

UNIVERSITÉ DU LIMOUSIN

UNIVERSITÉ DE CRAIOVA

---

# Développement d'applications pour l'apprentissage d'agents informatiques

---

**Étudiant :**

Samuel FELTON

**Directeur de stage :**

Jacques DUMONT

**Maître de stage :**

Costin BĂDICĂ





---

## Remerciements

Tout au long de mon stage, j'ai bénéficié du soutien de plusieurs personnes à qui je tiens à témoigner ma reconnaissance pour m'avoir permis de vivre cette expérience très enrichissante.

Tout d'abord, je souhaiterais remercier les départements des relations internationales de l'IUT du Limousin et de l'Université de Craiova. Leur aide m'a été très précieuse, et a rendu mon séjour plus facile.

Je tiens aussi à remercier M. Bădică pour m'avoir accepté et proposé un sujet de stage aussi passionnant. Je remercie également M. Becheru, qui m'a guidé et aidé dans mes démarches administratives.

Enfin, je souhaite remercier M. Dumont, qui m'a apporté ses conseils et son suivi.

---

# Sommaire

<b>Introduction</b>	<b>1</b>
<b>1 Contexte du projet</b>	<b>2</b>
1.1 L'université de Craiova . . . . .	2
1.2 Le groupe IDS . . . . .	5
1.3 L'application existante . . . . .	6
<b>2 Mise en place du projet</b>	<b>10</b>
2.1 Organisation et déroulement du stage . . . . .	10
2.2 Compréhension des technologies utilisées . . . . .	14
2.3 Analyse de l'application . . . . .	20
<b>3 Réalisation et bilan</b>	<b>23</b>
3.1 Première itération . . . . .	23
3.2 Améliorations et modifications . . . . .	29
3.3 Poursuite et bilan . . . . .	34
<b>Conclusion</b>	<b>36</b>
<b>Sources documentaires</b>	<b>I</b>
<b>Liste des figures, tableaux et procédures</b>	<b>II</b>
<b>Annexes</b>	<b>VII</b>

---

## Introduction

Afin de finaliser ma formation auprès du département Informatique de l'IUT de Limoges, j'ai été amené à effectuer un stage de dix semaines dans un milieu professionnel. Cette immersion a pour but de mettre en pratique les notions acquises pendant les deux années de formation, tout en les enrichissant de nouvelles connaissances.

C'est l'université de Craiova, et plus particulièrement la faculté d'informatique, qui m'a accueilli du 11 avril au 17 juin 2016.

Le stage vise à étendre et consolider les recherches de M. Bădică, un enseignant et chercheur, qui touchent aux domaines de la Programmation Orientée Agent et de l'Intelligence Artificielle. Le but est principalement de proposer une application qui facilite les tests et le paramétrage des différentes options d'une expérience, décrite dans un article de recherche. Cela permet aussi d'avoir une base sur laquelle implémenter d'autres essais et configurations. L'expérience place des agents dans une grille à deux dimensions. Leur but est de «s'échapper» en atteignant la sortie. Sur le long terme, les agents apprennent les meilleures actions, grâce aux techniques de l'apprentissage par renforcement.

Pour ce faire, j'ai, après une phase d'observation et d'apprentissage, repris le programme initial. J'ai développé une interface graphique, rendant l'expérience utilisateur plus agréable, et surtout le temps de mise en oeuvre moins long. Je l'ai ensuite améliorée lors d'une deuxième itération. L'ajout d'autres types d'agents a aussi été fait, permettant de tester leur validité et d'avoir une vue claire de leurs actions.

Un article de M. Bădică reprenant les avancées réalisées depuis la parution de l'article initial doit être publié. Il reprendra les travaux réalisés au sein de l'interface graphique. Cela a mis en évidence les besoins de sauvegarde et d'interprétation automatique des données, à travers la génération de graphiques.

Le travail en autonomie, dans un domaine théorique, m'a appris à avoir un retour critique sur moi-même et à aller toujours plus loin. J'ai pu, tout en suivant les demandes de mon tuteur, développer ma prise d'initiatives.

# 1 Contexte du projet

## 1.1 L'université de Craiova

### 1.1.1 L'université

C'est à l'université de Craiova que j'ai effectué mon stage, du 11 avril au 17 juin 2016.

Elle fut fondée en 1947 et comptait au départ 4 instituts. C'est la plus grande université de la province d'Olténie (le Sud-Ouest de la Roumanie) et du Comté de Dolj. Elle fait partie de l'Association des Universités Européennes et intègre le système de crédits ECTS. En 2014, elle était classée 11<sup>ème</sup> université de Roumanie.

Ses missions sont :

- L'avancement et le transfert de savoir vers la société à travers un apprentissage avancé et la recherche scientifique ;
- L'éducation continue et de qualité, afin que les spécialistes puissent répondre aux demandes de l'environnement socio-économique grâce à l'insertion professionnelle ;
- dispenser une formation initiale et continue pour les enseignants ;
- La contribution à l'avancement des sciences fondamentales et appliquées à travers la recherche, l'innovation et le transfert de technologies ;
- Le développement personnel des étudiants en matière de créativité, aussi bien individuelle que collective ;
- La promotion du libre échange d'idées et de l'esprit critique ;
- La promotion des valeurs Européennes dans les domaines scientifiques, culturels et éducationnels, grâce aux programmes de coopération internationale académique.

L'université en chiffres, c'est :

- 16 facultés ;
- 2 collèges ;
- 17 098 étudiants *undergraduates* (niveau inférieur ou égal à Bac +3) ;
- 4 997 étudiants en Master ;
- 488 doctorants ;
- 948 enseignants et chercheurs.

### Symbolique

La devise de l'université est : « *Vita sine litteris mors est* », La vie sans apprentissage est la mort.

Son logo (voir Figure 1), affiché sur son site Web ainsi que sur les documents officiels, présente un blason avec 3 parties distinctes :

- En bas à gauche, un lion avec une étoile, symbole de la province historique d'Olténie ;
- En bas à droite, un livre ouvert, symbole du savoir et de l'enseignement ;
- En haut, les armoiries de Roumanie, l'aigle d'or avec un sceptre et une épée, symboles de souveraineté.



FIGURE 1 – Logo de l'université de Craiova

### Relations internationales

L'université propose dans la majorité de ses cursus des cours dispensés dans des langues étrangères. Il y a par exemple, une spécialisation «Informatique en anglais ».

Axée sur l'international, elle a des partenariats avec une grande variété d'établissements de l'enseignement supérieur à travers l'Europe et le monde.



FIGURE 2 – Carte des partenariats Erasmus+ de l'université en 2013

### 1.1.2 la faculté

La faculté d'automatisation, d'informatique et d'électronique (*Facultatea de Automatica, Calculatoare și Electronica*) est située à l'est de la ville, près des usines Ford et d'entreprises tertiaires. Elle est bâtie à côté du parc Electroputere, ancienne «super-usine», dédiée à la création de composants électroniques et super-calculateurs. Le parc est maintenant un centre commercial d'une superficie de 71 000 m<sup>2</sup>.



FIGURE 3 – Façade de la faculté

La faculté propose différents diplômes, allant de la licence (son équivalent) au doctorat. ils couvrent plusieurs thèmes de l'informatique, comme l'ingénierie logicielle, l'intelligence artificielle ou la recherche de données par moyens visuels.

Elle a aussi des relations privilégiées avec les entreprises implantées localement, telles que Ubisoft ou IBM Romania.



## 1.2 Le groupe IDS

C'est avec les membres du groupe Intelligent Distributed Systems (IDS) que j'ai collaboré et échangé des idées lors de mon stage. Ce groupe a été fondé par M. Bădică et est principalement composé de doctorants et étudiants de la faculté.

Son but est de répondre aux challenges posés par les avancées en matière d'Intelligence Artificielle et de systèmes répartis. Le groupe souhaite répondre aux besoins de la fusion de ces deux champs de l'informatique, et explorer les solutions théoriques et pratiques possibles.



FIGURE 4 – Le logo IDS

Le groupe a participé à des projets tels que :

- le projet DIADEM (Distributed information acquisition and decision-making for environmental management) ;
- L'implémentation d'un système intelligent basé sur les technologies du Web pour le E-Learning ;
- Une plateforme pour des agents de E-commerce ;
- Le projet K-Swan (Interoperable Knowledge-based Framework for Negotiating Semantic Web Agents)

## 1.3 L'application existante

Le projet part d'une application développée en partie par M. Bădică dans un article [6] publié à l'occasion de l'AgTAmI Workshop, une conférence sur l'utilisation des technologies basées sur le paradigme agent dans l'industrie.

### 1.3.1 But initial

Le but de cet article est de stimuler la recherche en apprentissage machine au sein de la communauté des développeurs utilisant la Programmation Orientée Agent\* (POA) basée sur le modèle Belief-Desire-Intention\* (BDI) et les Multi-Agent System\* (MAS), quel que soit le langage. Le but de ces recherches est aussi de combler le retard qu'a la programmation agent sur les autres méthodes de programmation en matière d'apprentissage machine. Ce retard provient du fait que la communauté de développeurs, même si active, est assez restreinte, la POA ne bénéficiant pas de la même popularité que la Programmation Orientée Objet\* (POO). La Programmation Orientée Agent n'est que très peu enseignée et les ressources sur Internet assez rares.

Les auteurs de l'article ont aussi souhaité mettre en oeuvre et exploiter au maximum les avancées les plus récentes de la POA pour montrer cette progression aux créateurs d'agents intelligents (capables de raisonnement et d'apprentissage), qui selon eux ignorent souvent les bénéfices de ce paradigme et préfèrent des solutions traditionnelles.

### 1.3.2 L'expérience

L'expérience est inspirée du livre *Artificial Intelligence : A Modern Approach*. Elle présente un agent situé dans un environnement bi-dimensionnel (un quadrillage). Cet environnement est décomposé en coordonnées de type ligne-colonne. chacun de ces emplacements appartient à l'un des types suivants :

- Un espace libre sur lequel l'agent peut être placé ;
- Un obstacle ou un mur : l'agent ne peut être sur un emplacement de ce type ;
- Un état terminal : un emplacement libre qui signifie la fin de l'essai. Il peut être négatif (état d'échec, l'agent est par exemple tombé dans un trou) ou positif (l'agent a atteint une destination demandée).

chaque état se voit attribué une récompense, transmise à l'agent lorsqu'il s'y trouve. Ces récompenses l'aideront à estimer l'utilité d'un état en association à une action.

L'agent a à sa disposition quatre actions : **down**, **up**, **left**, **right**. Dans cet article, l'agent applique une méthode d'Apprentissage par renforcement\* (*Reinforcement Learning*) connue sous le nom de *Temporal Difference Learning*.

Son but principal est d'estimer la fonction d'utilité associée à une Politique\* (*policy*) (c'est un agent passif, il ne cherche pas à optimiser ou explorer ses options). Pour cela, l'agent va répéter un grand nombre de fois l'expérience. Pour chaque essai, son but est d'atteindre un stade terminal, de préférence positif.

L'environnement utilisé, ainsi que les politiques associées, est présenté en Figure 5. Une flèche représente la direction que prendra l'agent à cet emplacement. Par exemple, à

l'emplacement (3,1) (ligne,colonne), l'agent devra aller à droite. Les emplacements (2,4) et (3,4) représentent les états terminaux.

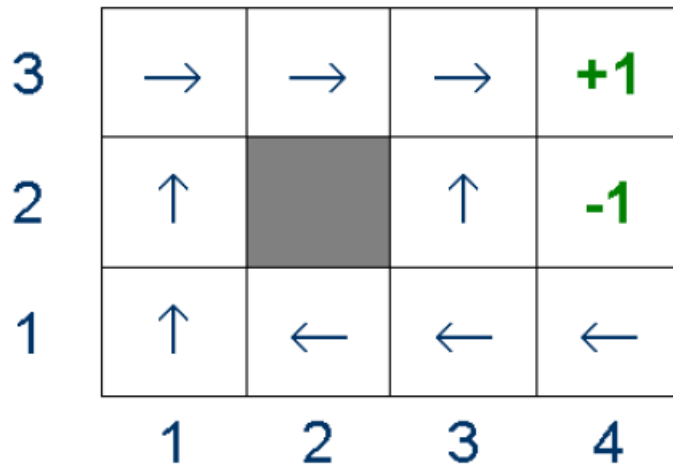


FIGURE 5 – L'environnement utilisé dans l'article

À chaque déplacement, l'agent perçoit la récompense (gain ou perte), transmise par l'environnement (voir Figure 6). L'emplacement (2,4) symbolise un état d'échec et se voit donc attribué une récompense égale à -1. La position (3,4) représente la destination de l'agent et a donc une récompense de 1. Pour que tous les emplacements aient un impact, les emplacements libres ont une récompense égale à -0,04. Cela pourrait par exemple correspondre à une perte d'énergie liée au déplacement de l'agent pour arriver à cet état.

