Ecole Nationale d'Ingénieurs de Brest

Intelligence Artificielle

— Travaux Pratiques —

Apprentissage par renforcement : optimisation d'un algorithme d'apprentissage par différence temporelle

P. De Loor, M. Polceanu

– 2018 –

Table des matières

1 Apprentissage par renforcement, l'algorithme SARSA

1.1 Notes de cours sur l'apprentissage par renforcement

L'apprentissage par renforcement est un apprentissage de l'interaction entre un systéme et son environnement. Le systéme doit trouver l'action qu'il doit effectuer lorsqu'il est dans une situation donnée. L'apprentissage se base sur la métaphore de la carotte et du béton. L'environnement peut "punir" ou "récompenser" le systéme en fonction des actions que celui-ci a fait. Cependant il existe des différences importantes par rapport à de nombreux autres types d'apprentissage :

- l'environnement ne récompense (ou punit) pas systématiquement chaque action. Au contraire, dans la plupart des cas, il ne donne aucune indication au systéme. Ceci évite d'avoir à définir une fonction de "fitness", qui permettrait de dire si l'action a été bonne ou non (comme lors de l'utilisation d'algorithmes génétiques). Si l'on prend l'exemple du labyrinthe, on peut imaginer que la seule rétribution possible soit celle correspondant à l'arrivée à la porte de sortie.
- aucun exemple n'est à fournir au systéme, il ne s'agit donc pas d'apprentissage supervisé. Ce point peut toutefois étre nuancé par le fait qu'une récompense puisse étre vue comme la signification qu'un exemple, exploré par le systéme, est bon. Toute la différence est dans le fait que c'est le systéme qui explore les différentes possibilités et estime seul si il a appris ou non.
- Pour apprendre, l'algorithme répercute les rétributions sur les états et les actions précédentes. Chaque état ou chaque combinaison état-action posséde alors une valeur appelée qualité correspondant idéalement à la rétribution attendue (sur une période donnée) par le choix de l'état (ou de l'action-état). L'algorithme apprend donc des enchaénements d'actions qui ont permis d'obtenir une récompense.
- lorsqu'il ne peréoit pas de rétribution, il répercute une part de la différence entre la qualité de l'état atteind et la qualité de l'état précédent sur l'état précédent (cette différence correspond à une erreur de prédiction). Ainsi, si le nouvel état posséde une qualité supérieure à l'état précédent, la qualité de l'état précédent va étre augmentée (et réciproquement).
- il existe un équilibre à trouver entre l'exploration (choisir l'état suivant (ou l'action suivante) au hasard) et l'exploitation (préférer l'état (ou état-action) qui posséde la meilleure qualité. Si l'environnement est dynamique, il faut toujours garder une part d'exploration. L'idéal étant que l'exploration soit plus importante lorsque les résultats sont mauvais (et réciproquement).

Les éléments sur lesquels s'appuie un algorithme d'apprentissage par renforcement tel que le SARSA sont les suivants :

- une liste d'éléments "état(s)-action(a)" possédant une qualité $Q(s, a)$ qui est initialement un nombre aléatoire.
- une variable α comprise entre 0 et 1, caractérisant le taux d'apprentissage.
- une politique de choix entre l'exploration et l'exploitation. Il existe une multitude de politiques permettant d'adapter le taux d'exploration au degrés de confiance que l'on fait à l'apprentissage effectué.
- une variable λ comprise entre 0 et 1, caractérisant le taux de remise (elle fixe la

répercution de l'erreur de prédiction sur l'"état-action" précédent).

— la taille des épisodes d'apprentissage. Cette taille correspond à un nombre maximum d'actions à effectuer avant de remettre le systéme dans un état aléatoire. En fait la notion d'épisode est surtout utilisée lorsque la remise concerne un long historique d'actions et lorsque le systéme peut se retrouver "bloqué" loin de pouvoir atteindre une récompense.

