Université de Caen
UFR de Sciences
Département d'Informatique
Bâtiment Science 3, Campus II

LCSTALK

UN OUTIL DE MODÉLISATION ET D'ANALYSE DE SYSTÈMES DE CLASSEURS MONO ET MULTI-AGENTS

Stage de DEA réalisé au  à CAEN

Maître de stage : SERGE STINCKWICH
Période du stage : 1er avril – 20 septembre 2004

Rapport rédigé par
MICHAËL PIEL
mpiel@etu.info.unicaen.fr
DEA Intelligence Artificielle et Algorithmique

Le 28 septembre 2004

Remerciements

Ce travail a été rendu possible par le soutien direct ou indirect de nombreuses personnes que je tiens à remercier ici.

Tout d'abord, je remercie tout particulièrement Serge STINCKWICH, mon maître de stage, sans qui ce travail n'aurait pas été possible, et qui, par ses conseils et son expérience dans les domaines des systèmes multi-agents et des systèmes de classeurs, a positivement contribué à l'élaboration de ce rapport.

Je tiens à remercier Marinette REVENU pour me faire l'honneur de participer au jury de la soutenance.

Enfin, je souhaite remercier Olivier SAGIT et Samir SAIDANI pour leurs remarques éclairées et leur constant soutien durant toute la durée de ce stage.

Table des matières

Introduction

Ce rapport présente les détails de mon stage de DEA effectué au laboratoire GREYC-MAD (Modèles, Agents, Décision) pendant la période avril – septembre 2004.

Cadre du stage

Le cadre d'application de ce stage est celui du projet MAAM[1] (acronyme récursif pour Molécule = Atome | (Atome + Molécule)), où l'on s'intéresse à de nouvelles architectures de robotique collective, basées sur des composants auto-organisés. L'idée maîtresse est de définir des composants élémentaires nommés atomes robotiques permettant une construction rapide de structures complexes. La rapidité de configuration venant de techniques d'auto-organisation qui sont intégrées dans ces atomes. Les atomes robotiques sont conçus sur le principe des atomes de la physique, c'est-à-dire disposant de degrés de connectivité qui leur permettent de s'assembler pour former des molécules qui peuvent à tout moment se décomposer pour se recomposer en une autre molécule en fonction de la tâche à réaliser. Dans le cas de ce projet, un atome sera une structure mécanique à six pattes, chacune d'elles pouvant se solidariser deux à deux avec d'autres atomes. Les six pattes sont réparties autour d'un noyau sphérique suivant les trois directions de l'espace comme sur la figure ci-dessous.

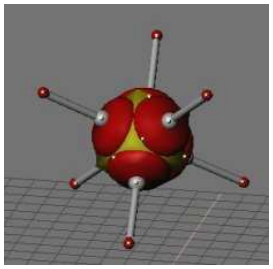

FIG. 1 – Représentation d'un atome.

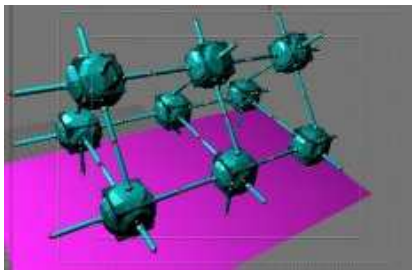

FIG. 2 – Représentation possible d'une molécule.

Débuté en janvier 2003, ce projet a pour objectif qu'au bout de trois ans, un ensemble d'atomes puisse réaliser un certain nombre de tâches élémentaires

[1] http ://www.iutc3.unicaen.fr/serge/ProjetMAAM

telles que l'assemblage en molécule simple et le passage d'un obstacle comme un cube de taille légèrement supérieure à la molécule.

Description du stage

Les systèmes de classeurs sont une approche d'apprentissage par renforcement similaire à l'approche "processus décisionnels de Markov". Ils offrent l'avantage par rapport à ces derniers d'avoir des capacités de généralisation plus importantes et de nécessiter moins d'états pour représenter leur environnement.

Du fait de ces avantages, l'équipe MAD compte utiliser les systèmes de classeurs dans ses travaux faisant partie du projet MAAM. C'est pourquoi, Il nous a semblé utile de réaliser une plateforme modulaire pour systèmes de classeurs nous permettant de tester différentes stratégies sur une multitude d'environnements, que ce soit en mono ou en multi-agents.

Par ailleurs, les systèmes de classeurs étant connus pour ne pas être adaptés à une problématique multi-agents, nous avons d'abord essayé d'implanter une méthode générale appelée méta-apprentissage sensée améliorer les performances de ces systèmes, puis nous avons tenté d'adapter un récent framework nommé CLRI dans le but d'analyser ces systèmes.

Ainsi, je commencerai par donner une description des systèmes de classeurs en général ainsi que de trois d'entre eux les plus connus qui sont ZCS, XCS et ACS. Je décrirai brièvement ce qu'est le méta-apprentissage et ce que l'on en a retiré. Puis je présenterai LCSTALK, un framework modulaire pour les systèmes de classeurs que j'ai réalisé. Enfin, j'exposerai une première tentative d'adaptation du framework CLRI aux systèmes de classeurs.

Chapitre 1

Systèmes de classeurs

1.1 Principaux mécanismes des LCS

Les LCS sont des systèmes à base de règles qui permettent de résoudre des problèmes d'apprentissage par renforcement et d'apprentissage latent.

Un exemple d'environnement utilisé comme test pour l'apprentissage avec les LCS est présenté à la figure 1.1. Dans celui-ci, l'agent doit apprendre à rejoindre la position de récompense en un minimum de déplacements. Par déplacement, on entend ici un mouvement d'une case à une autre ou un déplacement vers un mur (auquel cas l'agent ne bouge pas).

L'agent est situé dans un environnement et a une boucle sensori-motrice qui lui permet de percevoir et d'agir sur son milieu à chaque instant. La situation dans laquelle se trouve l'agent est décrite par un ensemble d'attributs de perception locale. Dans notre exemple de la figure 1.1, cette perception locale est [01001101]. On décrit par convention l'environnement à partir de la case située au nord de l'agent en tournant dans le sens des aiguilles d'une montre.

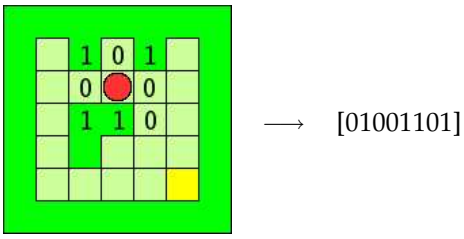

FIG. 1.1 – Attributs et situation.

Les règles de comportement des LCS sont appelées « classeurs ». Chaque classeur est composé au moins d'une partie `condition`, d'une partie `action` et d'une `valeur sélective` comme représenté à la figure 1.2. L'agent recherche dans son ensemble de règles si une ou plusieurs conditions s'apparient avec sa perception courante. S'il en trouve, il choisit de réaliser l'action qui correspond au classeur qui possède la plus forte valeur sélective (celle qui lui apportera potentiellement la plus grande récompense). Sinon, une nouvelle

règle est créée pour cette situation et on réalise une action au hasard. La valeur sélective représente la capacité du classeur à résoudre un problème.

Condition	Action	Valeur sélective
[01#01#01]	↑	69

FIG. 1.2 – Constitution d'un classeur type.

La boucle sensori-motrice à donc plus ou moins la forme suivante :

1. Perception par l'agent de sa situation σ dans l'environnement.
2. Création d'un `match set` (ou ensemble d'appariement) en appariant cette situation σ avec tous les classeurs du système.
3. Choix d'une action à exécuter parmi les classeurs du `match set`.
4. Création d'un `action set` (ou ensemble d'action) avec tous les classeurs du `match set` contenant cette action.
5. Exécution de l'action par l'agent.
6. Exécution de l'algorithme génétique (qui peut être exécuté à d'autres endroit de la boucle, suivant le LCS utilisé).
7. Exécution d'un algorithme de renforcement (bucket Brigade ou autre) si le système reçoit une récompense de l'environnement.