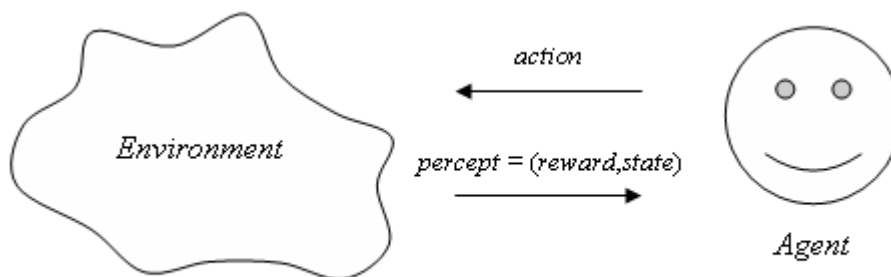


FIGURE 6 – Échanges entre un agent et son environnement dans le cadre de l'apprentissage

Les perceptions et actions de l'agent sont aussi incertaines dans cette expérience. Ainsi, l'agent a 80 % de chance d'entreprendre l'action prévue par la politique liée à ces coordonnées. Le pourcentage restant représente pour lui un risque de tourner à gauche ou à droite relativement à la politique définie. Par exemple, un échec lié à la politique de l'agent pour l'emplacement (2,3) pourrait être d'aller vers la gauche (où il y a un obstacle) ou le haut (échec de l'essai). Ce risque est présenté par la Figure 7.

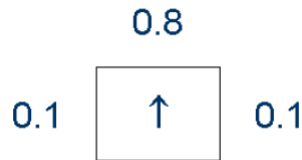


FIGURE 7 – Probabilités de succès ou d'échec liées à une politique

Pour créer cette expérience, les auteurs ont utilisé Jason pour représenter les agents et Java pour simuler l'environnement.

Au niveau de l'environnement, la solution la plus simple a été choisie, les données étant représentées par des tableaux statiques et les méthodes de l'environnement y accédant grâce à un système d'index. Les obstacles (les bords de l'environnement étant considérés comme tels) sont traduits par :

```
final boolean walls[][] = {
    {true,true,true,true,true,true},
    {true,false,false,false,false,true},
    {true,false,true,false,false,true},
    {true,false,false,false,false,true},
    {true,true,true,true,true,true},
};
```

Code 1 – Obstacles dans l'application initiale

En ce qui concerne la conception de l'agent, les capacités de l'architecture BDI ont été exploitées. La base de croyances (*Belief Base*) sert à stocker les Politiques, les règles et autres informations nécessaires à l'apprentissage. Le code Jason ci-dessous est un extrait des connaissances initiales de l'agent :

```
limit(10000). //Number of trial limit
terminal_state(s(_,_,t)).
non_terminal_state(s(_,_,n)).
//Rule returning true if the number of trial N is below the fixed limit
below_limit(N) :-
    limit(L) &
    N < L.
policy(s(1,1),up). //For coordinates (1,1) go up
```

Code 2 – Belief Base de l'agent TDL

Cet exemple présente bien l'utilisation de l'architecture BDI, qui sera expliquée plus en détail dans la deuxième partie.

### 1.3.3 Résultats

L'expérience a porté ses fruits, et le comportement de l'agent correspond à ce qui était attendu.

Pour produire les résultats, les auteurs ont lancé huit fois le programme, tout en enregistrant les valeurs des fonctions d'utilité de l'agent. Pour chaque exécution, une limite de 50 000 passages (un passage représente un mouvement) a été fixée. En moyenne, l'agent a fait 9 470 essais (jusqu'à ce qu'il atteigne un stade terminal).

Pour chaque politique, une moyenne des valeurs de la fonction d'utilité a été faite et un graphique tracé (voir Figure 8)

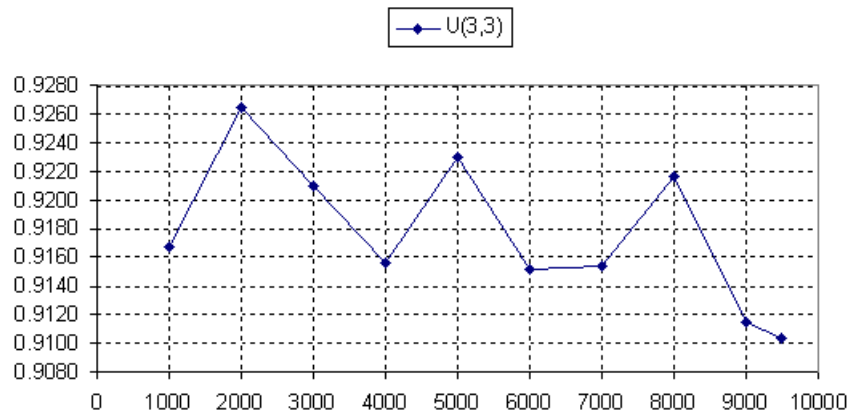


FIGURE 8 – Fonction d'utilité de l'état (3,3) pour un agent TDL, avec l'action *left* assignée

Des problèmes ont cependant été notés, en partie liés à la technologie utilisée.

Par exemple, le manque de maturité de Jason a posé des problèmes de performance, surtout dûs au fait que le langage n'implémentait pas ce qui est fait depuis la version 1.4, sortie récemment) l'Optimisation de la récursion terminale (*Tail Call Optimization*)\* pour les plans, qui permet d'éviter la consommation de mémoire liée à la récursion.

Cette expérience est la base de ce sur quoi repose mon sujet de stage, qui vise à approfondir et élargir ce que permet l'application.

## 2 Mise en place du projet

### 2.1 Organisation et déroulement du stage

Mon stage a débuté le 11 avril 2016 à midi, l'heure convenue avec mon tuteur pour notre rencontre. Lors de cette première journée, j'ai pu prendre mes repères et rencontrer les personnes qui m'ont aidé lors de ce stage (principalement des membres du groupe IDS).

Nous avons ensemble discuté de l'organisation du stage et des besoins du projet. Il a été décidé que mes premières semaines de travail seraient dédiées à mon apprentissage des technologies et théories liées au projet. Nous avons décidé de travailler par cycle, afin d'avoir un meilleur retour sur l'application et éviter une perte de temps.

#### 2.1.1 Matériel et documents

Le projet ne nécessitant pas de matériel particulier, j'ai pu utiliser mon ordinateur pour développer. M. Bădică m'a donné les clefs du laboratoire IDS, pour que je puisse venir librement. Il ne m'a pas imposé d'horaires fixes, lui même n'étant pas toujours à la faculté.

Mon tuteur m'a aussi mis à disposition des documents auxquels je n'aurais pas eu accès autrement (articles dans des revues payantes...). Ceux-ci ont été d'une grande aide pour le développement et la recherche d'idées pour l'application.

#### 2.1.2 Formulation des besoins

Avant de réaliser le projet, il a fallu identifier les besoins auxquels devait répondre l'application.

#### Cible de l'application

L'application est à vocation expérimentale. Elle répond à un besoin de recherche dans le domaine de l'intelligence artificielle appliquée à la POA.

Les utilisateurs seront donc surtout des chercheurs. M. Bădică sera l'utilisateur principal, car il souhaite pouvoir développer ses idées suite à l'expérience présentée auparavant. En effet, le code n'étant pour l'instant que peu modulaire (et donc difficilement modifiable), il est peu aisé de produire divers environnements où tester le comportement des agents cognitifs.

#### Première itération

Le besoin prioritaire est d'avoir un environnement interactif, qui puisse être modifiable à la volée. Cela signifie donc qu'une Graphical User Interface\* (GUI) est de rigueur. Cette interface doit être pratique et accessible afin de rendre l'expérience (User eXperience\* (UX)) agréable.

L'environnement doit pouvoir être modifié en profondeur. Pour cela, l'utilisateur dispose de plusieurs options :

- Modifier la taille de l’environnement. Le nombre de lignes et de colonnes doit être paramétrable de manière indépendante (c’est-à-dire que l’environnement est rectangulaire et non carré) ;
- Changer les différents paramètres d’un état : le type d’état, la récompense associée ;
- Changer les politiques d’un agent Temporal Difference Learning (TDL) pour chaque état.

La GUI doit afficher les données d’apprentissage de l’essai en cours. Ainsi, pour chaque état, les données suivantes doivent être affichées :

- Sa récompense ;
- Son type (obstacle, espace libre, état terminal) ;
- la politique associée ;
- La valeur d’utilité estimée par l’agent pour la politique ;
- Le nombre de fois que l’agent est arrivé à cet état.

L’interface doit aussi être actualisée en temps réel afin de pouvoir constater l’évolution de la réflexion d’un agent.

L’utilisateur doit pouvoir sauvegarder ou charger un environnement et ses caractéristiques.

## Deuxième itération

Lors de la deuxième phase d’analyse (après la première itération et les premiers retours), de nouveaux besoins ont été mis en lumière.

Tout d’abord, il est nécessaire que le programme fonctionne avec d’autres types d’agents. Un agent utilisant l’algorithme de Q-Learning a été développé et doit être adapté pour fonctionner avec l’environnement. Les besoins de l’agent étant différents, cela signifie que l’interface doit incorporer des mécanismes tout aussi différents. L’agent n’utilisant plus de Politiques, la GUI doit afficher l’action favorisée par l’agent, mais aussi conserver les données relatives aux autres actions. Celles-ci doivent être présentées de manière claire et précise.

Un autre type d’agent doit aussi être développé. Celui-ci repose sur l’algorithme d’apprentissage State-Action-Reward-State-Action (SARSA), une variante du Q-Learning ayant toutefois un fort impact sur les résultats obtenus.

L’environnement doit aussi pouvoir accommoder plusieurs agents, qui dérouleront leurs essais de manière simultanée. Des travaux ont déjà été initiés dans ce sens par mon tuteur, desquels je m’inspirerai.

À des fins de sauvegarde et de comparaison, il est aussi nécessaire de pouvoir exporter les données de l’essai. Ces données devront rendre compte de l’évolution de l’utilité des Politiques au cours du temps pour chaque agent. Ainsi, un export Comma-separated Values\* (CSV) est à considérer. Il serait aussi appréciable de pouvoir visualiser ces données directement, en générant par exemple des graphiques.

D’autres pistes sont aussi à explorer, comme intégrer un brouillage des perceptions des agents (celles-ci n’étant dans la vraie vie qu’une estimation), ou l’utilisation d’artéfacts pour contrôler les interactions entre agents et environnement.

## Modélisation UML des besoins

Pour mieux cerner les besoins du projet, j'ai réalisé un diagramme de Use Case\* les représentants :

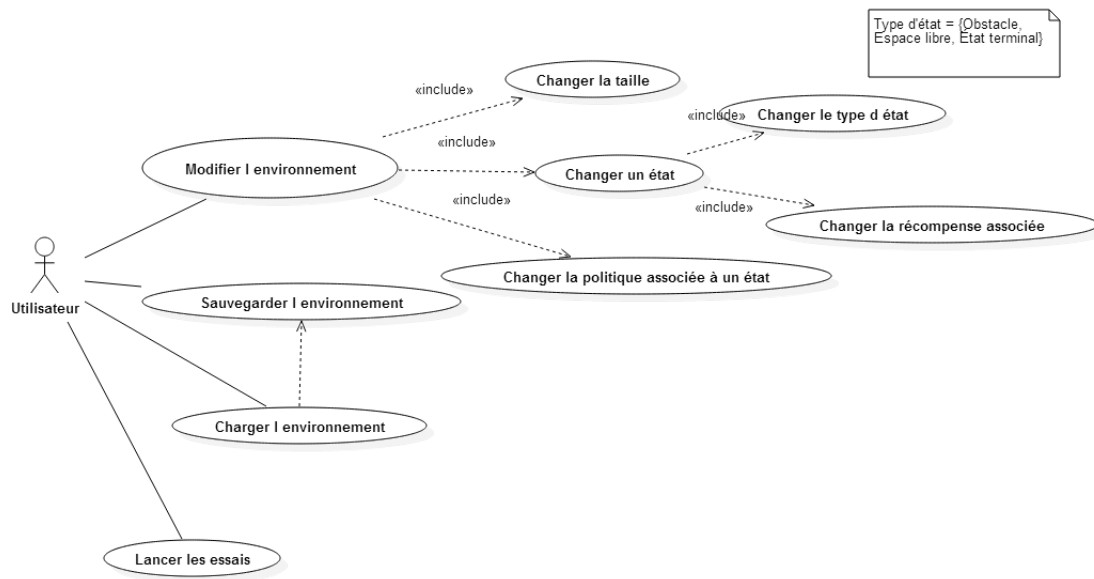


FIGURE 9 – Use case initial de l'application

### 2.1.3 Planning

Le travail n'étant pas planifié à l'avance, aucun diagramme de Gantt n'a pu être réalisé au début du stage. En effet, le travail en itération ne s'y prête que peu, les besoins et tâches à accomplir étant formulées pour être évaluées directement.

