous que vos méthodes dans QLearningAgent appellent getQValue au lieu d'accéder directement aux Q-valeurs, afin que, lorsque vous redéfinissez getQValue dans votre agent approximatif, les nouvelles Q-valeurs approximatives soient utilisées pour déterminer les actions.

Exécutez maintenant votre agent Q-learning approximatif avec l'extracteur de caractéristiques SimpleExtractor, qui permet à Pacman d'apprendre à gagner facilement:

python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid

Même des jeux beaucoup plus larges ne devraient poser aucun problème pour votre ApproximateQAgent (avertissement : cela peut prendre quelques minutes).

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l
mediumClassic
```

Si vous ne commettez aucune erreur, votre agent Q-learning approximatif devrait gagner presque à chaque fois même avec seulement 50 parties d'entraînement.

Pour tester votre code, lancez l'autograder:

```
python autograder.py -q q8
```

Toutes nos félicitations! Vous avez maintenant un agent Pacman apprenant!