La valeur sélective des classeurs est ajustée à chaque pas de temps et le système ne conserve pas de mémoire des informations produites dans le passé par la boucle sensori-motrice. Le processus d'apprentissage est donc incrémental et permet d'aborder des environnements changeants.

1.1.1 La généralisation

Par rapport aux algorithmes d'apprentissage par renforcement de la famille Q-learning, les LCS ont l'avantage de posséder un mécanisme de généralisation qui exploite les régularités dans la dynamique d'interactions entre l'agent et l'environnement.

Le principal avantage de la généralisation est qu'un classeur puisse s'apparier à plusieurs situations. Ceci est réalisé par l'introduction d'un symbole *passe-partout* représenté par le caractère #. Cet attribut sert à remplacer un 0 ou un 1 de la partie condition. Cela signifie que l'on peut ignorer cet attribut dans le processus d'appariement d'un classeur avec la situation courante de l'agent.

Ainsi, un seul classeur (c'est-à-dire une règle) peut être associé à plusieurs situations différentes de l'agent, comme dans la figure suivante :

$$[01\#1] \quad \uparrow \quad = \begin{cases} [01\mathbf{0}1] & \uparrow \\ [01\mathbf{1}1] & \uparrow \end{cases}$$

FIG. 1.3 – Principe de généralisation.

1.1.2 Les algorithmes génétiques

Le problème de la généralisation dans un système de classeurs consiste à trouver les parties `condition` et `action` de sorte que les symboles # soient bien placés. Pour ce faire, les systèmes de classeurs utilisent souvent des algorithmes génétiques pour faire évoluer une population de classeurs : chaque classeur est évalué tout au long de l'interaction de l'agent avec l'environnement.

Ainsi, dans la plupart des systèmes de classeurs, la création et la sélection de classeurs fiables est réalisée par des algorithmes génétiques. Un algorithme génétique est appliqué à une population de classeurs (ou une fraction de celle-ci) et se déroule en générations successives. À chaque génération, des opérateurs de croisement et de mutation (présentés ci-dessous) permettent de créer de nouveaux individus à partir des meilleurs individus de la population, et un mécanisme de sélection supprime les moins bons. Ainsi, au fur et à mesure des générations, les individus sont de plus en plus adaptés à la résolution du problème. le processus d'un algorithme génétique est généralement comme suit :

- Sélection d'un ou plusieurs individus *parents* dans la population, la plupart du temps par *roulette wheel*.
- Création de nouveaux individus *fils* par copie directe des parents (ou clonage).
- Application de croisements et/ou de mutations sur ces nouveaux individus avec des probabilité fixes.
- Mise à jour de divers paramètres (valeur sélective…).
- Insertion de ces nouveaux individus dans la population.
- Suppression d'un certain nombre d'individus considérés comme inadaptés à la résolution du problème.

Crossover

Le crossover consiste, pour deux conditions, à s'échanger une partie de leurs attributs par croisement. Il existe plusieurs techniques pour le faire, les deux principales étant,

Le *one-point crossover* :

Consiste à échanger tous les attributs à partir d'un certain index choisi aléatoirement.

$$\begin{array}{l|l} [\#00 & \#10\#0] \\ [1\#0 & 11\#\#0] \end{array} \longrightarrow \begin{array}{l} [\#00\mathbf{11\#\#0}] \\ [1\#0\mathbf{\#10\#0}] \end{array}$$

le *two-point crossover* :

Consiste à échanger tous les attributs contenus entre deux index choisis aléatoirement.

$$\begin{array}{l|l|l} [\#00 & \#10 & \#0] \\ [1\#0 & 11\# & 1\#] \end{array} \longrightarrow \begin{array}{l} [\#00\mathbf{11\#\#0}] \\ [1\#0\mathbf{\#10}1\#] \end{array}$$

Mutation

La mutation sélectionne aléatoirement un attribut de la condition et le modifie par un autre attribut distinct et aléatoire.

[#0011**0**#0]
↓
[#0011**#**#0]

L'action peut aussi être mutée en une autre différente, suivant le système de classeur.

1.1.3 L'algorithme du *Bucket Brigade*

L'algorithme du *Bucket Brigade* intègre un système économique dans la population de classeurs. Appliqué pour la première fois par Holland [Holland, 1985] aux systèmes de classeurs, le *Bucket Brigade* est un algorithme d'apprentissage permettant d'ajuster les valeurs sélectives des classeurs en rétro-propageant la récompense environnementale de classeur en classeur par un mécanisme de taxe à chaque étape.

Les classeurs représentent les agent financiers et la valeur sélective du classeur est le capital de l'agent. Le mécanisme est constitué de trois étapes :

1. La *vente aux enchères* : chaque agent concerné donne une enchère pour participer.
2. La *compétition* : une compétition est mise en place pour déterminer les agents vainqueurs. La probabilité d'être victorieux est proportionnelle à l'enchère relative (sélection par l'algorithme dit *roulette wheel*).
3. La *répartition des rétributions* : chaque agent
   - reçoit une rétribution de l'environnement s'il fait partie des vainqueurs,
   - doit payer son enchère s'il a été vainqueur,
   - reçoit une rétribution des vainqueurs s'il est à l'origine de la situation (précédent vainqueur).

1.2 ZCS

ZCS (Zeroth-level Classifier System), proposé par Wilson [Wilson, 1994], est un système de classeurs minimaliste qui ne conserve que le strict nécessaire au fonctionnement d'un système de classeurs (voir l'architecture, figure 1.4). La structure des classeurs est la plus simple possible : un classeur est caractérisé par une partie `condition`, une partie `action` et une `force` lui est associée (voir figure 1.2).

Pour ajuster les valeurs sélectives des classeurs (représentées par leur partie `force`), ZCS emploie l'algorithme du *Bucket Brigade*, celui-là même décrit plus haut. Enfin, ZCS utilise un algorithme génétique pandémique (c'est-à-dire appliqué à toute la population) similaire à celui énoncé à la section précédente et rajoute un procédé de *covering* pour créer de nouveaux classeurs lors de la formation du `match set`, lorsque celui-ci est vide par exemple (c'est-à-dire quand aucun classeur de la population ne s'apparie avec la situation courante).

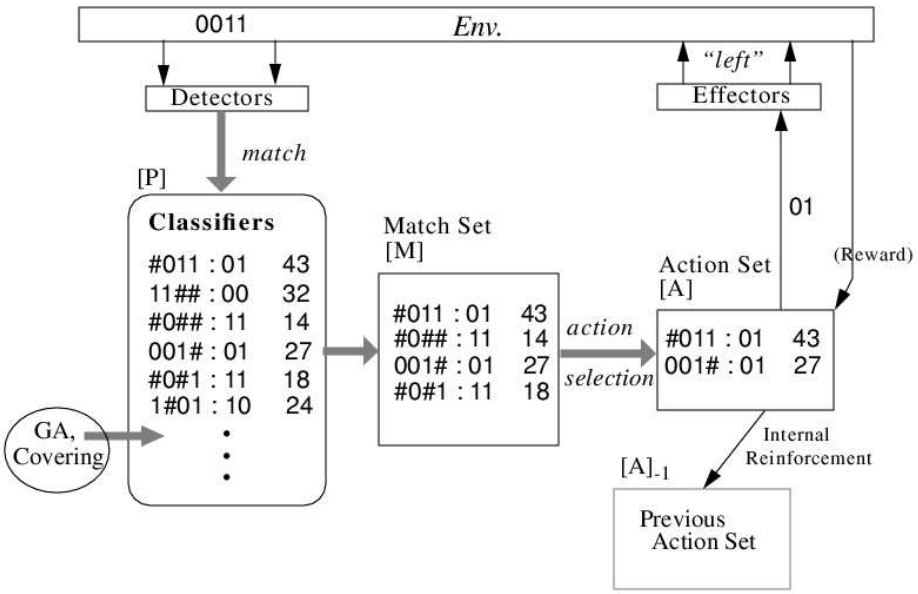

FIG. 1.4 – Architecture de ZCS.