L'algorithme Sarsa (pour $S_{t-1}, A_{t-1}, R_{t-1}, S_t, A_t$) est celui-ci :

```
Initialiser Q(s,a) aleatoirement
Repeter
{    Mettre le systeme dans un etat s
     Choisir une action a en fonction de la politique
     Faire pour chaque pas de l'episode
     {
          Executer l'action a
ecompense ( Prendre connaissance de la r
          Prendre connaissance de l'
                 Choisir l'action suivante a'   soft
          se basant sur les valeurs de Q(s',*) (* correspond a toutes les actions possibles)
          Q(s,a) = Q(s,a) + α * [r + λ * Q(s',a') - Q(s,a)]
          s = s'
          a = a'
     }
}
```

1.2 L'application ReinforcementLearning

L'application que vous allez utiliser pour expérimenter l'apprentissage par renforcement a été développée en JAVA. Récupérez les fichiers vous étant destinés sur Moodle et désarchivez-les dans un répertoire dédié. Le script `compiler.sh` (sous linux ou `compiler.bat` sous windows) vous permet de compiler et exécuter le code. Il s'agit d'une première version qui doit étre améliorée mais elle peut actuellement représenter l'évolution d'un algorithme de type SARSA, appliqué à la recherche de briques oranges dans un petit labyrinthe. Une petite session de démonstration peut vous étre utile. Pour cela, lancez l'application (fichier run.sh dans le répertoire de base du projet. Au lancement, vous voyez apparaitre en haut à gauche un petit labyrinthe, les briques oranges étant des récompenses pour des agents qui doivent apprendre à les retrouver. Dans la fenétre de droite il n'y a pour l'instant rien, il s'agit d'une vue montrant l'évolution de l'apprentissage sous forme de couleur : une case rouge rapporte de nombreux points ($Q^*(s)$). Dans la fenétre au milieu à gauche, vous pouvez cliquer sur le bouton *RewardChange* et voir une modification de la position des briques oranges. Cette fonctionnalité permet de tester la vitesse de ré-apprentissage en environnement dynamique. C'est de cette fenétre que vous pouvez créer des agents, fixer les paramétres de leur algorithme SARSA (taux d'exploration, de remise et d'apprentissage) afin de les mettre en conccurence. Attention, la puce *ViewMoving* de la vue du

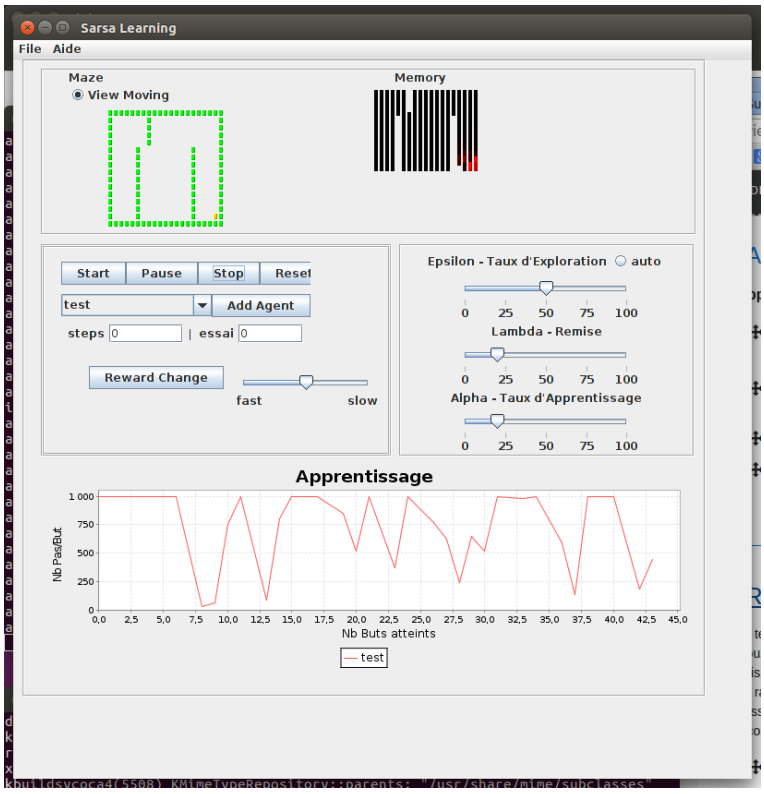

FIGURE 1 – Copie d'écran de l'application ReinforcementLearning

labyrinthe permet de montrer le déplacement de l'agent indiqué dans la fenétre centrale, mais ceci ralentit trés fortement l'application. Vous pouvez obtenir une visualisation de ce qu'a appris l'agent en cliquant sur `Refresh` de la vue de la mémoire. Enfin, la fenétre du bas indique les temps mis par l'agent pour rejoindre une brique orange, au fur et à mesure des expériences. Pour aider l'agent, un nombre de pas maximum est fixé par défaut à 1000 entre différents épisodes d'apprentissage.