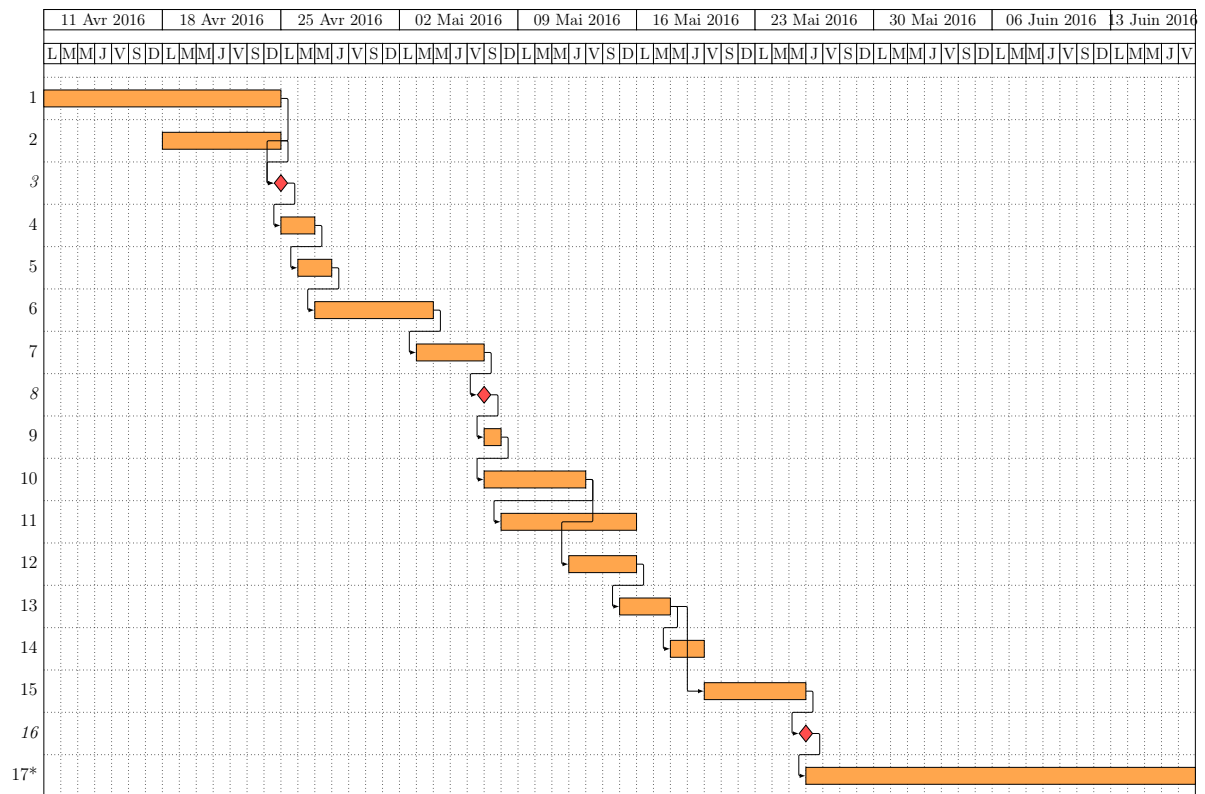
Un Gantt a été réalisé à mi-parcours afin d'évaluer le coût en temps du développement des différentes fonctionnalités.

La première phase de travail a été dédiée à l'apprentissage et à l'exploration des technologies mises en oeuvre. Cette phase a été essentielle à la compréhension de l'expérience initiale et des défauts et qualités qu'elle présente.

Puis, la première itération a commencé. La première tâche effectuée a été de Refactor\* (retravailler) le code afin de le rendre plus adaptable (répondre au besoin de Scaling\*) et réduire le risque d'erreur. Après cette période de *refactoring*, le travail sur la GUI a pu débuter.

Le temps restant après la deuxième itération (et la remise de ce mémoire) sera consacré à la génération de statistiques à partir de l'application et au développement d'une nouvelle expérience basée sur celle-ci, reprenant les règles d'un jeu de poursuite. Les statistiques seront normalement utilisées dans le prochain article de mon tuteur, qui s'appuiera sur ce projet.





Index	Nom	Début	Fin	Durée (jours)
1	Phase d'apprentissage	11/04/2016	24/04/2016	13
2	Observation de l'existant	18/04/2016	24/04/2016	6
3	Début 1 <sup>ère</sup> itération	24/04/2016	24/04/2016	0
4	Maquettage et prototypage	25/04/2016	26/04/2016	2
5	Refactoring du code existant	26/04/2016	27/04/2016	2
6	1 <sup>ère</sup> phase de développement	27/04/2016	03/05/2016	7
7	Tests et refactoring	03/05/2016	06/05/2016	4
8	Début 2 <sup>ème</sup> itération	06/05/2016	06/05/2016	0
9	Identification des besoins supplémentaires	06/05/2016	07/05/2016	2
10	Adaptation de l'application initiale	07/05/2016	12/05/2016	6
11	Développement de l'agent SARSA	08/05/2016	15/05/2016	8
12	Développement de nouvelles fonctionnalités	12/05/2016	15/05/2016	3
13	Développement de l'outil de génération de graphiques	15/05/2016	17/05/2016	3
14	Génération et interprétation de données pour l'article de M. Badica, Documentations des outils	18/05/2016	19/05/2016	2
15	Phase de test et d'amélioration du programme	20/05/2016	25/05/2016	6
16	Itération finale	25/05/2016	25/05/2016	0
17	Développement du jeu de poursuite (prévisionnel)	26/05/2016	17/06/2016	23

TABLE 1 – Tâches présentées sur le Gantt

## 2.2 Compréhension des technologies utilisées

Une grande partie du projet utilisant des technologies et théories non vues à l'IUT, une phase d'adaptation a été nécessaire.

### 2.2.1 Jason

#### AgentSpeak

AgentSpeak est un langage abstrait créé par Anand Rao en 1996. Ce langage avait pour vocation d'aider à la compréhension de l'architecture BDI et du raisonnement des agents. Le langage n'avait pas vocation à être utilisé pour le développement, mais l'intérêt qu'il a suscité lors de sa publication en a fait un langage de choix. AgentSpeak est un langage de programmation basé sur la logique, c'est-à-dire qu'il repose sur des règles et des plans pour décrire comment résoudre un problème. Cette logique est très proche des mathématiques, et peut être facilement comprise à la lecture.

#### Jason

Jason est un projet développé principalement par Jomi F. Hübner et Rafael H. Bordini. Il est Open Source\* et sous licence GNU Lesser General Public License\* (GNU LGPL). Il est développé en Java.

Le langage s'appellait initialement JASON, pour «**J**ava-based **A**gentSpeak interpreter used with **S**ACI for multi-agent distribution **o**ver the **n**et». Cependant, la diversification des infrastructures de distributions proposées a fait que l'acronyme n'a plus de sens. Il a donc été transformé pour devenir Jason.

Le logo de Jason est une peinture de Gustave Moreau, Jason et Médée, datant de 1865.



FIGURE 10 – Logo simplifié de Jason

Jason propose un ensemble d'outils pour développer des systèmes multi-agents (MAS) :

- Un interpréteur AgentSpeak. AgentSpeak étant un langage abstrait, celui-ci ne propose pas directement de compilateur. Jason doit donc transformer celui-ci en Java pour pouvoir l'exécuter (toute l'infrastructure du langage est faite en Java). Jason propose aussi ses propres additions, telles que l'envoi de message ou les actions internes. Il est aussi possible de modifier le code Java grâce à une relation d'héritage, ce qui permet une plus grande liberté ;
- La création d'environnement, à travers une classe Java ;
- La configuration du projet dans un fichier dédié ;
- La plateforme faisant le lien entre les différents agents (envoi de messages et de signaux), mais aussi entre agents et environnement (perceptions et actions) ;

- Un Debugger\*, permettant d’inspecter l’esprit des différents agents, leurs croyances, leurs intentions.

### Modélisation de l’environnement

Comme dit précédemment, l’environnement en Jason est représenté par une classe Java, qui hérite de la classe `jason.environment.Environment`.

Cet environnement dispose de plusieurs méthodes permettant la communication avec les agents. Ces méthodes sont partiellement présentées dans le tableau ci-dessous.

Méthode	Effet
<code>executeAction(java.lang.String agentName, Structure act)</code>	Appelée quand un agent exécute une action sur l’environnement, Il est possible de récupérer l’action appelée et les arguments avec la Structure <code>act</code> , qui représente le <i>literal</i>
<code>clearPercepts(java.lang.String agName)</code>	Supprime les perceptions d’un agent en particulier
<code>clearPercepts()</code>	Supprime les perceptions communes à tous les agents
<code>addPercept(java.lang.String agName, Literal... per)</code>	Ajoute une perception à l’agent passé en paramètre
<code>removePercept(java.lang.String agName, Literal... per)</code>	Supprime la ou les perceptions passées en paramètres de l’agent

TABLE 2 – Principales méthodes de l’environnement pour la communication avec les agents

### Panneau de contrôle et utilisation du Debugger

Jason propose une fenêtre permettant de contrôler le déroulement du programme. Ce panneau est désactivable dans le fichier de configuration du projet. Il permet :

- d’afficher les logs des agents et du programme. La fenêtre devient la sortie standard du programme (à la place de la traditionnelle console) ;
- de mettre en pause le programme ;
- de créer ou détruire un agent ;
- de visualiser le code source des agents.

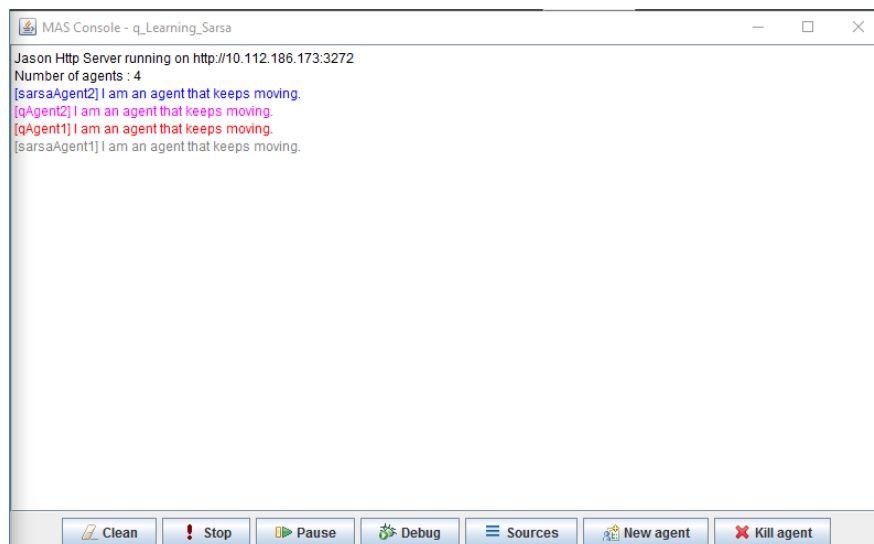


FIGURE 11 – Panneau de contrôle du MAS

L'utilisation du debugger est très simple : à tout moment, il est possible de mettre en pause l'exécution du programme pour inspecter les agents. Cet affichage permet de vérifier qu'un agent a bien reçu une perception ou que sa base de croyance (Belief Base) est bien à jour (et qu'une croyance n'a pas plusieurs valeurs).