1.3 XCS

Un an après ZCS, Wilson propose un nouveau système de classeurs qu'il appelle XCS [Wilson, 1995]. La principale différence avec ZCS (voir aussi l'architecture, figure 1.6), est de ne plus considérer la `force` d'un classeur comme valeur sélective, qui représente la récompense attendue en effectuant l'action de la partie `action` dans toutes les situations appariées à la partie `condition`.

Caractéristiques

Deux attributs sont ainsi rajoutés aux classeurs (voir figure 1.5) : une partie `prédiction` et une partie `erreur`. La partie `prédiction` joue le même rôle que la partie `force` d'un classeur ZCS mais n'est pas utilisée comme valeur sélective. La partie `erreur` est une mesure de l'erreur commise sur cette prédiction. De cette façon, la valeur sélective d'un classeur ne dépend pas de l'importance de la récompense attendue, mais de sa capacité de la prévoir avec précision.

Condition	Action	Prédiction	Erreur	Valeur sélective	Numérosité
[01#0#101]	↑	48	0.12	69	2

FIG. 1.5 – Constitution d'un classeur XCS.

D'autre part, l'algorithme génétique utilisé n'est appliqué qu'à un sous-ensemble de la population (plus précisément l'`action set`) et non à la totalité de la population, comme dans ZCS. Ceci est sensé être plus effectif au niveau du crossover car les classeurs de ce sous-ensemble sont apparentés.

Enfin, un dernier attribut appelé `numérosité` est ajouté aux classeurs. Il a pour but de donner une certaine force aux classeurs qui sont souvent *découverts* par le système. En effet, un système de classeurs ne peut pas contenir deux classeurs identiques (en se basant sur la condition et l'action). Or, lors de l'exécution de l'algorithme génétique par exemple, il se peut que l'un des classeurs fils nouvellement crée existe déjà dans la population. Donc, au lieu d'ignorer ce "nouveau" classeur (à l'instar de ZCS), on incrémente d'une unité la numérosité du classeur existant. Ainsi, un classeur ne sera effectivement supprimé que lorsque sa numérosité atteindra zéro. Un tel classeur est appelé *macro-classeur* et contient autant de *micro-classeurs* que la valeur de sa numérosité.

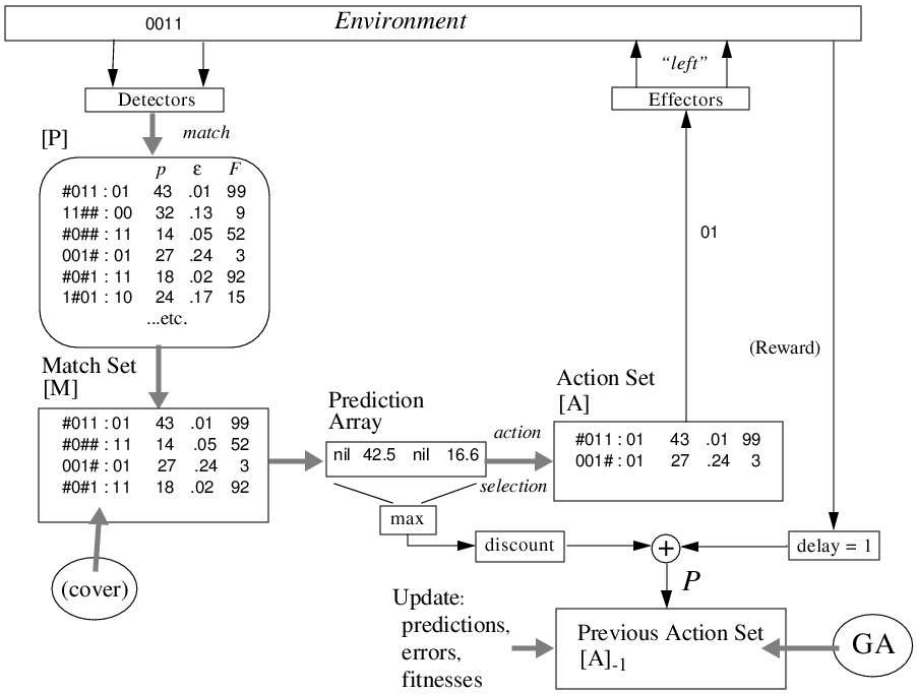

FIG. 1.6 – Architecture de XCS.

Performances

Kovacs [Kovacs, 1997] a montré que XCS découvre des classeurs fiables et aussi généraux que possible. Un classeur est dit fiable si, dans chaque situation correspondant à la condition, choisir l'action proposée mène toujours effectivement au même cumul de récompense. Il est dit aussi général que possible dès lors qu'il ne peut pas contenir plus de symboles # tout en restant fiable. Un classeur compatible avec plusieurs paires (s, a) à différents niveaux de récompense attendue n'est pas fiable parce qu'il est trop général.

De plus, XCS tend vers des ensembles de classeurs complets, où à chacune des paires (s, a) possibles correspond au moins un classeur puisque la valeur sélective se fonde sur l'exactitude de la prévision plutôt que sur l'optimalité de l'action.

Ainsi, XCS construit une carte complète des récompenses attendues et utilise la généralisation pour réduire le nombre de classeurs. Il décide de la fiabilité d'un classeur de telle manière que la généralisation ne peut se produire que parmi les paires (s, a) à un même niveau de récompense attendue.

1.4 ACS

ACS est une combinaison de LCS avec la théorie d'apprentissage psychologique du contrôle comportemental anticipatoire [Hoffmann, 1993]. Pour réaliser cette théorie dans ACS, on ajoute une partie `effet` à la structure d'un classeur. Cette partie `effet` est la clé du processus d'apprentissage anticipatoire (ALP) qui permet à ACS d'apprendre la représentation interne complète d'un environnement [Seward, 1949], et autorise ainsi plusieurs processus cognitifs. Il a été montré qu'ACS permet également un apprentissage latent [Stolzmann, 1999] signifiant la capacité d'apprendre un modèle interne de l'environnement sans rencontrer de récompense.

1.4.1 Le framework ACS

La représentation de la connaissance

La connaissance d'ACS est représentée par une population de classeurs similaire aux autres LCS's. Cependant, un classeur d'ACS possède une partie `effet` supplémentaire et une mesure de la qualité. Ainsi, chaque classeur est constitué des parties suivantes :

- La partie `condition` (C) spécifie les situations perçues.
- La partie `action` (A) spécifie l'action que le classeur veut exécuter.
- La partie `effet` (E) anticipe les changements causés par l'action A sur la condition C.
- La `marque` (M) spécifie tous les attributs des situations σ où le classeur n'a pas anticipé correctement.
- La `qualité` q mesure la précision des anticipations.
- La `mesure de récompense` r prédit la future récompense donnée par l'environnement après l'exécution de l'action A.

Condition	Action	Effet	Marque	Qualité	Récompense	Numérosité
[01#0#101]	↑	[01#0##00]	##{0,1}##{0}#	69	36	2

FIG. 1.7 – Constitution d'un classeur ACS.

Un symbole # dans une condition a la même signification que celle décrite plus haut. En revanche, dans la partie `effet`, le symbole #, appelé *pass-through*, prédit que cet attribut restera inchangé après l'exécution de l'action. La marque a la structure M $\in (\{s_{11}, \ldots, s_{1o_1}\}, \ldots, \{s_{L1}, \ldots, s_{Lo_L}\})$ où $o_1, \ldots, o_L$ sont le nombre des différentes valeurs de chaque attribut et s_{ij} les différentes valeurs. Ainsi, la marque informe si le classeur n'a pas fonctionné correctement. Si c'est le cas elle donne de plus amples informations sur les propriétés des situations au moment où il n'a pas fonctionné correctement. En utilisant ces propriétés, ACS est capable de créer une version "corrigée" de ce classeur (processus de différenciation).