1.3 Objectifs

Ce TP a pour objectif de vous familliariser avec l'apprentissage par renforcement et les problèmes associés à sa mise au point. Vous effectuerez d'abord des expériences pour évaluer l'impact des différents paramétres d'un algorithme SARSA sur les performances d'apprentissage et de ré-apprentissage en environnement dynamique. Vous coderez ensuite des adaptations optimisant la gestion des aspects dynamiques. Enfin, vous abordez la problématique de perceptions locales en modifiant les méthodes de perception des agents.

1.4 Implémentation en JAVA

Nous vous fournissons un ensemble de classes implémentées en JAVA. Les différents "packages" de l'application que vous allez utiliser sont représentés figure 2.

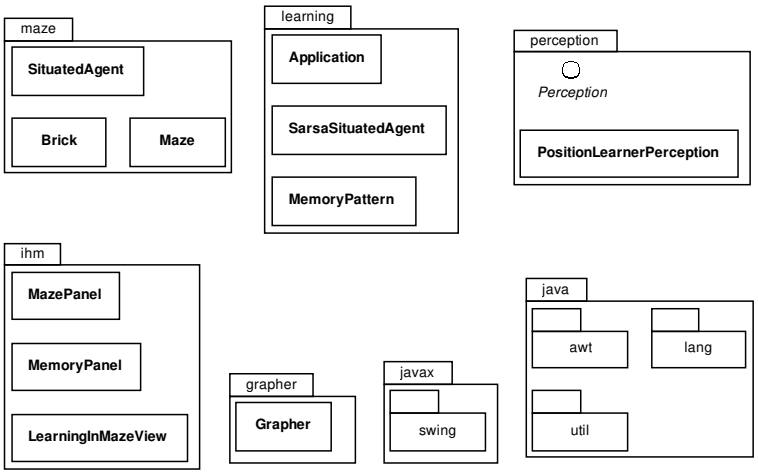

FIGURE 2 – *les packages de l'application ReinforcementLearning.*

1.4.1 Description rapide des packages

Voici les packages utilitaires que vous ne devriez pas avoir à modifier :

- `java` les librairies de base de java.
- `javax` librairie swing (graphique) standart.
- `ihm` (interface homme-machine) sert à l'affichage de l'ensemble des éléments de cette application. Vous n'avez pas besoin de le modifier.
- `grapher` utilise une librairie permettant de tracer des graphes (jFreeChart). Vous n'avez pas besoin de le modifier, sauf si vous voulez afficher des graphiques différents de ceux prévus initialement.

Voici les packages qui vous concernent plus directement :

- package `maze` (figure 3)
  Ce package permet de simuler le déplacement d'un agent `situatedAgent` dans un labyrinthe `Maze` formé de briques `Brick`.
  Les éléments susceptibles de vous intéresser ici sont les attributs suivants du `SituatedAgent` :
  - `_myPerception` qui sera précisé par la suite est la perception à laquelle sera associée un apprentissage.
  - `_possibleActions` est un tableau de chaénes de caractéres énumérant toutes les actions possibles de l'agent. Initialement, il s'agit des 4 actions de bases 'haut', 'bas', 'gauche' et 'droite'.
  - `_A` est une chaine de caractére définissant l'action que va effectuer l'agent lorsqu'il exécutera la méthode `runAction()`.
  - `_R` correspond aux points associés à la brique rencontrée suite à l'exécution de `runAction()`

  Notons également que les instances de la classe `Brick`, possédent outre une position `_position`, une valeur `_value` qui permettra de définir des briques 'récompense'. Actuellement, une brique 'récompense' vaut 10 points. Enfin, la classe `maze` n'est