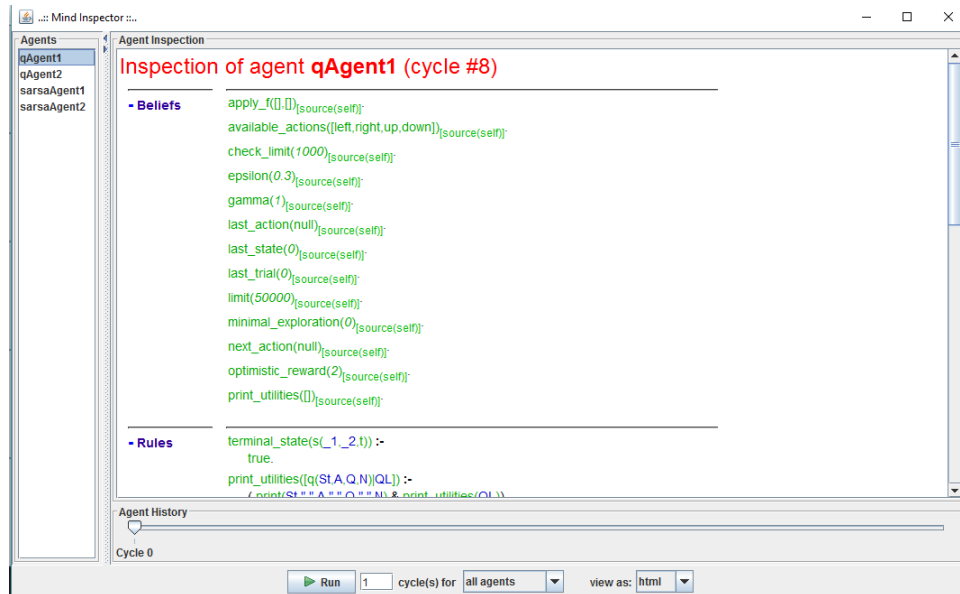


FIGURE 12 – Mind inspector de Jason

### Configuration d'un projet Jason

Un projet Jason a, par défaut, une structure assez simple. Les fichiers sources des agents (.asl) et de l'environnement (.java) sont séparés. Le fichier de configuration (.mas2j) se trouve à la racine du projet.

MAS q\_Learning\_Sarsa {

```

    infrastructure: Centralised // type d'infrastructure, centralised = Tous les agents sur un
    ↪ même ordinateur
    environment: complexEnv.MDPEnv //classe de l'environnement
    agents:
        qAgent qAgent.asl #2; // suit le format : nom_agent fichier_source [#nombre]
        sarsaAgent sarsaAgent.asl #2;
    aslSourcePath:
        "src/asl"; //Répertoire où chercher les fichiers AgentSpeak
}
```

Code 3 – Fichier de configuration du projet Q-learning/SARSA

### Architecture d'un agent

Les agents Jason sont bâtis sur le modèle BDI, qui est inspiré de la psychologie humaine. Ce modèle sépare ce qui caractérise un agent cognitif en trois parties distinctes :

- Ses croyances (*beliefs*) : c'est sur elles que reposent le monde vu par l'agent. Ce sont les règles pré-établies (des convictions par exemple) et ce qu'il perçoit du monde qui l'entoure (recueilli par des capteurs ou des sens). Ces données sont stockées dans une «base de données », souvent appelée Belief Base ;
- Ses désirs (*desires*) : ce que l'agent aimerait accomplir. Ce que les intentions visent à accomplir ;
- Ses intentions : ce que l'agent souhaite faire pour satisfaire ses désirs. Cela passe par l'exécution d'un plan et de ses actions (internes ou externes).

Les plans sont similaires à des fonctions, et leur sélection dépend des pré-conditions (souvent basées sur les croyances de l'agent). Voici des exemples de plan, pris dans l'agent utilisant le Q-learning, qui montrent plusieurs spécificités de la programmation en Jason (les commentaires ont été ajoutés pour faciliter la compréhension) :

```
//Exécute un mouvement, selon l'état S de l'agent
+!do_one_move(S) : true <-
  !det_policy(S,A); // trouve la prochaine action (A est initialisé après cet appel)
  A; //exécution de A : A est un atome qui peut être (left,right...)
  -+last_action(A). // Mise à jour de la Belief Base, la dernière action exécutée est A

//Determine l'action A pour l'etat S
+!det_policy(S,A) : true <-
  .findall([Q,A1,N],qvalue(S,A1,Q,N),Qs); // trouve toutes les Q-Values de l'état S et les stocke
  ⇐ dans QS
  ?apply_f(Qs,Fs); //Applique la fonction d'exploration et retourne le résultat dans Fs
  ?best_action(Fs,A). //Place la meilleure action trouvée dans Fs et l'unifie avec A
```

Code 4 – Exemples de plan de l'agent Q-Learning

Les agents raisonnent en cycle, et un schéma représentant ce cycle peut être trouvé en annexe.

#### 2.2.2 L'apprentissage machine

L'apprentissage par renforcement permet à un système d'apprendre l'utilité d'états dans lequel il peut se trouver, en association avec une action entreprise. Cette technique d'apprentissage prend ses racines dans la psychologie humaine, et a en partie été inspirée par l'expérience de la boîte de Skinner.

L'apprentissage par renforcement est principalement utilisé lorsque l'environnement est trop grand (voire infini) et que l'estimation des valeurs d'utilité doit être faite de manière régulière, c'est-à-dire ne pas attendre la fin d'une suite d'essais (jusqu'à ce que l'agent se trouve dans un état terminal).

Ceci suppose donc que l'environnement suit un modèle Markovien et plus précisément un processus de décision Markovien. Un modèle Markovien doit comporter :

- Un ensemble d'états  $S$  ;
- Un ensemble d'actions  $A$  ;
- Une fonction de récompense  $R(S,A)$  ;
- Une description d'une action pour un état donné.

La principale caractéristique d'un modèle de Markov est que les effets d'une action dans un état ne dépendent pas des états précédents.

### L'apprentissage passif

Dans l'apprentissage passif, un agent se contente d'estimer l'utilité de chaque politique qui lui est dictée. C'est comme ça qu'a été programmé le premier agent, baptisé TDL. Il ne prend pas de décision et se contente de suivre le chemin tracé.

La fonction d'apprentissage du TDL est la suivante :

$$V(s_t) \leftarrow V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

FIGURE 13 – Mise à jour de la valeur d'utilité d'une politique

où :

- $\alpha$  est la vitesse d'apprentissage. Elle est comprise entre 0 et 1. Plus elle tend vers 1, plus l'agent apprend vite.
- $\gamma$  est le facteur de réduction. Celui-ci influence l'importance des futurs récompenses par rapport aux récompenses immédiates. Un facteur de 0 fera que l'agent n'observe que la récompense immédiate, tandis qu'une valeur de 1 fera que l'agent pensera sur le long terme.

### L'apprentissage actif

Au contraire, dans l'apprentissage actif, le système cherche à optimiser ses actions. Il va, au fil du temps, apprendre quelles sont les meilleures politiques.

L'algorithme du Q-Learning reprend les bases du TDL. Cependant, lorsqu'il faut mettre à jour la Q-value (la valeur d'utilité), le Q-Learning va choisir la valeur maximale du prochain état.

Voici l'algorithme générique du Q-Learning, pris dans [5] :

```
Initialize  $Q(s, a)$  arbitrarily
Repeat (for each episode):
  Initialize  $s$ 
  Repeat (for each step of episode):
    Choose  $a$  from  $s$  using policy derived from  $Q$ 
    (e.g.,  $\epsilon$ -greedy)
    Take action  $a$ , observe  $r, s'$ 
     $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
     $s \leftarrow s'$ 
  until  $s$  is terminal
```

FIGURE 14 – Algorithme du Q-Learning

La méthode SARSA est une variante du Q-Learning. La différence majeure est que SARSA se sert de la Q-value de la prochaine action (du couple  $S', A'$ ) pour mettre à jour la valeur de l'état actuel. Cela reflète mieux la réalité de ce qui se passe dans un essai et propose peut-être donc une optimisation plus fidèle.

### 2.2.3 Autres technologies utilisées

Pour le développement, j'ai utilisé d'autres technologies qui le facilitent :

- l'Integrated Development Environment (IDE) Eclipse, avec le Plug-in\* Jason. Celui-ci permet de lancer une application Jason, avec ses composants spécifiques (MAS, console) ;
- Git, avec GitLab pour interface. Git est un système de gestion de fichier (*versioning*) permettant de sauvegarder l'avancée du travail sur un serveur distant, ainsi que collaborer de manière simple ;
- La bibliothèque Swing de Java, qui permet de créer des interfaces graphiques. Malgré son âge (et ses failles) j'ai choisi Swing car une grande source de documentation est disponible. J'ai de plus, avec le module d'IHM suivi en première année, une relative expérience avec ce framework.

Une autre technologie a été étudiée lors de ma phase de préparation : Common ARTifact infrastructure for AGents Open environments (CArtAgO). CArtAgO est basé sur le modèle *Agents & Artifacts*. Il introduit la notion d'artéfacts, pouvant influencer l'environnement. Ces artéfacts sont utilisés par les agents en tant que ressources ou outils. Un Pont\* vers Jason est déjà disponible avec la version de base de CArtAgO. Je n'ai pas utilisé CArtAgO pour les premières itérations. Cette technologie sera cependant très certainement mise en pratique lors du développement du jeu de poursuite.

## 2.3 Analyse de l'application

### 2.3.1 Maquettage

Afin d'obtenir le meilleur rendu possible, j'ai créé des maquettes. Elles m'ont permis de présenter de façon claire l'aspect général de l'application et les comportements possibles grâce à l'interface.

La figure 15 montre l'aspect de base de l'application, avant qu'une série d'essais n'ait été déclenchée. Le panel de gauche montre l'environnement, avec pour chacune des positions la politique de l'agent, la récompense, l'utilité estimée de l'état et le nombre de passages de l'agent. Les différentes couleurs représentent :

- Le blanc : un espace libre ;
- Le gris : un obstacle/mur ;
- Le rouge : un état terminal d'échec ;
- Le vert : un état terminal positif ;
- Le bleu : un état sélectionné par l'utilisateur pour la modification.

Le panel de droite est un formulaire qui permet de modifier un état sélectionné. J'ai pensé remplacer ce formulaire par une fenêtre pop-up au clic de l'utilisateur. Cependant, j'ai jugé cette solution intrusive et très redondante. Elle rendrait sûrement la modification de l'environnement pénible et longue. Avec un formulaire dans la même fenêtre, il est possible pour l'utilisateur de visualiser directement les informations de l'ensemble de l'environnement. Cela permet aussi d'obtenir de manière très simple et intuitive un système de multi-sélection des états pour une modification commune.

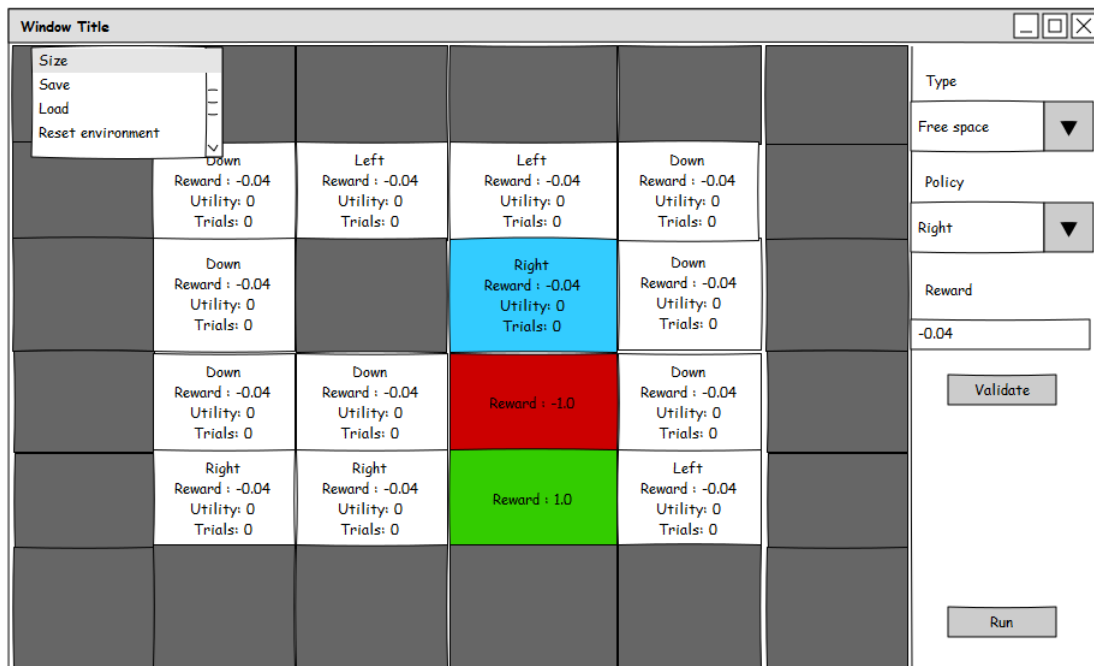


FIGURE 15 – Maquette de base de la première itération



### 2.3.2 Modèle de l'application

Pour pouvoir fonctionner correctement et de manière «saine», l'application se doit de reposer sur de bonnes bases. Cela est aussi nécessaire car un développement en cycle, avec un ajout de fonctionnalités régulier, signifie un retour sur le code constant. Si le code n'est pas maintenable, le travail en sera d'autant plus long et la correction de bugs plus difficile.