Une action comportementale

Une action comportementale est similaire aux autres LCS's mais diffère dans les parties anticipatoires. Une fois l'action exécutée, la récompense associée $\rho(t)$ ainsi que la nouvelle situation dans l'environnement $\sigma(t+1)$ sont perçues. En considérant ρ et $\sigma(t+1)$ l'*action set* est modifié par l'ALP (qui sera introduit dans la prochaine sous-section). La figure 1.8 donne un aperçu du processus.

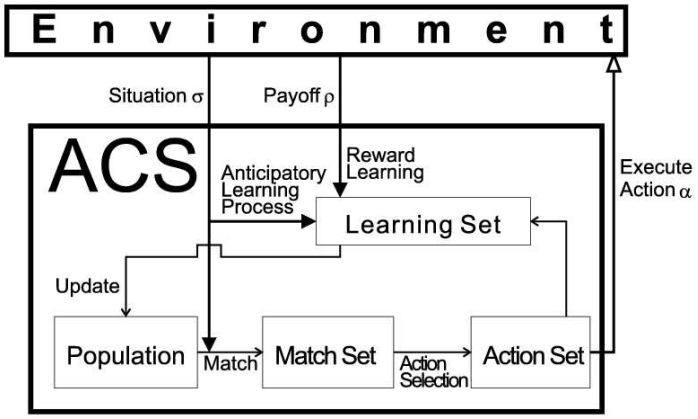

FIG. 1.8 – ACS utilise ses propres anticipations et perceptions successives pour former de nouveaux classeurs à partir de ceux qui s'apparient avec la perception courante et possèdent la même action.

1.4.2 Les processus d'apprentissage dans ACS

La théorie du contrôle comportemental anticipatoire

La figure 1.9 montre l'idée basique de cette théorie. Premièrement, un comportement R est toujours accompagné par les conséquences anticipées E_{ant} et la situation donnée actuelle abstraite à une condition S_{start}. Deuxièmement, une comparaison continue prend place entre les anticipations et les perceptions successives. Si la comparaison est valide (respectivement invalide) cela mène à une augmentation (diminution) de la relation entre l'anticipation et la relation stimulus-réponse correspondante. Finalement, les anticipations imprécises mènent à davantage de différenciation des conditions.

Le processus d'apprentissage anticipatoire (ALP)

Afin de réaliser cette théorie dans un système artificiel, il est nécessaire que les anticipations soient représentées d'une certaine façon. L'approche la plus simple est de construire directement des règles $S_{start} - R - E_{ant}$.

L'ALP distingue les cas suivants :

- Premièrement, un classeur anticipe incorrectement la situation $\sigma(t+1)$ résultante. Comme dans la théorie, cela résulte en une diminution entre la relation de l'anticipation et la partie stimulus-réponse, représentée par

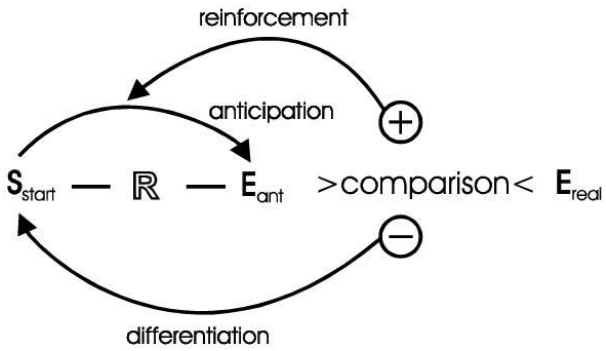

FIG. 1.9 – Le mécanisme d'apprentissage du contrôle comportemental anticipatoire établit que cet apprentissage est basé sur la comparaison des effets anticipés E_{ant} et des effets réels E_{real}.

une diminution de la qualité du classeur. En outre, le classeur est *marqué* par $\sigma(t)$, ce qui va aider l'ALP à différencier la règle de ces situations infructueuses.

- Deuxièmement, un classeur anticipe correctement $\sigma(t+1)$. De la même façon qu'au premier cas la qualité du classeur est maintenant augmentée, ce qui résulte en une augmentation entre la relation $S - R$ et E. De plus, si le classeur a une marque, alors l'ALP génère un nouveau classeur avec une partie `condition` plus spécifique considérant $\sigma(t)$ et les attributs de la marque. Ceci représente la différenciation des conditions établie dans la théorie.

1.5 Le meta-apprentissage dans les systèmes de classeurs

Une des voies de recherche pour améliorer les performances des systèmes de classeurs dans le multi-agents est d'utiliser une approche de méta-apprentissage introduite par Jürgen Schmidhuber [Schmidhuber, 1996][Zhao and Schmidhuber, 1996]. Cette approche est basée sur des politiques auto-modificatrices (self-modifying policies). De tels systèmes ont alors la possibilité de modifier leur propre façon de se modifier. Schmidhuber propose un algorithme de modification des politiques d'apprentissage nommé "Success-story algorithm" (SSA), qui a l'avantage d'être indépendant de l'environnement, c'est-à-dire plus adapté à une utilisation dans un contexte multi-agents.

1.5.1 Définition du meta-apprentissage

Considérons un système apprenant exécutant une séquence d'actions durant sa vie dans un environnement incertain. Les actions peuvent nécessiter différents temps d'exécution. De manière occasionnelle, l'environnement procure à l'agent une valeur réelle de renforcement. La somme de tous les renfor-

cements obtenue entre la naissance du système (au temps $t = 0$) et le temps t est noté $R(t)$. Durant toute la durée de sa vie, le but du système est de maximiser $R(T)$, le renforcement cumulé du système jusqu'au temps de sa mort (T non connu à l'origine).

La politique courante d'un agent est modifiée par les PMPs (Policy Modification Processes), appelés également processus d'apprentissage. Plusieurs PMPs peuvent prendre des durées différentes de temps. La énième PMP de la vie d'un système est dénoté PMP$_i$ et démarre au temps $t_i^1 > 0$ et fini au temps $t_i^2 > t_i^1$ et calcule une modification de politique nommée M_i.

Programme

Pour ces principes de bases, Schmidhuber utilise des animats dont le fonctionnement est une sorte de programme de type assembleur. Ce programme possède n *cellules* dont les adresses varient de 0 à $n - 1$. Ces cellules sont divisées en deux ensembles :

- Une section *programme* dont les cellules sont interprétées comme du code exécutable.
- Une section de *travail* dont les cellules sont lues et écrites par les instructions exécutées dans la section *programme*.

Un *pointeur d'instruction* (*IP*) donne à tout moment la position de la cellule programme courante. Enfin, pour chaque cellule programme i il existe une distribution de probabilité P_i sur les instructions. La plus forte probabilité indique quelle instruction devra être exécutée lorsque la cellule sera pointée par l'*IP*. Chaque cellule programme contient donc une instruction (auto-modificatrice ou non) dont le ou les arguments sont situés aux adresses contenues dans les cellules programme suivantes. Précisons que ces adresses réfèrent à des cellules de la section de travail.

Dans cette implémentation, les actions auto-modificatrices sont au nombre de quatre :

***GetProb()* :** Peut être utilisé pour écrire dans des cellules de travail. Potentiellement utile à des buts d'introspection.

***DecProb()* :** Diminue d'un certain facteur la valeur d'une cellule.

***IncProb()* :** Augmente d'un certain facteur la valeur d'une cellule.

***EndSelfMod()* :** Aide à grouper les actions auto-modificatrices et d'autres actions en PMPs ou SMSs (self-delimiting self-modification sequences) de la figure 1.10.

Critère d'accélération du renforcement

Le système applique continuellement des PMPs à sa politique et mesure les changements résultants dans le taux de renforcement.

En effet, au fur et à mesure de ces modifications le renforcement de l'agent doit subir une accélération, ou tout du moins, ne pas diminuer. Supposons qu'une PMP$_i$ commence au temps t_i^1 et finisse au temps $t_i^2 > t_i^1$. Alors pour $t \geq t_i^2$ et $t \leq T$, il est possible de calculer cette accélération par un rapport renforcement/temps :

$$Q(i,t) = \frac{R(t) - R(t_i^1)}{t - t_i^1}$$

Au temps t, le critère est satisfait si pour chaque PMP$_i$ qui a réalisé une modification M_i de la politique encore valide,

- $Q(i,t) > \frac{R(t)}{t}$, où M_i est l'unique PMP valide existant sur la pile.
- $Q(i,t) > Q(k,t)$, pour tout $k < i$ où PMP$_k$ a réalisé une M_k valide.