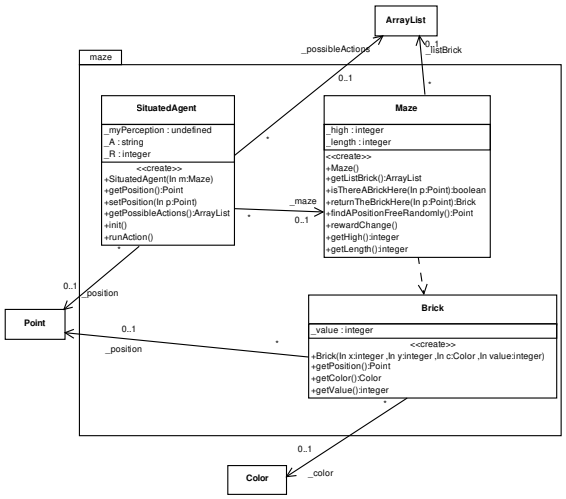

FIGURE 3 – les classes du package `maze`.

pour l'instant pas bien terminée et c'est dans le constructeur que le labyrinte est créé (dans le futur, les labyrinthes pourront étre sauvegardés dans des fichiers externes).

— package `learning` (figure 4)
Ce package est le package principal pour ce qui concerne l'apprentissage. Il contient la classe `Application` qui gére l'apprentissage de plusieurs `SarsaSituatedAgent`. Les méthodes de cette classe sont activées par l'IHM de l'application. La classe `MemoryPattern` représente la brique de base de l'apprentissage : une action `_action`, une qualité `_qualitie` et une perception `_perception`. Les perceptions seront détaillées dans le package dédié `perception`. Le `SarsaSituatedAgent`, qui hérite de `SituatedAgent`, est détaillé dans le paragraphe suivant.

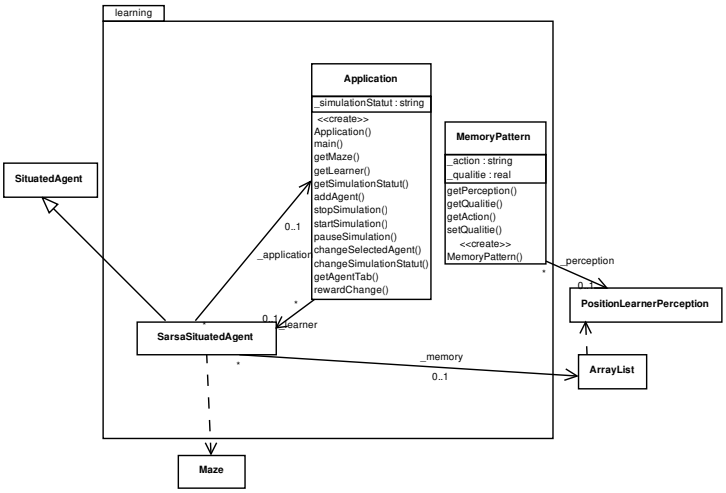

FIGURE 4 – les classes du package `learning`.

— package `perception` (figure 5)
Ce package doit à terme contenir de nombreuses classes définissant des perceptions différentes d'un agent situé `SarsaSituatedAgent`. Actuellement il ne contient que la classe `PositionLearnerPerception` qui est une perception caractérisée par la position absolue (en X et Y) de l'agent (attribut `_position` de la classe `Point`).