#### L'architecture MVC

Le modèle Model-View-Controller (MVC) permet une séparation claire des responsabilités au sein d'un programme. Ces dernières sont découpées en trois parties :

- Le modèle, s'occupe de la gestion des données et de la logique de l'application ;
- Le contrôleur gère et manipule le modèle en fonction des signaux reçus ;
- La vue présente le programme à l'utilisateur. Elle est mise à jour en fonction des changements du modèle.

Le programme Java interagissant avec un MAS, il a été nécessaire de transformer ce modèle afin de pouvoir communiquer avec les agents. Ainsi, le contrôleur assume aussi le rôle d'environnement.

Ce schéma présente la relation entre les différentes composantes de l'application :

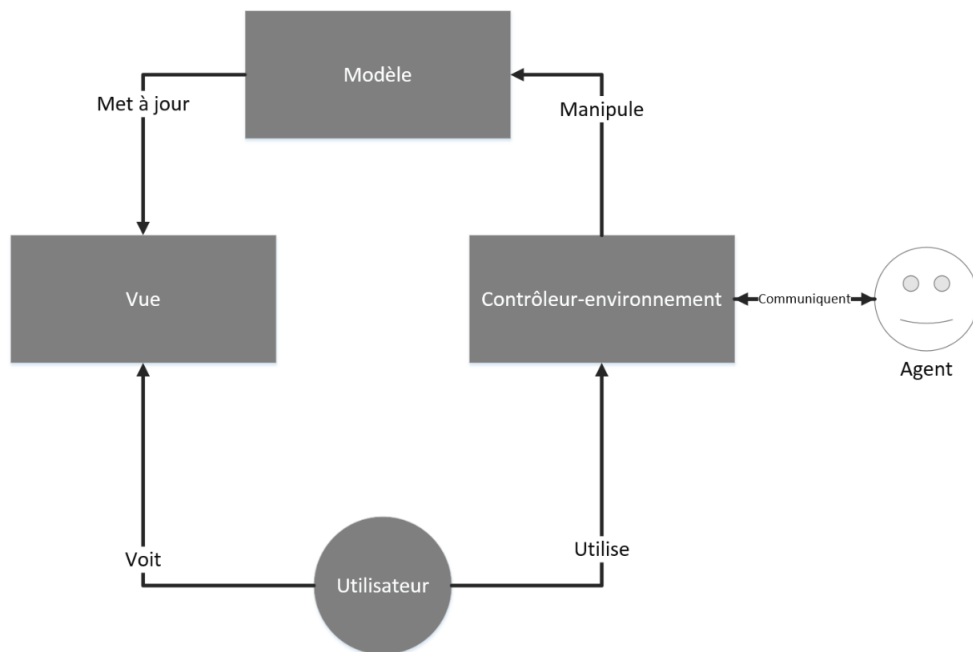


FIGURE 16 – Modèle MVC adapté

La modélisation UML de cette partie de l'application est la suivante :

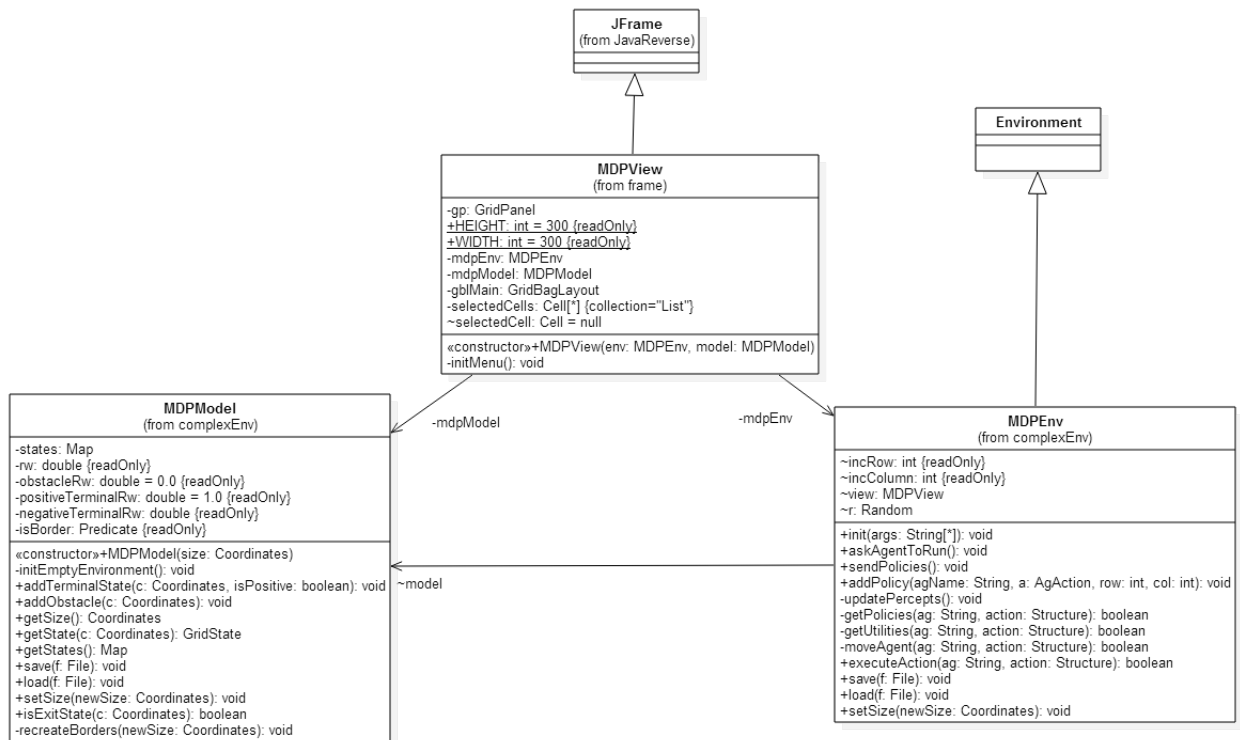


FIGURE 17 – Diagramme des principales classes du programme

### Le *pattern* Observateur

Afin de garder la vue à jour, j'ai choisi d'utiliser le *pattern* Observateur. Celui-ci établit un fort lien entre une classe observée et une classe observatrice. Dès lors d'un changement dans l'observé, l'observateur est notifié et peut adapter ses informations et son comportement en conséquence.

Cela est très utile pour s'assurer de la mise à jour d'une interface par rapport à son modèle. Ici, ce sont les boutons représentant les différents état de l'environnement qui sont mis à jour lors de la modification de leur contrepartie du côté modèle.

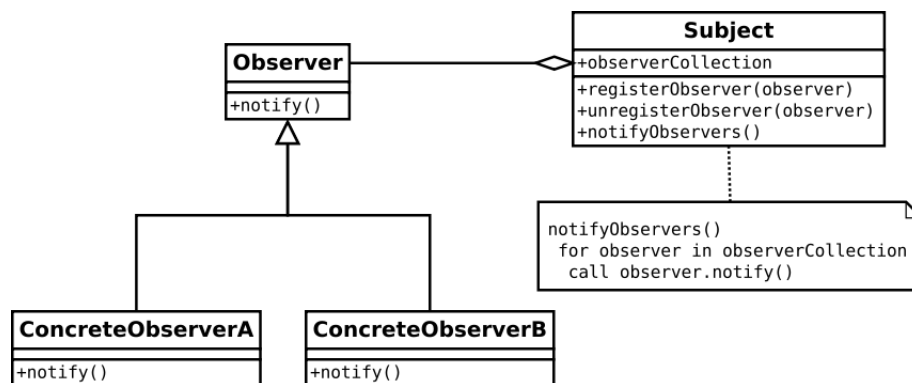


FIGURE 18 – Représentation du *pattern* Observateur

## 3 Réalisation et bilan

### 3.1 Première itération

#### 3.1.1 Création de l'environnement

##### Reprise du code existant

Mon premier travail, une fois l'analyse terminée, a été de reprendre le code existant. Celui-ci, bien que fonctionnel, présente plusieurs défauts :

- Toutes les données sont stockées au sein d'une même classe : les responsabilités ne sont pas clairement définies ;
- La représentation des données est fixe (et non modifiable après l'instanciation de l'environnement), ce qui rend le code peu flexible ;
- L'utilisation des types primitifs au lieu des structures de données (comme une classe) fait que l'information est dispersée dans plusieurs variables (tableaux).

Les données de l'environnement sont, dans le code initial, réparties dans plusieurs tableaux (booléens pour les obstacles, flottants pour les récompenses, etc). J'ai décidé de regrouper ces informations au sein d'une classe, car elles sont fortement liées. En effet, chaque élément des différents tableaux fait référence à un emplacement de l'environnement, un état.

J'ai donc créé la classe `GridState`, qui contient les attributs suivants :

- Une récompense, représentée par un *double* ;
- Un booléen, afin de savoir si l'emplacement est un obstacle ;
- Deux booléens afin de savoir si l'état est final (positif ou négatif) ;
- La politique que l'agent doit associer à cet état ;
- La valeur d'utilité dernièrement perçue par l'agent TDL, ainsi que le nombre d'essais effectués.

Pour améliorer la flexibilité de l'environnement, j'ai aussi décidé de reprendre le mode de stockage de ces données. Elles sont dans le code initial, stockées dans un tableau fixe. J'ai décidé de me reposer sur les mécanismes de généricité de Java. Ceux-ci proposent des conteneurs, fonctionnant de manière identique quel que soit le type d'objet passé en paramètre (il doit cependant être défini dès le départ).

Afin de faciliter la recherche des données, une carte (*Map*) a été choisie. Une carte permet d'associer une clé à une valeur. En connaissant la clé, il est possible de savoir si elle se trouve dans la carte et d'y accéder.

L'environnement étant un quadrillage, le choix de la clé fut assez simple. La classe *Coordinates* comporte deux membres, la ligne et la colonne. Chaque emplacement est donc identifié par sa ligne et sa colonne, dont le couple est unique.

## La GUI

Une grande partie du travail sur l'éditeur pour l'agent TDL était sur l'interface. J'ai tout d'abord repris mes maquettes afin d'analyser le lien possible entre composants graphiques et le modèle de données. Il est vite apparu que la partie gauche de la maquette représentait l'ensemble de l'environnement et que chacun de ses boutons était en lien avec un état particulier.

Il y a donc un fort couplage entre ces boutons (codé au sein de la classe *Cell*) et la classe *GridState*.

J'ai d'abord travaillé sur le panel «environnement», afin de vérifier que l'interface était intuitive et qu'elle répondait bien aux besoins d'affichage. Puis, j'ai réalisé le formulaire de modification d'état, une fois que le reste était fonctionnel.

L'environnement est modélisé par la classe *GridPanel*, qui hérite de *JPanel* (les panels Java Swing sont des conteneurs pouvant être placés au sein d'une fenêtre). Cette classe contient simplement un Tableau de *Cell*. Ce tableau est construit à partir de la classe *MDPModel* qui contient l'ensemble des états. C'est à l'instanciation du panel que se fait le lien entre une *Cell* et un *GridState*, grâce au *pattern* Observateur. Dès lors, lorsque le contrôleur modifie l'état, la vue en est aussitôt informée. Afin de garder un accès aux données, la *Cell* garde tout de même une référence vers cet état. Elle peut ainsi le transmettre lors de la modification au sein du formulaire par exemple.

Le formulaire fonctionne de la manière suivante :

- Si aucun état n'est sélectionné, le formulaire est désactivé (excepté le bouton pour lancer les essais) ;
- Lorsque l'utilisateur sélectionne un état, le formulaire s'active. Un utilisateur peut sélectionner plusieurs états (système de multi-sélection). La sélection se fait au clic, tout comme la désélection ;
- Les différentes valeurs sont renseignées par l'utilisateur. Certaines configurations sont impossibles. Par exemple, Il est impossible pour un emplacement étant désigné comme obstacle d'avoir une politique. Dans ce cas, la *combo box* (liste d'options) est verrouillée sur le choix «null» ;
- Lors de la validation, tous les états choisis sont modifiés et les cellules désélectionnées automatiquement, laissant l'utilisateur libre de faire d'autres modifications ou de lancer le programme.