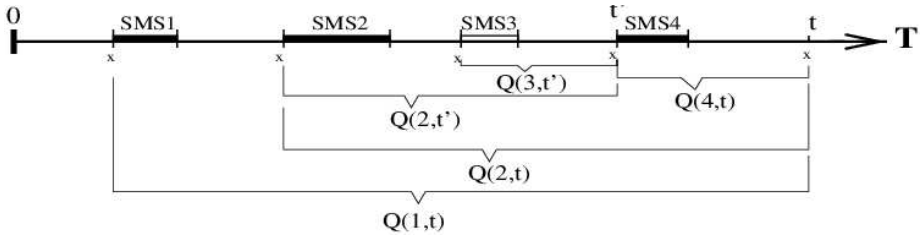

FIG. 1.10 – Satisfaction du critère d'accélération du renforcement.

À la fin d'une PMP, les informations nécessaires pour restaurer l'ancienne politique sont sauvegardées sur une pile. Ultérieurement, le système teste si la PMP courante satisfait le critère d'accélération du renforcement. Si c'est le cas, la PMP est conservée jusqu'à ce que la prochaine ait lieu. Sinon, on restaure les politiques conservées en les dépilant jusqu'à ce que le critère ci-dessus soit à nouveau satisfait.

1.5.2 Application aux systèmes de classeurs

Une première application de ces principes aux systèmes de classeurs a été réalisée l'année dernière par Erwan Livolant lors de son stage de DEA [Livolant, 2003, pages 17–20] (voir aussi [Stinckwich et al., 2004]).

Toutefois, un problème s'est posé. Il semble que le temps nécessaire pour observer une amélioration des performances soit excessivement long (Schmidhuber lui-même présente des expériences dans lesquelles des temps de 10^9 time-steps sont requis).

En 1996, Wilson a émis une critique de sur ces principes d'accélération du renforcement [Wilson, 1996, page 6]. Pour lui, les performances d'un tel système « *would improve impractically slow* » et ressemblaient plus à « *des mutations aléatoires avec préservation des meilleures* ». Néanmoins, Wilson pensait que cette théorie pouvait être un guide important et mener à d'autres approches, en particulier en ce qui concerne le problème exploration/exploitation.

Chapitre 2

LCSTalk, un framework modulaire pour les systèmes de classeurs

2.1 Présentation de LCSTALK

LCSTALK[1] est une plateforme que j'ai développé en SQUEAK[2]/SMALLTALK[3] servant à implémenter des systèmes d'apprentissage par classeurs. SQUEAK a été choisi comme environnement de développement car il intègre, en plus d'utiliser un langage objet puissant, des outils d'inspection et de débugage très performants.

LCSTalk a été nominé aux `Innovation Technology Awards` de l'ESUG[4] 2004 (European Smalltalk Users Group) qui s'est déroulé début septembre et a été classé $5^{ième}$.

Enfin, LCSTalk est aussi utilisé dans d'autres travaux de l'équipe comme celui d'Erwan Livolant (*cf.* [Stinckwich et al., 2004]) et celui d'Olivier Sagit (*cf.* [Sagit, 2004]).

2.1.1 Fonctionnalités

Cette plateforme a été conçue dans le but de mesurer les performances de systèmes d'apprentissage par classeurs sur un éventail assez large et divers d'environnements. elle a donc été conçue, dans la mesure du possible, de façon assez abstraite et modulaire. Ceci afin de pouvoir rajouter par la suite de nouveaux systèmes de classeurs. Elle s'adresse à des développeurs et des utilisateurs finaux.

En plus de ces systèmes de classeurs LCSTALK a été conçu pour fonctionner en mono et multi-agents.

Environ une vingtaine d'environnements de type *Maze* et *Woods* sont pour l'instant disponibles comme environnements de tests. Mais de même que pour

[1] Les sources sont disponibles sur http ://www.squeaksource.com/LCSTalk/

[2] http ://www.squeak.org

[3] http ://www.smalltalk.org

[4] http ://www.esug.org/conferences/twelfthsmalltalkjointevent2004/innovationtechnologyawards/winnersandnominations

les systèmes de classeurs de nouveaux environnements sont facilement ajoutables.

Une fois le système de classeurs et un environnement choisis (ZCS et Woods1 par exemple) il suffit alors de cliquer sur le bouton Run problems pour que la simulation démarre. Celle-ci s'achève lorsque le nombre de problèmes prévus (1000 pas défaut) est réalisé ou si l'on clique sur l'un des boutons *pause* ou *Stop*.

2.1.2 Description de l'interface

L'interface est une fenêtre composée de trois éléments distincts regroupant l'intégralité des informations utiles à l'expérimentateur (*cf.* capture d'écran figure 2.1).

Environnement

La première partie, en haut à gauche, affiche l'environnement sous forme de cases dans lequel se déplace l'agent, lui-même représenté par un cercle de couleur aléatoire. Une case vert clair correspond à une case vide dans laquelle l'agent peut se déplacer, une case jaune correspond à un état but (nourriture par exemple) et les espaces vert foncé sont des murs (ou n'importe quoi d'infranchissable).

À cet emplacement sont aussi affichés la progression de la simulation (du moins sur les cent derniers tours) ainsi que le bouton permettant le déclenchement et l'arrêt de celle-ci.

Carte du système de classeurs

La seconde partie, en haut à droite, affiche la population de classeurs de l'agent (ou du premier agent, dans le cas d'un système multi-agents) avec différents attributs (dont le nombre et le type varie suivant le système de classeurs utilisé).

La partie gauche est commune à chaque système. Elle représente la partie `condition` du classeur, c'est-à-dire la perception de l'agent situé dans l'environnement. Les autres attributs sont les valeurs de certains paramètres donnés par le type de système de classeur comme la `valeur sélective` ou la `numérosité`, etc.

Statistiques des données du système

Quant à la dernière partie, en bas, c'est un graphique affichant différents paramètres de la simulation, comme le nombre de déplacements par problème ou la taille de la population de classeurs (dans le cas où celle-ci n'est pas statique). Là aussi, ces paramètres varient en fonction du type de LCS utilisé. On peut également générer une version `Gnuplot` pour l'utiliser dans un article.