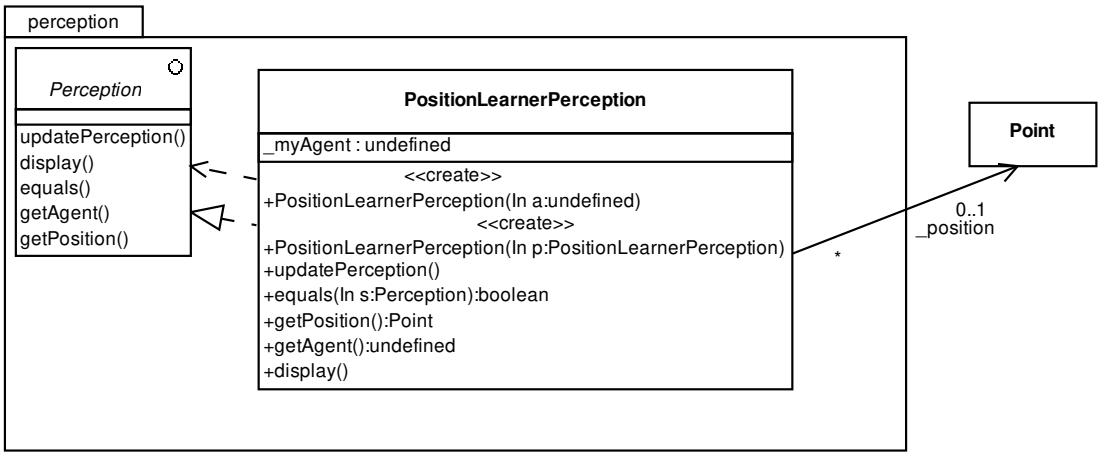

FIGURE 5 – les classes du package `perception`.

1.4.2 La classe `SarsaSituatedAgent`

Cette classe implémente un agent situé `SituatedAgent` qui apprend à retrouver des briques `Brick` rapportant 10 Points, au milieu d'un labyrinthe `Maze`. La boucle d'un apprentissage actif est implémentée dans la méthode `sarsaAlgorithmeStep()` :

```
//***************Boucle d'apprentissage actif ********************
public void sarsaAlgorithmeStep(){
     runAction();
     chooseAPAction();
        learn();
    }
```

`runAction()` execute l'action courante décidée par le pas précédent, `chooseAPAction()` sélectionne l'action suivante par exploitation ou exploration. Enfin, `learn()` est le pas d'apprentissage SARSA.
Pour comprendre l'implémentation de l'algorithme SARSA, les attributs importants de cette classe sont :

— `_S : PositionLearnerPerception` est la perception courante (de l'agent)
— `_A : string` (hérité de `SituatedAgent`), représente l'action courante
— `_R : integer` (hérité de `SituatedAgent`), représente le renforcement lié à l'éxecution de l'action courante

- _SP : PositionLearnerPerception est la perception succedant à l'exécution de _A (s' de SARSA)
- _AP : string représente l'action suivante envisagée (a' de SARSA)
- _alpha est le taux d'apprentissage de l'agent
- _lambda est le taux de remise
- _epsilon est le taux d'exploration
- _memory est un tableau qui contient des MemoryPattern associant un triplet perception/action/qualité (voir la présentation du package section précédente). C'est ce tableau qui est utilisé pour retrouver les Q(s,a) gréce aux méthode getQSA(In perception:PositionLearnerPerception, In action:string):real et getBestValueForS state:PositionLearnerPerception):real qui renvoie respectivement Q(s,a) et $max(Q(s,*))$ (avec * = ensemble des actions possibles).

L'exécution de l'action courante modifie la position de l'agent (runAction de la classe SituatedAgent), la perception (updatePerception()) et la variable _SP. Si une mémoire n'est pas associée à cette perception, un tableau lui est affecté (il existe autant de qualités que de couple perception-action possibles) grace à la méthode createNewMemoryWith

```
//********************Execution d'une action ************************
public void runAction(){
     super.runAction();
     _SP=_myPerception;
     if(!(existeAMemorieWith(_myPerception)))
     {
     createNewMemoryWith(_myPerception);
     }
    }

//***************Execution d'une action dans la classe  SituatedAgent****
    public void runAction(){
Point newPosition=null;

     if(_A.equals("haut")){
     newPosition = new Point((int)(_position.getX()),(int)(_position.getY()-1));
     }
     if(_A.equals("bas")){
     newPosition = new Point((int)(_position.getX()),(int)(_position.getY()+1));
     }
     if(_A.equals("droite")){
     newPosition = new Point((int)(_position.getX()+1),(int)(_position.getY()));
     }
     if(_A.equals("gauche")){
     newPosition = new Point((int)(_position.getX()-1),(int)(_position.getY()));
     }

     Brick b=_maze.returnTheBrickHere(newPosition);
     _R=0;
```

```
if(b!=(Brick)null)
_R = b.getValue();

// les murs rewards -1
if(_R!=-1){
_position = newPosition;
}
_myPerception.updatePerception();
}
```

La méthode `chooseAPAction()` va fixer `_AP` soit par exploration, soit par exploitation, en fonction du taux d'exploration.

```
//**************Choix de l'action suivante ***********************
public void chooseAPAction(){
     //Exploration ou exploitation

     float choose = _randomGenerator.nextFloat();
     if(choose<_epsilon){
     chooseAPActionRandomly();
        } //exploration

     else {
     chooseAPGreedyAction();
     } //exploitation ;
    }
```

