Université Claude Bernard Lyon 1
Département Informatique

SMA & IA

TP1. Planification stochastique & MDP

Laëtitia Matignon

Pour ce TP, vous devez :

- récupérer le TP sur `https://github.com/laetitiamatignon/tp-ia-mdp` ;
- créer un **dépôt privé** avec les sources du TP sur votre compte `github`, et ajouter l'enseignante (username `laetitiamatignon`) comme collaboratrice de votre dépôt.

Les éléments à compléter/rendre pour le TP1 sont :

- la classe `ValueIterationAgent.java` à compléter ;
- les réponses aux questions de la partie 5 (valeurs des paramètres et justification) à mettre dans le fichier `RAPPORT.md` à co mpléter OU dans un document **au format pdf** à ajouter au dépôt.

L'objectif du TP est d'implémenter l'algorithme de planification *value iteration* et d'étudier l'influence des différents paramètres sur la politique optimale obtenue.

1 Mise en place du TP

Le dépôt `github` contient :

- ./lib/IAMDPLib.jar : bibliothèque pour affichage graphique et mdp
- ./javadoc : javadoc de la bibliothèque IAMDPLib
- dans le dossier `src`, le package `agent.planningagent` contenant le fichier `ValueIterationAgent.java` que vous devez compléter
- dans le dossier `simu`, le package `simuTP1` contenant les fichiers java pour tester l'algorithme dans différents environnements.

Le diagramme de classes de ce code est donné en annexe de cet énoncé de TP.

Le code et les librairies fournies dépendent de JavaFX, vous devez donc utiliser JavaSE8+JavaFX. Le JDK8 est téléchargeable sur `http://jdk.java.net/8/` ou sur `https://perso.liris.cnrs.fr/laetitia.matignon/index/5ASMA/install`.

Pour démarrer le TP (sous eclipse) :

1. créer un nouveau projet Java vide
2. copiez-collez dans le dossier de votre projet tous les dossiers de l'archive
3. faites un *refresh* de votre projet
4. ajoutez au **Build Path** de votre projet la librairie `IAMDPLib.jar` (clic droit sur votre projet, puis *Properties*, *Java Build Path*, *Lirairies*, *Add Jar*).
5. ajoutez au **Build Path** de votre projet la javadoc (clic droit sur votre projet, puis *Properties*, *Java Build Path*, *Lirairies*, spécifier la *Javadoc location* en donnant le chemin vers le dossier `javadoc`).
6. Ajouter le dossier `simu` à votre projet ((clic droit sur votre projet, puis *Properties*, *Java Build Path*, *Source*, *Add Folder*, ajouter le dossier `simu`)

2 Environnement déterministe ou stochastique

Exécutez la classe `testMoveGridworld.java` du package `simulation`. Vous pouvez contrôler un agent via les touches du clavier. **Il faut pour cela appuyer sur Enter pour ne plus avoir le focus dans la zone de texte.** L'agent est représenté par un rectangle bleu, les états absorbants par un rectangle rouge avec la valeur de récompense à l'intérieur, et les murs par une case grise.

Question 1 *Changer de labyrinthe en utilisant les méthodes statiques de la classe `GridworldMDP`.*

La valeur du paramètre `bruit` correspond à la probabilité que l'agent n'aille pas dans la direction voulue. Par exemple, avec une valeur de x pour le bruit, l'agent qui exécute l'action NORD aura une probabilité de $1-x$ d'aller dans la case située au NORD de sa case actuelle, et ira dans la case située à l'EST de sa case actuelle avec une probabilité de $x/2$ et dans la case située à l'OUEST de sa case actuelle avec une probabilité de $x/2$. **La prise en compte du paramètre modifié dans le JTextField ne se fait qu'une fois la touche Entrée utilisée dans le JTextField.**

Question 2 *Vérifiez, en observant le comportement de l'agent lorsque vous le déplacez, qu'un bruit égal à 0 correspond bien à un environnement déterministe et qu'un bruit supérieur à 0 correspond bien à un environnement stochastique.*

3 Agent aléatoire

Exécutez la classe `testRandomAgent.java` du package `simulation`. Vous obtenez la vue associée à tout agent de type `PlanningValueAgent`. Pour plus de détails sur cette vue, voir l'annexe 6.1.

Tout agent de type `PlanningValueAgent` possède un attribut `mdp` à partir duquel l'agent peut récupérer l'ensemble des actions autorisées dans un état, l'ensemble des états

accessibles de l'environnement, la fonction de récompense, la fonction de transition ... (se référer à la javadoc pour l'ensemble des fonctions proposées par la classe `MDP`).

On s'intéresse tout d'abord à un agent aléatoire. Cet agent ne planifie pas ; la valeur de chacun des états est donc toujours 0. Par contre, lorsqu'un épisode est lancé, cet agent exécute une action aléatoire parmi toutes les actions possibles dans son état courant.

Question 1 *Lisez et comprenez le code proposé pour un agent aléatoire qui est dans la classe* `AgentRandom` *du package* `agent.planningagent`*. Testez votre agent aléatoire en exécutant la classe* `testRandomAgent.java` *du package* `simulation`*.*

4 Algorithme *value iteration*

On souhaite maintenant implémenter un agent qui planifie hors-ligne sa politique avec l'algorithme *value iteration*.

L'algorithme *Value iteration* [Bellman,1957] calcule itérativement la fonction de valeur optimale V^* à partir du modèle MDP.