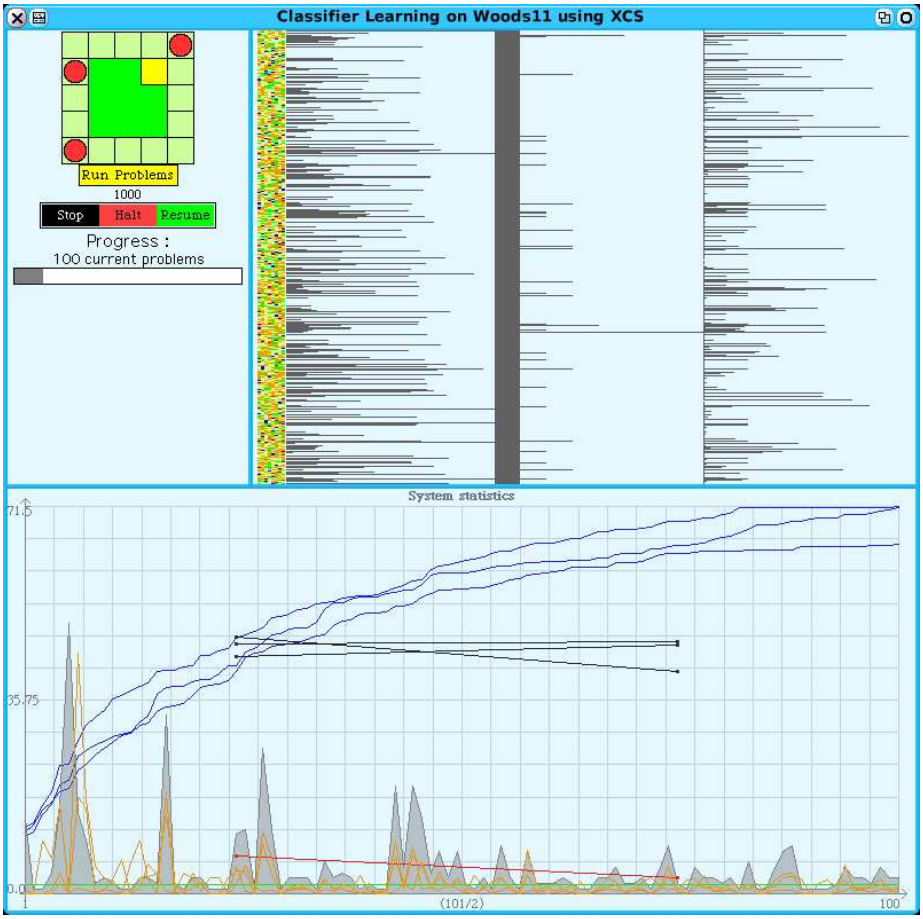

FIG. 2.1 – Exemple d'interface pendant une exécution

2.2 Architecture de LCSTALK

2.2.1 Organisation des classes

Les classes (voir le diagramme de la figure 2.2, page 21) sont divisées en six paquetages :

`LCSTalk-Base` contient toutes les classes servant de *moteur* à la plateforme. On y trouve la classe `LCSTalk` qui crée l'interface et gère le déroulement de la simulation et la classe `Agent`.

`LCSTalk-Classifiers` comprend les différents types de classeurs, par exemple `ZCSClassifier` ou `XCSClassifier`. Chacun de ces classeurs dérivent d'un classeur générique commun `Classifier` qui comprend les attributs et méthodes que tous doivent comporter. C'est le cas pour le triplet $< condition - action - fitness >$ qui est la base de tout classeur.

`LCSTalk-ClassifierSets` contient les classes qui gèrent une liste de classeurs comme `XCSClassifierSet`. Ces classes héritent de `ClassifierSet` et possèdent les algorithmes spécifiques à chaque LCS

comme la mise à jour des valeurs sélectives ou les algorithmes génétiques.

LCSTalk-Systems comprend les différents systèmes de classeurs. La classe `ClassifierSystem` est *abstraite* et gère le comportement de base des systèmes de classeurs. Chaque système doit en hériter (soit directement, soit par l'une de ses sous-classes). Les classes importantes (pour l'instant) sont `ZCS`, `XCS` et `ACS`. Ce paquetage contient aussi une classe nommée `RCS` qui simule un système dont tous les déplacements sont aléatoires, ce qui est pratique pour comparer les performances des autres LCS.

LCSTalk-Environnements quant à lui regroupe les environnements dans lesquels se déplacera l'agent. Ceux-ci sont divisés en deux groupes aux caractéristiques distinctes : les *Mazes* et les *Woods*. Les *Mazes* sont des environnements fermés, tandis que les *Woods* ont les bords joints et sans murs, simulant un environnement infini.

LCSTalk-Support regroupe les classes restantes comme `Classifier-SystemMorph` qui crée la carte graphique de la liste des classeurs (en haut à droite) ou `Statistics` qui crée le graphique des statistiques (en bas).

2.2.2 Description du framework

Les classes de base

LCSTalk :

C'est la classe principale. Elle s'occupe de créer l'interface ainsi que l'environnement, le ou les agents suivant les paramètres donnés par l'utilisateur. C'est aussi elle qui contrôle les le déroulement d'une simulation et en synchronise les différentes parties. En effet, pour le mono-agent il n'y a pas de problème, mais en multi-agents il est nécessaire de diviser l'exécution d'un problème en plusieurs phases dont certaines doivent être effectuées simultanément par tous les agents. Ce partitionnement est aussi nécessaire pour trouver un schéma d'exécution commun à tous les LCS, ce qui n'est pas évident du fait de leur diversité. Ainsi l'exécution d'un problème s'effectue en deux phases distinctes, elles-mêmes divisées en deux parties chacune :

- Une phase d'exploration (qui n'est pas exécutée par tous les LCS, comme ZCS par exemple) :
  - Une sous-phase de déplacement.
  - Une sous-phase de renforcement.
- Une phase d'exploitation :
  - Une sous-phase de déplacement.
  - Une sous-phase de renforcement.

Ce sont ces sous-phases, et en particulier celles de déplacement des agents, qui doivent absolument être exécutés simultanément, car leurs mouvements doivent être indépendants les uns des autres.

Agent :

Cette classe gère tout ce qui est relatif à un agent, que ce soit sa position, ses déplacements, son propre système de classeur et peut accéder à l'environnement pour percevoir sa situation par exemple.

Les listes de classeurs

Les listes de classeurs, comme `XCSClassifierSet` contiennent tous les classeurs d'un agent. Leur taille peut être fixe (ZCS) ou non. Elles gèrent les algorithmes qui agissent sur toute la population de classeurs comme les algorithmes génétiques ou les créations de *match sets* et d'*action sets*.

Les classeurs

Comme il a été dit plus haut chaque classeur possède au moins les trois attributs suivants : condition, action et fitness (ou valeur sélective). Ceci est réalisé dans la classe abstraite `Classifier`.

De plus cette classe impose à chacune de ses sous-classes de définir une liste des attributs (par exemple *fitness*) à afficher dans la carte des classeurs, nommée *observedValues*. Par défaut cette liste est vide.

Ainsi, `ZCSClassifier`, qui est le plus simple des classeurs, ne redéfinit que la méthode d'initialisation pour modifier cette liste. D'autres, comme `XCSClassifier`, rajoutent un nombre plus ou moins important d'attributs servant à leur fonctionnement.

Les systèmes de classeurs

Chaque système de classeurs possède un *squelette* commun représenté par la classe `ClassifierSystem`. En effet, tous doivent contenir quatre phases différentes (même si certaines ne font rien) énoncées plus haut. Celles-ci sont nécessaires pour synchroniser les agents lors d'un problème multi-agents. Enfin, chaque système de classeurs décide quels paramètres ou fonctions seront affichés dans le graphique de statistiques, permettant ainsi d'en suivre l'évolution.