L'interface comporte aussi une barre de menu, où les options utilisées moins fréquemment sont disponibles (voir Figure 19).



FIGURE 19 – Le menu de l'application

## Sauvegarde et chargement

Les méthodes de chargement et de sauvegarde fonctionnent de la manière suivante :

- À leur appel, les méthodes ouvrent un dialogue de type `JFileChooser`, qui permet de visualiser les dossiers du système de fichier. L'utilisateur sélectionne ou crée un fichier avec l'extension dédiée aux sauvegardes de l'environnement (`*.mdpenv`).
- La vue transmet la cible (de type `File`) de la sauvegarde ou du chargement à l'environnement, qui à son tour appelle une méthode du modèle pour finaliser la sauvegarde.

Seules les données du modèle sont sauvegardées. En effet, il est inutile de sauvegarder l'interface Swing ou la classe de l'environnement.

Ces méthodes tirent partie des flux Java pour faciliter la sauvegarde. Les informations sont ainsi sérialisées (les classes *GridState* et *Coordinates* implémentent toutes deux l'interface *serializable*, qui permet la sauvegarde) et stockées dans le fichier sous format binaire.

Ci-dessous est la méthode de la classe *MDPModel* qui effectue la sauvegarde :

```
public void save(File f) throws SaveException {
    Objects.requireNonNull(f);
    try (ObjectOutputStream oos = new ObjectOutputStream(new
↪   FileOutputStream(f))) {
        oos.writeObject(size);
        oos.writeObject(states);
    } catch (FileNotFoundException e) {
        throw new SaveException("The file was not found !");
    } catch (IOException e) {
        throw new SaveException("I/O Exception");
    }
}
```

Code 5 – Sauvegarde de l'environnement dans un fichier

Si un problème est perçu lors de la sauvegarde, une erreur remonte vers la GUI, qui fait apparaître un message d'erreur.

## Redimensionnement de l'environnement

La méthode de redimensionnement de l'environnement est critique pour le fonctionnement de l'application. Elle ne doit pas créer de problèmes et minimiser les erreurs de l'utilisateur.

Pour présenter le choix à l'utilisateur, j'ai choisi de créer une fenêtre pop-up, prenant le dessus sur le reste de l'interface. Le nombre de lignes et de colonnes sont modifiables. Une fois le choix validé, l'environnement est recréé, en suivant ces étapes :

- Si l'environnement rétrécit, les états superflus sont supprimés et retirés de la carte des états dans le modèle. Si l'environnement devient plus grand, des états par défaut sont rajoutés (espaces libres, avec une récompense égale à zéro) ;

- Les anciennes bordures sont détruites et de nouvelles sont créées, afin d’encercler l’environnement et conserver sa cohérence ;
- L’interface est recrée pour refléter ces changements ;
- La nouvelle taille effective est conservée dans le modèle.

J’ai choisi de conserver les anciens états afin de limiter les erreurs de l’utilisateur. Ainsi, s’il crée un environnement trop grand mais qu’il le configure tout de même, il lui est possible de le rétrécir sans perdre son travail.

J’ai choisi de complètement recréer l’interface après un redimensionnement car c’est la solution la plus simple : elle ne contient aucune donnée et sa création est assez rapide pour que le changement ne soit pas ressenti par l’utilisateur. Cela évite une redondance de code et un grand nombre de tests afin de vérifier la validité (le code emprunte un chemin déjà connu, le constructeur de *MDPView*).

### Rendu final

Le rendu final de l’interface est très proche de l’idée originale, présentée dans les maquettes. Certains détails ont été modifiés, comme l’affichage des politiques, maintenant présentées sous formes de flèches.



FIGURE 20 – L’application à la fin de l’itération, après un jeu d’essais de l’agent

### 3.1.2 Interactions entre l'agent et l'environnement

#### Envoi des politiques

Afin que l'agent puisse effectuer ses parcours dans l'environnement, il doit disposer des politiques liées aux emplacements qu'il va atteindre. Jusqu'ici, l'environnement étant fixe, celles-ci étaient stockées dans sa *Belief Base*, «en dur ». Ceci n'étant plus applicable, il a fallu trouver un moyen de communiquer les choix de l'utilisateur à l'agent. Ceci passe par la méthode *sendPolicies* de l'environnement qui appelle pour chaque état la méthode *addPolicy* :

```
//Envoie un literal correspondant à : policy(s(c.y,c.x),action)
public void addPolicy(String agName, AgAction a, Coordinates c) {
    final String action = getActionTextForAgent(a);
    final Literal s = ASSyntax.createLiteral("s", ASSyntax.createNumber(c.y),
↪ ASSyntax.createNumber(c.x));
    final Literal l = ASSyntax.createLiteral("policy", s,
↪ ASSyntax.createAtom(action));
    addPercept(agName, l);
}
```

Code 6 – Envoi d'une politique à l'agent

La réception de ces politiques déclenche un évènement au sein de l'agent. Cet évènement internalise les perceptions, qui seraient autrement effacées par la fonction de mise à jour de l'environnement. Elle supprime aussi les anciennes politiques associées à l'état.

#### Réception des résultats par l'environnement

Pour que l'interface puisse être mise à jour, l'agent doit transmettre ses résultats à l'environnement. Cela se fait par une action externe de l'agent.

L'agent exécute le plan suivant à la fin de la suite d'essais pour envoyer ses estimations des valeurs d'utilité :

```
+!send_utilities <-
    .findall(u(St,U,N),utility(St,U,N),UL);
    send_utilities(UL).
```

Code 7 – Envoi des utilités par l'agent TDL

Cette action est perçue par l'environnement dans la méthode *executeAction*, qui itère sur la liste envoyée en paramètre (UL) pour en extraire les données, grâce au système de *literals*.

**Démarrage d'un jeu d'essais**

Pour demander à l'agent de lancer l'exécution de son plan principal. Il est nécessaire que l'environnement envoie un signal. Cela suit la suite d'étapes suivante :

- L'utilisateur clique sur le bouton «Run » ;
- La vue signale à l'environnement cette intention ;
- L'environnement envoie les politiques que l'agent devra utiliser ;
- L'environnement envoie la perception *must\_run* :
- Quand l'agent reçoit cette perception, un évènement est déclenché, qui appelle le plan principal (*!start*) ;
- À la fin de l'exécution du plan, la perception est retirée de la *Belief Base* et l'agent peut à nouveau répondre à l'évènement.

Il est aussi nécessaire de «nettoyer »les informations d'une exécution afin de ne pas perturber les résultats suivants.



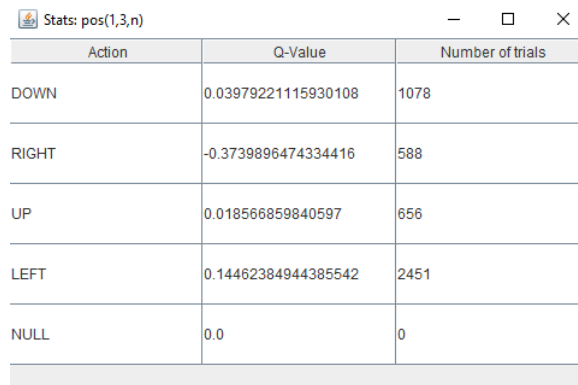
## 3.2 Améliorations et modifications

La deuxième itération étant très différente de la première (autre type d'agents, besoins d'interface différents), elle a été développée dans un projet séparé.

### 3.2.1 Adaptation de l'existant

Les agents Q-Learning et SARSA étant des agents actifs (qui testent différentes politiques d'eux-mêmes), l'affichage a dû être repensé. Les cellules de l'interface affichent maintenant la politique ayant la plus grande utilité. Cependant, il est aussi nécessaire de pouvoir visualiser les valeurs pour toutes les actions.

J'ai donc créé une nouvelle fenêtre qui se manifeste quand l'utilisateur effectue un clic droit. Elle contient toutes les informations que l'agent a acquises pour chaque action. Elle est, comme les cellules, mise à jour tous les 1 000 essais, quand l'agent envoie sa liste de valeurs Q (Q-values) à l'environnement. J'ai, encore une fois, repris le *pattern* Observateur qui répond à la demande de mise à jour à partir du modèle.



Action	Q-Value	Number of trials
DOWN	0.03979221115930108	1078
RIGHT	-0.3739896474334416	588
UP	0.018566859840597	656
LEFT	0.14462384944385542	2451
NULL	0.0	0

FIGURE 21 – La fenêtre de statistiques pour la position (1,3)

Les agents ne suivant plus une politique dictée, la *combo box* a été retirée du formulaire et la procédure d'envoi supprimée. La réception des valeurs d'utilité a été transformée pour accueillir les valeurs Q, qui ont au sein de l'agent une structure similaire.

### 3.2.2 Création de l'agent SARSA

Mon tuteur m'a aussi demandé de créer un nouveau type d'agent, qui est une variante de l'agent utilisant le Q-Learning. Je suis reparti du code du Q-Learning.

La fonction de mise à jour utilisant maintenant la prochaine action réelle, il a fallu repenser le code (le Q-learning choisissant toujours l'action ayant la plus haute valeur).

Les branches principales de la mise à jour ont maintenant des paramètres supplémentaires, afin de pouvoir accéder aux informations sur l'action suivante, ainsi que l'état. Celles-ci sont déclenchées une fois que l'agent a vérifié qu'une mise à jour était nécessaire (et non une création de la valeur).

```

//St1 : nouvel état, A1 : nouvelle action
//R : Récompense pour (St,A), Q_St : valeur Q pour (St,A)
//M1 : nouveau nombre d'essais pour (St,A)
//mise a jour pour un état non terminal
+!update_qvalue(St1,A1,_,St,R,Q_St,A,M1) : non_terminal_state(St) <-
  ?gamma(Discount_Factor);
  Learning_Rate = 60.0/(M1+59.0);
  ?qvalue(St1,A1,Q_St1,_);
  Q1 = Q_St + Learning_Rate *(R+Discount_Factor*Q_St1-Q_St);
  -qvalue(St,A,_,_);
  +qvalue(St,A,Q1,M1).

+!update_qvalue(,_,_,St,R,_,null,M1) : terminal_state(St) <-
  -qvalue(St,null,_,_);
  +qvalue(St,null,R,M1).

```

Code 8 – Mise à jour au sein de l'agent SARSA

Il a aussi fallu que je réadapte l'agent pour répondre au dilemme exploration/exploitation. Ce problème montre la nécessité pour un agent intelligent de trouver un équilibre entre l'exploration et l'exploitation de ce qu'il connaît déjà. Si l'agent explore trop, alors son savoir ne sera jamais vraiment utilisé pour des tâches utiles (pas d'optimisation). Au contraire, si l'agent n'utilise que les politiques qu'il considère optimales, il risque de passer à côté de meilleures options (risque de ne pas trouver l'optimisation).

Une méthode pour équilibrer les choix est la sélection  $\epsilon$  - *greedy*. Dans celle-ci,  $\epsilon$  est une variable comprise entre 0 et 1. À chaque sélection, une variable aléatoire est créée, elle aussi entre 0 et 1. Si elle est inférieure ou égale à  $\epsilon$ , une action aléatoire est choisie. Sinon, c'est la meilleure action connue qui est choisie.

Voici une implémentation (cas général) de la politique de sélection  $\epsilon$  - *greedy* :

```

//A1 = output parameter
+!epsilon_greedy_action_selection(Fs,St,A1): Fs == [] & epsilon(Eps) <-
  .random(Rand);
  !epsilon_greedy_action_selection(Fs,A1,Eps,Rand).
+!epsilon_greedy_action_selection(Fs,A1,Eps,Rand): Rand <= Eps <-
  !select_random_action(Fs,A1).
+!epsilon_greedy_action_selection(Fs,A1,Eps,Rand): Rand > Eps <-
  ?best_action(Fs,A1).

```

Code 9 – Sélection  $\epsilon$  - *greedy*

### 3.2.3 Environnement multi-agents

Le besoin de créer des données et de pouvoir les comparer a mené au remaniement de l'environnement pour qu'il puisse accueillir plusieurs agents.

Les agents n'ont pas eu à subir de changements majeurs. En effet, ils sont entraînés à travailler seuls et ne se gênent pas.

C'est donc l'environnement qu'il a fallu modifier. L'environnement stocke la position de l'agent dans la version avec un seul agent. Il a donc fallu créer un conteneur pour pouvoir stocker les différentes positions et transmettre les bonnes perceptions à l'agent exécutant l'action (et non pas que lui et son collègue partagent la même position, ce qui rendrait les résultats faux et le code instable).