- Initialisation arbitraire de $V_0(s) \forall s$
- Mise à jour de $V_k(s)$ en utilisant les valeurs estimées à $k-1$ des états accessibles s' suite à une action a :

$$\forall s \in S \quad V_k(s) \leftarrow \max_{a \in A} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_{k-1}(s')]$$

- Répète jusqu'à convergence :
  critère d'arrêt $\delta = \max_{s \in S} |V_k(s) - V_{k+1}(s)| < \epsilon$

On peut ensuite extraire la politique π_k à partir de V_k en calculant la politique gloutonne :

$$\forall s \in S \quad \pi_k(s) = \arg\max_{a \in A} \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

Question 1 *Complétez la classe* `ValueIterationAgent` *du package* `agent.planningagent` *(endroit où il y a //*** VOTRE CODE).*

Question 2 *Vérifiez votre algorithme en utilisant la classe* `testValueIterCours` *et en vérifiant que les valeurs calculées au cours des premières itérations sont correctes (utilisez le labyrinthe* `BookGrid` *qui est l'exemple vu en cours).*

5 Influence des paramètres

Maintenant que votre algorithme fonctionne correctement, on va s'intéresser à l'influence des différents paramètres sur la politique optimale calculée. Pour cela, on va utiliser d'autres labyrinthes.

5.1 BridgeGrid

Dans l'environnement *BridgeGrid* (`testValueIterationBridge`), avec les valeurs par défaut de γ à 0.9 et du bruit à 0.1, la politique optimale ne permet pas à l'agent de traverser le pont (i.e. d'atteindre la récompense maximale de 10).

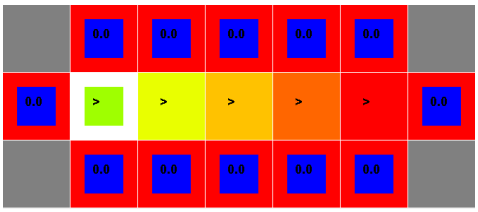

FIGURE 1 – Politique optimale permettant à l'agent de traverser le pont dans l'environnement *bridge grid*.

Question 1 *Changez **un seul des deux paramètres**, soit γ soit le bruit, de sorte à ce que la politique optimale permette à l'agent de traverser le pont (cf. figure 1).*

5.2 DiscountGrid

Dans l'environnement *DiscountGrid* (`testValueIterationDiscountGrid`), on distingue deux types de chemin représentés sur la figure 2 (image de gauche) :

- des chemins courts mais risqués qui passent près de la ligne du bas ; ces chemins sont représentés par la flèche rouge ;
- des chemins plus longs mais sûrs qui passent par le haut du labyrinthe ; ces chemins sont représentés par la flèche verte.

Question 2 *En partant de valeurs initiales $\gamma = 0.9$, bruit=0.2, $r_{other} = 0.0$, vous devez obtenir une politique optimale qui suit un chemin sûr pour atteindre l'état absorbant de récompense +10 (cf. figure 2, image de droite).*

*En partant de ces valeurs initiales, changez **un seul des trois paramètres**, soit γ, soit le bruit, soit la récompense, de sorte à obtenir une politique optimale :*

1. *qui suit un chemin risqué pour atteindre l'état absorbant de récompense +1 ;*
2. *qui suit un chemin risqué pour atteindre l'état absorbant de récompense +10 ;*
3. *qui suit un chemin sûr pour atteindre l'état absorbant de récompense +1 ;*
4. *qui évite les états absorbants.*

Les réponses aux 2 questions de la partie 5, ainsi que la justification de vos choix pour les valeurs des paramètres, doivent être dans votre compte-rendu.

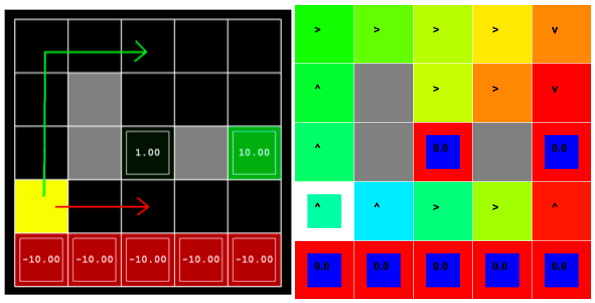

FIGURE 2 – Discount Grid : à gauche, chemins risqués (flèche rouge) et chemins sûrs (flèche verte) ; à droite, poitique optimale qui suit un chemin sûr pour atteindre l'état absorbant de récompense +10.

6 Annexes

6.1 Vue

La vue associée à tout agent de type `PlanningValueAgent` permet :.

- d'afficher dans le labyrinthe la valeur de chaque état (*vue valeurs*) et la politique (*vue politique*). La vue politique affiche pour chaque état ce que renvoie la méthode `getPolitique`, i.e. la ou les meilleures actions à suivre dans chaque état. La vue valeurs affiche pour chaque état ce que renvoie la méthode `getValeur`, i.e. la valeur de chaque état ;
- de planifier en effectuant un certain nombre de mises à jour de la fonction de valeur (bouton *lance nb iterations* qui exécute `updateV()` un certain de fois (*PlanningValueAgent : :run(int nbIterations)*)
- de planifier en effectuant la mise à jour de la fonction de valeur jusqu'à convergence (bouton *planifie jusqu'à convergence* qui exécute `updateV()` un certain de fois (*PlanningValueAgent : :run()*)) ;
- de modifier différents paramètres (la prise en compte du paramètre modifié ne se fait qu'une fois la touche Enter utilisée dans le JTextField).
- de lancer un Episode, c'est-à-dire que l'agent réalise des actions dans l'environnement jusqu'à ce qu'il atteigne un état absorbant ; l'action réalisée par l'agent dans chaque état où il se situe est celle qui est renvoyée par la méthode `getAction`.

6.2 Diagramme UML