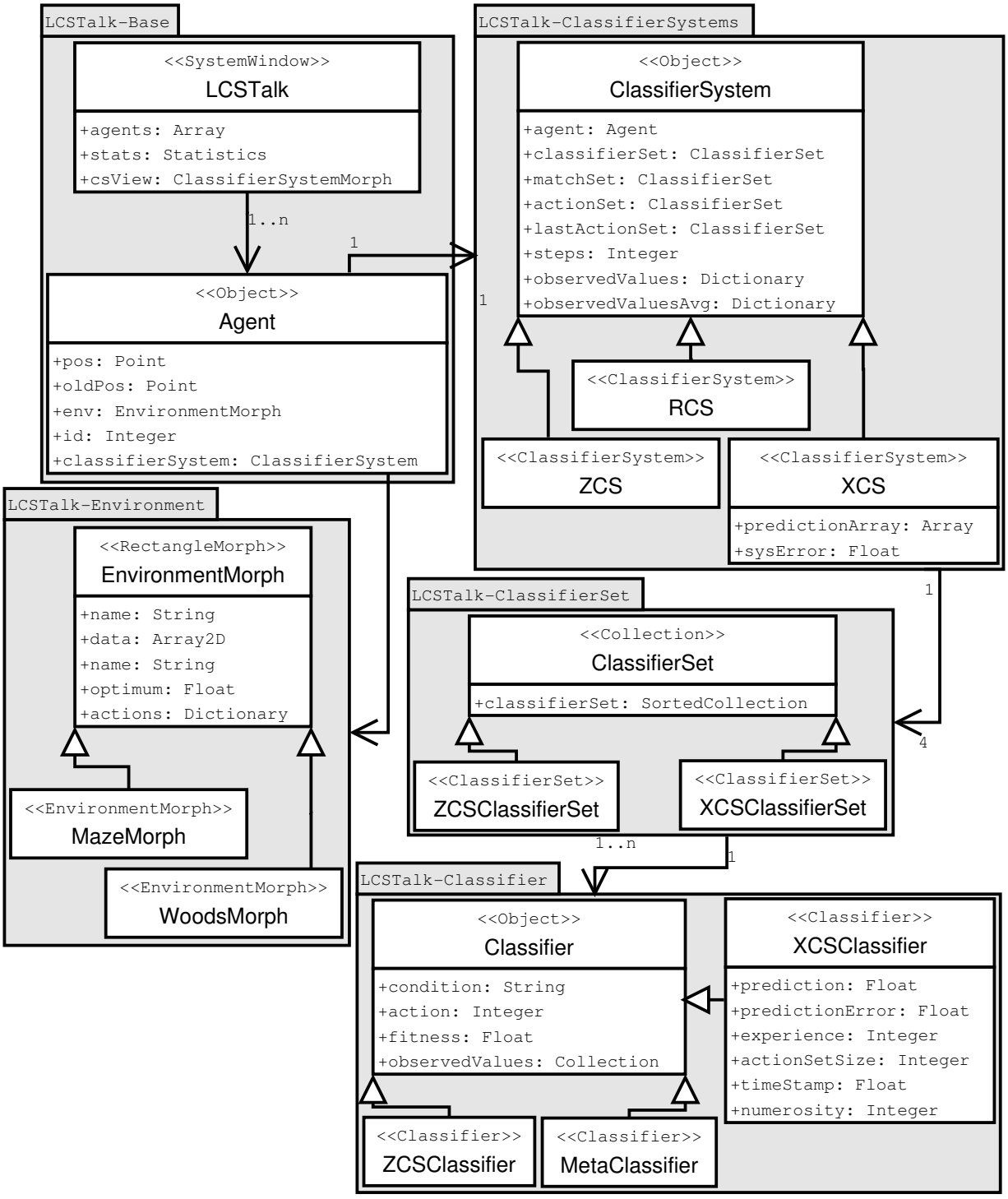

FIG. 2.2 – Diagramme de classes de LCSTalk.

Chapitre 3

Le framework CLRI, analyse du comportement des agents

Partant du fait que le nombre de systèmes multi-agents (SMA) est en constante augmentation, il est devenu particulièrement important de pouvoir analyser ces systèmes. Jusqu'à présent, les principales recherches dans ce domaine consistent en des expérimentations où l'on étudie les SMA's en faisant varier différents apprentissages ou paramètres du système. Les résultats sont alors rassemblés et analysés de façon à déterminer dans quelles mesures les changements de comportements individuels des agents affectent le comportement global du système. C'est pourquoi, José M. Vidal et Edmund H. Durfee [Vidal and Durfee, 2003] ont décrit, à partir de ces observations expérimentales, un framework qui peut être utilisé pour modéliser et prédire le comportement d'un système multi-agents. Ce framework donne une équation de différence permettant de calculer la progression de l'erreur que fait un agent dans sa fonction de décision.

Nous nous proposons d'utiliser ce framework et de l'appliquer aux systèmes de classeurs en général, et plus particulièrement à LCSTalk, afin d'étudier le comportement des agents dans ces systèmes.

3.1 Un framework pour modéliser les SMA's

Un SMA est constitué d'un ensemble fini d'agents N, d'actions A et d'états W. Chaque agent possède un ensemble de capteurs pour percevoir l'environnement dans lequel il se trouve et qui lui servent à déterminer son état courant w. L'ensemble de tous ces états est W. A_i, où $|A_i| \geq 2$, est l'ensemble des actions que peut effectuer l'agent $i \in N$. Le temps t est supposé discret où t est un entier supérieur ou égal à 0. On suppose également qu'il n'existe qu'un seul état w pour chaque instant t. Enfin, on suppose que l'environnement est déterministe, c'est-à-dire que les actions combinées des agents auront toujours l'effet attendu.

3.1.1 Description du comportement d'un agent

Les agents considérés ici sont engagés dans une boucle action/apprentissage. Cette boucle est la suivante : au temps t, les agents perçoivent un état $w^t \in W$. Ils effectuent ensuite les actions dictées par leur fonction de décision δ_i^t ; toutes ces actions sont supposées être effectuées en parallèle. Enfin, ils reçoivent chacun une récompense que leurs algorithmes d'apprentissage respectifs vont utiliser pour modifier la fonction δ_i^t, en espérant ainsi se rapprocher de la fonction cible Δ_i^t. On suppose alors, qu'à chaque instant t, l'on peut décrire le comportement d'un agent par un doublet état–action.

Formellement, on désigne par **fonction de décision** (aussi appelée *politique*) le comportement d'un agent i, donnée par $\delta_i^t : W \rightarrow A_i$. Cette fonction relie chaque état $w \in W$ avec l'action $a_i \in A_i$ que l'agent i effectuera dans cet état, au temps t.

On appelle **fonction cible** (donnée par $\Delta_i^t : W \rightarrow A_i$), la fonction qui, pour chaque état w, donne l'action a_i que *devrait* effectuer l'agent i pour atteindre son but de façon optimale. Une telle fonction représente donc le comportement *parfait* pour un agent et est utilisée pour mesurer ses performances. L'agent n'a pas connaissance de cette fonction mais doit s'efforcer de faire tendre sa fonction de décision le plus près possible de sa fonction cible. Par ailleurs, comme le choix d'action d'un agent i dépend souvent des actions des autres agents, la fonction cible de i doit prendre en compte ces autres actions.

La mesure de l'exactitude du comportement d'un agent est donnée par une mesure d'erreur. L'erreur de la fonction de décision de l'agent i est définie par

$$\begin{aligned} e(\delta_i^t) &= \sum_{w \in W} \mathcal{D}(w) \mathbf{Pr}[\delta_i^t(w) \neq \Delta_i^t(w)] \\ &= \mathbf{Pr}_{w \in \mathcal{D}}[\delta_i^t(w) \neq \Delta_i^t(w)]. \end{aligned} \tag{3.1}$$

$e(\delta_i^t)$ nous donne la probabilité que l'agent i effectue une mauvaise action. Une erreur de 0 signifie que l'agent effectue toutes les actions dictées par sa fonction cible, tandis qu'une erreur de 1 signifie que l'agent effectue toujours une mauvaise action. Chaque action effectuée est donc correcte ou incorrecte, soit elle correspond à la fonction cible, soit elle ne correspond pas. Il n'y a pas de dégrés dans la mesure de l'inexactitude.

3.1.2 Un modèle d'algorithmes d'apprentissage

Sachant que l'algorithme d'apprentissage d'un agent doit modifier δ_i^t en δ_i^{t+1} afin de se rapprocher de Δ_i^t, les auteurs ont trouvé un ensemble de paramètres capables de modéliser les effets d'un grand nombre d'algorithmes d'apprentissage. Ces paramètres sont : le taux de changement (**C**), le taux d'apprentissage (**L**), le taux de rétention (**R**) et l'impact (**I**). Ces paramètres, avec les équations fournies, forment le framework **CLRI**.

Définition des paramètres

Le **taux de changement** d'un agent est défini comme la probabilité que l'agent modifie au moins l'une de ses décisions[1] incorrectes. Formellement, le

[1] j'appelle ici *décision* l'action qui serait choisie par la fonction de décision pour un état w au temps t, pouvant être représentée par le doublet $w^t \rightarrow a_i$.