Pour pouvoir retrouver la bonne position de l'agent, j'ai créé une carte ayant pour clé le nom de l'agent et pour valeur un objet *AgentData*, qui contient cette position, mais aussi d'autres variables utilisées pour la génération de graphiques.

Les noms des agents sont récupérés avant le lancement d'un jeu d'essais, et la carte initialisée.

Les agents sont cependant toujours créés dans le fichier de configuration `.mas2j`.

J'ai choisi de limiter la mise à jour de l'interface à un seul agent, afin d'éviter les conflits et la mise à jour multiple (et donc l'impossibilité pour l'utilisateur de suivre l'évolution des valeurs).

### 3.2.4 Export des données

Afin d'exporter les données, j'ai choisi d'utiliser le format CSV. Ce format est très simple à écrire et à lire. Il représente une grille de données, dont chaque élément est séparé par un caractère pré-défini (« ; » dans mon cas) et les lignes par un saut de ligne (« \n »). Ce format est facilement interprétable par un tableur sans manipulation particulière et son utilisation très répandue.

Le principal objectif de cet export est de pouvoir voir l'évolution des valeurs Q et du nombre d'essais au fil du temps. Il est donc nécessaire de stocker ces données à chaque fois que les agents notifient l'environnement. C'est la classe *AgentData* qui joue ce rôle. Comme expliqué précédemment, une instance existe pour chaque agent, ce qui permet de séparer les résultats.

La succession de valeurs Q (ou du nombre d'essais) est stockée dans une liste où l'élément en queue est la dernière valeur reçue. Ces listes sont disponibles pour chaque état combiné à une action.

J'ai choisi de sauvegarder valeurs Q et essais dans un même élément du fichier CSV, séparés par un espace. Ceci permet à la lecture, d'associer les deux. C'est aussi sûrement la méthode la plus simple pour la lecture du fichier.

Le code ci-dessous permet de transcrire les données (objets et primitifs Java) en texte, prêt à être sauvegardé dans un fichier CSV. J'ai choisi d'utiliser un *StringBuilder*, car c'est la manière la plus efficace de concaténer des chaînes de caractères en Java.

```

public String getQValuesCSV() {
    final StringBuilder sb = new StringBuilder();
    for (final MapKey mp : qValues.keySet()) {
        final List<Double> values = qValues.get(mp);
        final List<Integer> trials = trialNumbers.get(mp);
        if (values != null && !values.isEmpty() && trials != null && !trials.isEmpty()) {
            ↪ //vérification que les listes ont été trouvées et contiennent des valeurs
                sb.append(mp.coordinates.y + " " + mp.coordinates.x + " " + mp.action);
            ↪ // premier element de la ligne = position + action
                for (int i = 0; i < values.size(); i++) {
                    sb.append(";").append(values.get(i)).append(" ").append(trials.get(i));
                }
                sb.append("\n");
        }
    }
    return sb.toString();
}

```

Code 10 – Récupération des données d'un agent au format CSV pour la sauvegarde

Il était aussi nécessaire que cet export marche pour de multiples agents. Afin de rendre la lecture plus simple, les résultats de chaque agent sont stockés dans un fichier différent.. Afin que cela soit plus intuitif, l'utilisateur saisit un nom de sauvegarde : les fichiers suivront le format *sauvegarde\_nomagent.csv*

### 3.2.5 Génération de graphiques

La prochaine étape de l'export et de l'exploitation des données est la génération des graphiques.

Il a fallu d'abord choisir la technologie qui puisse répondre aux besoins. J'ai tout d'abord regardé les bibliothèques Java (afin de garder une cohérence et d'éviter la dispersion des langages). Cependant, celles qui sont libres et gratuites ne permettent pas de générer des graphiques de manière aisée. Le format Excel étant la propriété de Microsoft, il m'a semblé logique de regarder au sein des bibliothèques C#. C'est là que j'ai trouvé EPPlus, un module Open Source et gratuit permettant de manipuler les fichiers .xlsx (le nouveau format Excel).

La mise en place du projet de test fut très simple grâce au gestionnaire de paquets NuGets de Visual Studio. Le langage ayant une syntaxe proche du C/C++ (et par extension du Java), l'adaptation a été très rapide. Certaines spécificités m'ont été très utiles, comme les requêtes Language-Integrated Query (LINQ), qui intègrent un langage déclaratif (très proche du Structured Query Language (SQL)) pour manipuler les collections d'objets de manière rapide et claire.

Le projet est développé comme un outil à part, le Java et le C# étant incompatibles. Cela permet aussi une plus grande souplesse. Le programme C# se sert des fichiers CSV générés pour créer les graphiques. Il les lit, stocke les informations, les ordonne et commence la génération.

La génération fonctionne de la façon suivante : pour chaque agent, un fichier est créé, comme pour l'export CSV. Pour chaque état (position), une feuille de travail Excel est créée. Les valeurs récupérées dans le fichier CSV sont ensuite réécrites dans le fichier Excel (nécessaire à la création des graphiques). Enfin, les différents graphiques sont générés.

Plusieurs types de graphiques sont disponibles :

- Une représentation de l'évolution des valeurs Q et du nombre d'essais, avec une double graduation ;
- Une comparaison des valeurs Q pour chaque action d'un état ;
- Un «camembert » montrant la répartition finale du nombre d'essais.

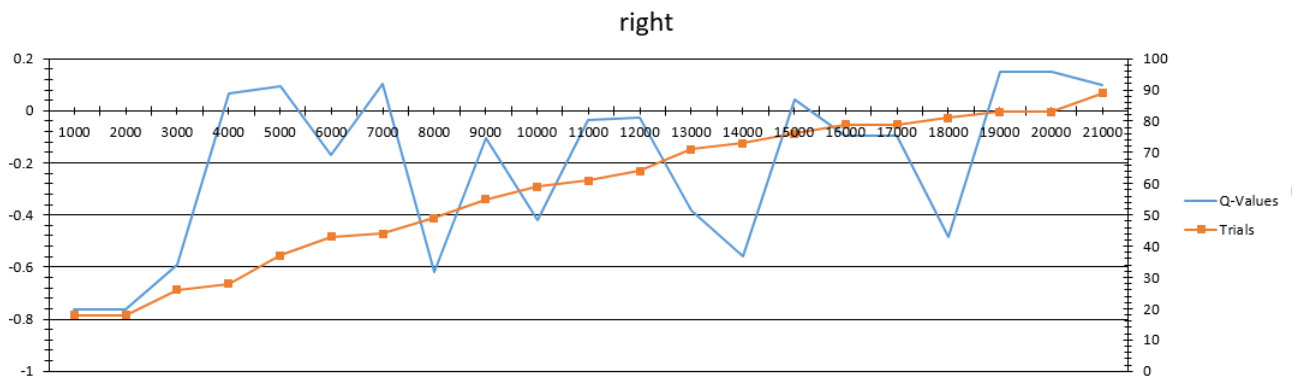


FIGURE 22 – Graphique généré pour l'action *right* de l'état (1,3)

L'image ci-dessus montre, sur 21 000 passages cumulés (sur tous les états) d'un agent, l'évolution liée à l'action *right* de l'état (1,3). Celle-ci le conduira vers un état peu utile (encerclé par des murs et un état d'échec). Si l'action échoue, et qu'il se dirige vers le bas, il sera tout aussi proche de l'état d'échec, ceci explique la valeur Q souvent négative et les «sursauts» de valeurs.

## 3.3 Poursuite et bilan

### 3.3.1 Résultats

L'application est fonctionnelle et produit les résultats attendus. L'agent SARSA a le comportement attendu, comme vu dans [5].

L'application pourrait bien sûr être améliorée, afin de présenter plus de résultats, et de données exploitables.

Il serait par exemple possible de repenser l'interface pour pouvoir accéder aux valeurs de chaque agent, ce qui n'est pas le cas actuellement.

D'autres méthodes de sélection d'actions pourraient être implémentées pour l'agent SARSA, comme une sélection avec un epsilon décroissant (qui ferait en sorte que l'agent choisisse de plus en plus la meilleure action, car il penserait avec le temps connaître ce qui est optimal).

### 3.3.2 Poursuite

Dans les trois semaines qui restent, mon attention sera portée sur une nouvelle expérience à réaliser. Celle-ci, déjà évoquée, reprend le principe d'un jeu de gendarmes et voleurs, où un agent doit en attraper un autre. Voici les spécificités et pistes exploitables présentées par mon tuteur :

- Dans un premier temps, seul le gendarme sera un agent intelligent ;
- La récompense sera dynamique pour le gendarme, elle sera attribuée en fonction de sa distance avec le voleur ;
- Le modèle est en cycle. À chaque fois que le gendarme effectue une action, le voleur doit aussi faire une action ;
- Implémenter un système d'orientation pour le voleur. L'action du voleur pourra être d'avancer ou de tourner de 90 degrés ;
- L'utilisation d'artéfacts. Le voleur pourra utiliser une arme pour détruire le policier. Cette arme sera considérée comme un outil pour le voleur, et son utilisation sera régie en interne. Elle pourra par exemple dicter la chance de s'enrayer ou le nombre de munitions restantes. Cela rendra son utilisation plus intuitive, mais aussi plus réaliste ;
- Le fait de se faire toucher par cette arme pourrait être un état final d'échec pour le gendarme ;
- Un système d'informations pourrait être implémenté, afin que le gendarme ait une idée de la position du voleur.

Je devrai aussi présenter ce que j'ai fait lors de mon stage au groupe IDS le lundi 6 juin.

Mon tuteur m'a aussi demandé d'être co-auteur d'un article qu'il va publier. Celui-ci présentera les avancées achevées depuis la publication de l'article présenté en première partie. J'aurai à présenter les outils que j'ai créés et l'assisterai dans la rédaction. Il devrait me présenter un brouillon début juin, où je pourrai alors contribuer et fournir un retour critique.

### 3.3.3 Difficultés rencontrées

J'ai lors de ce stage rencontré plusieurs difficultés.

Tout d'abord, la notion d'agents et d'environnement n'a pas été évidente à comprendre. Les mécanismes de programmation de Jason sont de prime abord très différents de ceux vus à l'IUT. Il m'a donc fallu un long temps d'adaptation et de compréhension avant de pouvoir produire quelque chose d'intéressant.

Une autre difficulté a été de trouver des ressources exploitables afin que je puisse comprendre la Programmation Orientée Agent. En effet, Jason étant un langage peu utilisé, il existe très peu de tutoriels et exercices.

Enfin, le fait de développer seul l'application m'a parfois confronté à un manque de recul, qui a pu abaisser la qualité de ce que j'ai développé.

### 3.3.4 Progression

Malgré ces difficultés, ce stage a été très enrichissant. Il m'a ouvert l'esprit sur l'aspect général de la programmation, en m'introduisant aux concepts des agents et de l'environnement.

Le fait de travailler seul, bien que difficile, m'a été bénéfique et m'a permis de renforcer mon esprit critique envers moi-même.

## Conclusion

J'ai, au cours de ces 7 semaines, procédé au développement d'une application rendant le contrôle de l'apprentissage d'agents Jason plus simple. Cela est passé par une phase d'apprentissage, puis d'analyse et enfin de programmation. Cette application vient renforcer les travaux déjà existants effectués par mon tuteur.

Dans les semaines restantes, je vais travailler sur un jeu de gendarmes et voleurs (*Pursuit-evasion game*), qui introduira de nouveaux concepts à appliquer aux agents Jason. J'assisterai aussi M. Bădică pour l'écriture de son article.

Ce stage m'a permis de consolider les bases de l'informatique, étudiées à l'IUT. J'ai aussi découvert de nouvelles technologies et concepts, qui m'étaient jusqu'alors inconnus. Malgré les difficultés rencontrées, cette période en milieu professionnel a renforcé mon autonomie et ma prise de décisions.

Cette immersion a été très enrichissante, tant sur le plan professionnel que personnel. La pratique dans des domaines plus théoriques que ceux étudiés à l'IUT m'a conforté dans le choix de continuer mes études, afin de découvrir d'autres technologies et orientations.