Enfin, la méthode `learn()` met à jour `Q(_S,_A)`, selon l'algorithme SARSA.

```
//***********algorithme SARSA*********************
 public void learn(){
     float QSA = getQSA(_S,_A);
     float QSAPrime = getQSA(_SP, _AP);
     float newQSA=QSA+_alpha*(_R+_lambda*(QSAPrime-QSA));
     if (newQSA >_bestQuality){_bestQuality=newQSA;}
     setQSA(_S,_A,newQSA);
     _S= new PositionLearnerPerception(_SP);
     _A= new String(_AP);
    }
```

1.5 Travail à réaliser

Nous vous demandons de rendre un rapport papier de votre travail dans des délais imposés. La qualité de ce rapport sera prise en compte pour l'évaluation du module. Sont évalués :

- Votre démarche d'analyse.
- Les performances de vos propositions.

— La qualité du code proposé.
— La quantité de travail effectué (nombre de questions traitées).

Attention, pour chaque modification effectuée, sauvegardez votre code et créez des versions différentes de l'application. Mieux, creez de nouvelles classes. Cependant, ceci est réservé aux programmeurs chevronnés car le code n'est pas encore assez souple pour que ceci se fasse facilement.

1.5.1 Prise en main des paramétres de l'algorithme

— Etudiez l'influence du taux d'apprentissage, du taux de remise et du taux d'exploration sur la vitesse d'apprentissage des agents. Faites-en un rapport explicatif. Attention, pour comparer des paramétrages différrents, il faut mettre les agents dans des situations similaires. Prendre garde des situations trops simples ou trop complexes à atteindre.
— Faites également des expériences lors d'une modification dynamique de l'environnement. Etudiez également l'impacte des paramétres sur la variabilité des temps aprés apprentissage ainsi que leur valeur moyenne finale.
— Vous pouvez modifier la taille et la configuration du labyrinthe dans le constructeur de la classe `Maze` (méthode `Maze`).

1.5.2 Gestion dynamique du taux d'exploration `_epsilon`

Afin d'améliorer l'algorithme et surtout son adaptation à un changement d'environnement, nous proposons de faire varier le taux d'exploration de faéon automatique en fonction de l'avancée de l'apprentissage. Ainsi, le taux d'exploration doit étre fort si il n'y a pas d'apprentissage et doit étre faible dans le cas contraire.

— Proposez une adaptation du code pour faire cette modification. Si vous voulez, vous pouvez exploiter la trace `_trace` d'apprentissage du `SarsaSituatedAgent`. Cette trace est un tableau lissé (limité à une certain nombre de `Points`) des 'performances' de l'agent, depuis le début de son apprentissage. Chaque point est un couple (nombreDeBriquesAtteneintes, TempsMisPourAtteindreUneBriqueOrange).
— Montrer sur des exemples, les performances de ce nouvel agent.

1.5.3 Perception partielle

Actuellement la perception d'un agent (qui caractérise un état de l'algorithme d'apprentissage) est de la classe `PositionLearnerPerception`. Elle est mise à jour par sa méthode `_updatePerception` qui recopie la position absolue de l'agent. Une perception plus réaliste correspond à un ensemble d'éléments peréus aux alentours de l'agent. Elle devient souvent non-markovienne et il faut parfois utiliser des mémoires intermédiaires pour la caractériser. Creez une nouvelle classe de perception permettant d'éviter une énumération de toutes les positions possibles de l'agent. Montrez les performance de l'agent ainsi obtenu.

1.5.4 Questions subsidiaires

— Implémentez un autre algorithme de gestion de l'équilibre exploration/exploitation comme le *softmax* ou la *méthode des poursuites*. Comparez-le à votre solution obtenue au 1.5.2

— Adaptez le code actuel pour que l'algorithme d'apprentissage soit une classe et que différents algorithmes (Q-learning ou autre) puissent étre mis en concurrence sur l'outil. Proposez une modélisation puis une implémentation de votre solution.

— Testez des labyrinthes plus complexes, plus grands, avec des obstacles repoussants ... et mettez à l'épreuve vos solutions.