---

## Sources documentaires

### Références

- [1] Stuart RUSSELL et Peter NORVIG : *Artificial Intelligence: A Modern Approach*. Pearson, 2009.
- [2] Site de l'université de Craiova. <http://www.ucv.ro>.
- [3] Informations générales sur craiova. <https://en.wikipedia.org/wiki/Craiova>.
- [4] Site du projet jason. <http://jason.sourceforge.net/>.
- [5] Unsw - reinforcement learning. <https://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html>.
- [6] Amelia BĂDICĂ, Costin BĂDICĂ, Mirjana IVANOVIĆ et Dejan MITROVIĆ : Approach of Temporal Difference Learning using Agent-Oriented Programming. *Agent Technology for Ambient Intelligence*, 2015.
- [7] Rémi MUNOS : Introduction à l'apprentissage par renforcement.
- [8] André Filipe de MORAES BATISTA, Maria das GRAÇAS BRUNO MARIETTO, Wagner Tanaka BOTELHO, Guiou KOBAYASHI, Brunno dos PASSOS ALVES, Sidney de CASTRO et Terry Lima RUAS : Principles of agent-oriented programming, multi-agent systems - modeling, control, programming, simulations and applications, 2011.
- [9] Amelia BĂDICĂ, Costin BĂDICĂ, Maria GANZHA, Mirjana IVANOVIĆ et Marcin PAPRZYCKI : Experiments with Multiple BDI Agents with Dynamic Learning Capabilities, 2016.
- [10] John Wiley & Sons LTD : Programming Multi-Agent Systems in AgentSpeak Using Jason, 2007.
- [11] Andrea OMICINI et Michele PIUNTI : Programming Intentional Agents in AgentSpeak(L) and Jason, 2007.
- [12] Bob GIVAN et Ron PARR : An introduction to markov decision processes.

---

# Liste des figures, tableaux et procédures

## Table des figures

1	Logo de l'université de Craiova . . . . .	3
2	Carte des partenariats Erasmus+ de l'université en 2013 . . . . .	3
3	Façade de la faculté . . . . .	4
4	Le logo IDS . . . . .	5
5	L'environnement utilisé dans l'article . . . . .	7
6	Échanges entre un agent et son environnement dans le cadre de l'apprentissage	7
7	Probabilités de succès ou d'échec liées à une politique . . . . .	8
8	Fonction d'utilité de l'état (3,3) pour un agent TDL, avec l'action <i>left</i> assignée . . . . .	9
9	Use case initial de l'application . . . . .	12
10	Logo simplifié de Jason . . . . .	14
11	Panneau de contrôle du MAS . . . . .	15
12	Mind inspector de Jason . . . . .	16
13	Mise à jour de la valeur d'utilité d'une politique . . . . .	18
14	Algorithme du Q-Learning . . . . .	18
15	Maquette de base de la première itération . . . . .	20
16	Modèle MVC adapté . . . . .	21
17	Diagramme des principales classes du programme . . . . .	22
18	Représentation du <i>pattern</i> Observateur . . . . .	22
19	Le menu de l'application . . . . .	24
20	L'application à la fin de l'itération, après un jeu d'essais de l'agent . . . . .	26
21	La fenêtre de statistiques pour la position (1,3) . . . . .	29
22	Graphique généré pour l'action <i>right</i> de l'état (1,3) . . . . .	33
23	Schéma de raisonnement d'un agent Jason . . . . .	VIII

---

## Liste des codes sources

1	Obstacles dans l'application initiale . . . . .	8
2	Belief Base de l'agent TDL . . . . .	8
3	Fichier de configuration du projet Q-learning/SARSA . . . . .	16
4	Exemples de plan de l'agent Q-Learning . . . . .	17
5	Sauvegarde de l'environnement dans un fichier . . . . .	25
6	Envoi d'une politique à l'agent . . . . .	27
7	Envoi des utilités par l'agent TDL . . . . .	27
8	Mise à jour au sein de l'agent SARSA . . . . .	30
9	Sélection $\epsilon$ - <i>greedy</i> . . . . .	30
10	Récupération des données d'un agent au format CSV pour la sauvegarde . . . . .	32

---

## Acronymes

**BDI** Belief-Desire-Intention. 6, 8, 14, 17

**CARTAgO** Common ARTifact infrastructure for AGents Open environments. 19

**CSV** Comma-separated Values. 11, 31–33

**GNU LGPL** GNU Lesser General Public License. 14

**GUI** Graphical User Interface. 10–12, 24, 25

**IDE** Integrated Development Environment. 19

**IDS** Intelligent Distributed Systems. 5, 10

**LINQ** Language-Integrated Query. 32

**MAS** Multi-Agent System. 6, 14, 21

**MVC** Model-View-Controller. 21

**POA** Programmation Orientée Agent. 6, 10

**POO** Programmation Orientée Objet. 6

**SARSA** State-Action-Reward-State-Action. 11, 19, 29

**SQL** Structured Query Language. 32

**TDL** Temporal Difference Learning. 11, 24

**UX** User eXperience. 10

---

## Glossaire

**Apprentissage par renforcement** Technique d'apprentissage machine visant à maximiser le gain reçu par un agent et optimiser ses actions au fur et à mesure du temps. Elle est particulièrement utilisée quand l'agent n'a pas connaissance du modèle ou que celui-ci est très large, voire infini. 6

**Belief-Desire-Intention** Modèle logiciel utilisé pour représenter les croyances (beliefs), les désirs (desires) et intentions d'un agent. Il s'inspire de la théorie de Michael Bratman sur le raisonnement pratique. 6

**Comma-separated Values** Extension de fichier indiquant que celui-ci contient des données délimitées par un séparateur, la virgule. Ce type de fichier est très utilisé car facile à construire et est interprété par les tableurs. 11

**Debugger** Programme informatique permettant de tester et trouver les bugs dans un autre programme. 15

**GNU Lesser General Public License** Licence permettant à l'utilisateur final de modifier une partie du code de l'application : les composants utilisant eux aussi cette licence. La partie propriétaire de l'application n'a pas à être obligatoirement distribuée. 14

**Graphical User Interface** Interface permettant à l'utilisateur et le système de dialoguer. L'utilisateur peut utiliser la souris pour émettre des signaux et modifier le système. L'interface comporte des images et autres éléments graphiques pour transmettre des informations. 10

**Multi-Agent System** Système regroupant plusieurs agents au sein d'un environnement commun. Les agents peuvent communiquer entre eux et échanger des informations et perceptions. 6

**Open Source** Un projet est open source quand son code source est disponible au public et peut être redistribué. 14

**Optimisation de la récursion terminale (*Tail Call Optimization*)** Processus d'optimisation de certaines fonctions récursives permettant de transformer les appels successifs en itérations, et ainsi économiser la pile d'exécution du programme et éviter les débordements de pile (*Stack overflows*). 9

**Plug-in** Composant logiciel ajoutant des fonctionnalités à un programme existant. 19

**Politique** Association entre un état et une action. 6, 8, 11

**Pont** En anglais, bridge. Désigne une partie d'un logiciel ou d'une librairie qui lui permet d'être utilisé avec un ou une autre. 19

**Programmation Orientée Agent** Paradigme de programmation reposant sur l'utilisation d'agents informatiques pour réaliser des tâches.. 6

**Programmation Orientée Objet** Principe de programmation selon lequel des objets encapsulés possèdent un état (des attributs) et des opérations. 6

---

**Refactor** Mot anglais désignant l'action de retravailler un code existant pour le rendre plus maintenable et pratique, sans pour autant affecter son comportement extérieur. 12

**Scaling** Mot anglais désignant la mise à l'échelle d'une chose. Ce mot est très utilisé dans le monde informatique pour désigner la capacité d'un système ou d'une application à s'adapter à des besoins changeants (souvent de plus en plus grands). 12

**Use Case** Liste d'actions définissant les cas d'utilisation d'un logiciel en relation avec un ou plusieurs acteurs. Typiquement représenté sous forme de diagramme. 12

**User eXperience** Acronyme faisant référence à l'expérience qu'a une personne avec un logiciel, c'est-à-dire les sensations qu'il éprouve et la capacité du logiciel à répondre à ses besoins. 10

---

# Annexes

## Architecture du projet (deuxième itération)

L'arborescence du projet Q-Learning/SARSA est la suivante :

```
q_Learning_Sarsa
├── bin
├── results
│   ├── csv
│   │   └── *.csv
│   └── xls
│       └── *.xlsx
├── src
│   ├── asl
│   │   ├── learningAgent.asl
│   │   ├── qAgent.asl
│   │   └── sarsaAgent.asl
│   └── java
│       ├── gui
│       │   ├── component
│       │   │   └── Cell.java
│       │   ├── frame
│       │   │   ├── MDPView.java
│       │   │   └── StateStatsFrame.java
│       │   └── panel
│       │       └── GridPanel.java
│       └── mdp
│           ├── AgentData.java
│           ├── Coordinates.java
│           ├── GridState.java
│           ├── MDPEnv.java
│           └── MDPModel.java
├── tools
│   └── chartGen.exe
└── q_learning_Sarsa.mas2j
```

## Schéma de raisonnement d'un agent Jason

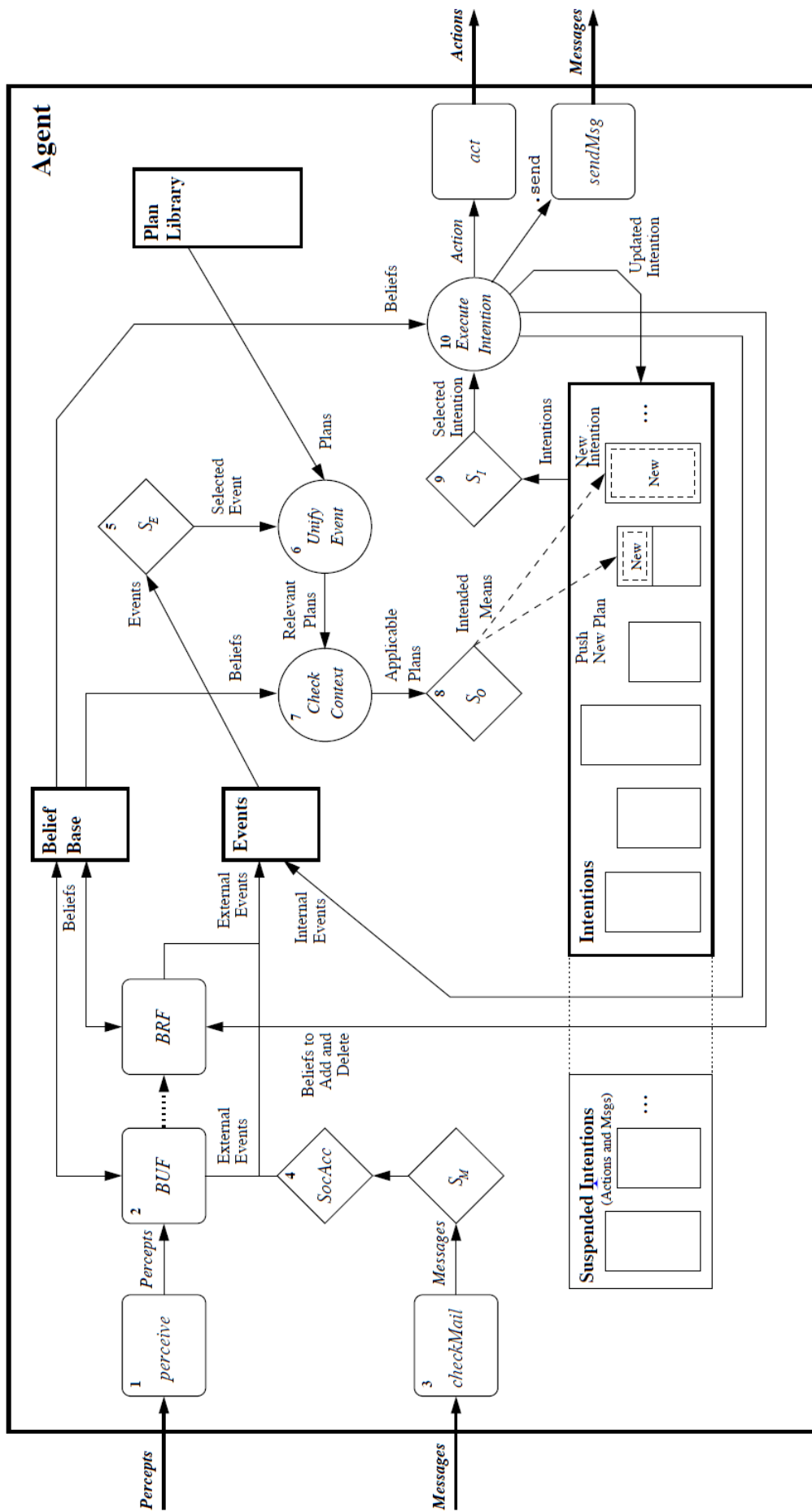


FIGURE 23 – Schéma de raisonnement d'un agent Jason