N	L'ensemble de tous les agents, où $i \in N$ est un agent particulier.
W	L'ensemble des états possibles de l'environnement, où $w \in W$ est un état particulier.
A_i	L'ensemble de toutes les actions que l'agent i peut effectuer.
δ_i^t :	$W \rightarrow A_i$ La fonction de **décision** de l'agent i au temps t. Elle donne quelle action l'agent i prendra à chaque état.
Δ_i^t :	$W \rightarrow A_i$ La fonction de **cible** de l'agent i au temps t. Elle donne quelle action l'agent i devrait prendre. Elle prend en compte les actions que les autres agents prendront.
$e(\delta_i^t)$	$= \mathbf{Pr}[\delta_i^t(w) \neq \Delta_i^t(w) \mid w \in \mathcal{D}]$ l'**erreur** de l'agent i au temps t. C'est la probabilité que l'agent i effectue une action incorrecte, étant donné que les états w proviennent d'une distribution de probabilité $\mathcal{D}$.

FIG. 3.1 – Résumé des notations utilisées pour décrire un SMA dans le framework CLRI.

taux de changement c_i pour l'agent i est défini par

$$\forall_w \mathbf{Pr}[\delta_i^{t+1}(w) \neq \delta_i^t(w) | \delta_i^t(w) \neq \Delta_i^t(w)] = c_i.$$

Le taux de changement indique la probabilité qu'un agent modifie une décision incorrecte en autre chose. Cet *autre chose* peut aussi bien être une action correcte ou incorrecte.

La probabilité qu'un agent modifie une décision incorrecte en une action correcte est appelée **taux d'apprentissage**. Il est noté l_i et est défini par

$$\forall_w \mathbf{Pr}[\delta_i^{t+1}(w) = \Delta_i^t(w) | \delta_i^t(w) \neq \Delta_i^t(w)] = l_i.$$

Ces deux taux nécessitent que deux contraintes soient satisfaites. Comme la modification d'une décision implique qu'il y a eu un changement, on doit toujours avoir $l_i \leq c_i$. De plus, si $|A_i| = 2$ alors $l_i = c_i$ puisqu'il n'y a que deux actions possibles, ainsi, celle qui n'est pas fausse doit être vraie.

On appelle **taux de rétention** la probabilité qu'une décision correcte le reste à l'itération suivante. Le taux de rétention, noté r_i est donné par

$$\forall_w \mathbf{Pr}[\delta_i^{t+1}(w) = \Delta_i^t(w) | \delta_i^t(w) = \Delta_i^t(w)] = r_i.$$

Dernier paramètre, on définit l'**impact** I_{ji} que la modification de la fonction de décision de l'agent j a sur la fonction cible de l'agent i :

$$\forall_{w \in W} I_{ji} = \mathbf{Pr}[\Delta_i^{t+1}(w) \neq \Delta_i^t(w) | \delta_j^{t+1}(w) \neq \delta_j^t(w)].$$

Le principe de ce modèle est que l'on peut décrire une grande variété d'algorithmes d'apprentissage en utilisant des valeurs appropriées pour c_i, l_i, et r_i.

3.1.3 Calcul de l'erreur des agents

Un cinquième paramètre est utilisé pour calculer cette erreur. Il est appelé **volatilité** et désigne la probabilité que la fonction cible va changer entre l'instant t et l'instant $t+1$. Formellement, la volatilité v_i est donnée par

$$\forall_w \, \mathbf{Pr}[\Delta_i^{t+1}(w) \neq \Delta_i^t(w)] = v_i. \tag{3.2}$$

Avec le paramètre d'impact, on peut calculer la volatilité attendue :

$$E[v_i^t] = 1 - \prod_{j \in N-i} 1 - I_{ji}\Big(c_j e(\delta_j^t) + (1 - r_j) \cdot (1 - e(\delta_j^t))\Big). \tag{3.3}$$

Si tout les agents du SMA sont identiques (*i.e.* ils ont les mêmes taux c, l, r et I, et ont la même erreur initiale) alors l'équation ci-dessus peut-être simplifiée en :

$$E[v_i^t] = 1 - \Big(1 - I_{ji}\Big(c_j e(\delta_j^t) + (1 - r_j) \cdot (1 - e(\delta_j^t))\Big)\Big)^{|N|-1}. \tag{3.4}$$

Étant donnés tous ces paramètres et la valeur de l'erreur au temps t, on introduit une équation de différence permettant de calculer l'erreur attendue d'un agent au temps $t+1$, telle que définie dans l'équation 3.1 :

$$\begin{aligned} E[e(\delta_i^{t+1})] = {} & 1 - r_i + v_i \left(\frac{|A_i| r_i - 1}{|A_i| - 1} \right) \\ & + e(\delta_i^t) \left(r_i - l_i + v_i \left(\frac{|A_i|(l_i - r_i) + l_i - c_i}{|A_i| - 1} \right) \right). \end{aligned} \tag{3.5}$$

Grâce à cette équation, il est possible de déterminer l'erreur attendue d'un agent à chaque instant en prenant $E[e(\delta_i^{t+1})]$ comme $e(\delta_i^t)$ à la prochaine itération.

3.2 Application aux systèmes de classeurs

Le problème que nous nous posons est d'appliquer aux systèmes de classeurs les concepts de base énoncés au début de ce chapitre et de les implémenter dans la plateforme LCSTALK.

3.2.1 Transposition des concepts de base

Les principes que nous avons à transposer sont ceux résumés dans le tableau de la figure 3.1.

Pour N et A_i les définitions restent les mêmes. Le nombre d'actions et les actions elles-mêmes seront les mêmes pour tous les agents, et on aura généralement $|A_i| = 4$ ou $|A_i| = 8$, ce qui correspond aux directions de déplacement.

L'ensemble W des états possibles de l'environnement devient l'ensemble des situations possibles de l'environnement. La fonction de décision, qui constitue la politique d'un agent, correspond alors tout à fait au système de classeurs que possède un agent.

En ce qui concerne la fonction cible, elle reste à implémenter. Il s'agit de calculer, pour chaque état $w \in W$, quelles actions sont optimales pour se rapprocher du but, et de conserver dans une liste les doublets $w \rightarrow a$ trouvés.

Enfin, l'erreur expérimentale $e(\delta_i^t)$ est calculée selon la formule 3.1 en comparant les actions effectuées par l'agent avec celles qu'il *aurait dû* faire.

3.2.2 Valeur des paramètres

Détermination de I/v_i

La définition du paramètre d'impact servant à calculer la volatilité (*cf.* équation 3.4) est pratiquement impossible à appliquer aux systèmes de classeurs. En revanche, le paramètre v_i lui-même l'est assez facilement. en effet, la volatilité est la probabilité que la fonction cible va changer au court du temps. En ce qui nous concerne, il y a deux cas :

- Si le système est mono-agent, alors le but de l'agent est de trouver la case de récompense. Or l'environnement est statique, ce qui veut dire que la récompense est fixe. Dans ce cas, $\Delta_i^t = \Delta_i^{t+1}$ et $v_i = 0$. L'équation 3.5 devient alors :

$$E[e(\delta_i^{t+1})] = 1 - r_i + e(\delta_i^t)\,(r_i - l_i)\,. \tag{3.6}$$

- Par contre, si le système est multi-agents, alors le but d'un agent i sera lié aux autres agents (par exemple manger l'agent j). Dans le cas d'une stratégie proie/prédateur, la fonction cible Δ_i^t de l'agent i change lorsque l'agent j (la proie de i) change de position dans l'environnement. Le but n'est donc plus fixe mais mouvant et on aura $\Delta_i^t \neq \Delta_i^{t+1}$. On aura donc $v_i = 1$ à chaque pas sauf pour ceux où l'agent ciblé j ne change pas de position (bloqué par un mur par exemple). Calculer v_i revient donc à calculer la probabilité de l'agent j à changer de position, c'est-à-dire trouver le taux d'actions effectuées par j aboutissant à un déplacement effectif.

Détermination de c_i, l_i et r_i

En revanche, la détermination des paramètres c_i, l_i et r_i est beaucoup plus problématique. Ceci est dû à la structure même des classeurs, et en particulier du fait de la généralisation qu'ils réalisent.

En effet, le comportement d'un agent (doublets état–action) est représenté par sa fonction de décision. Ainsi, à chaque décision correspond un classeur, ou plus exactement un *macro-classeur*. En effet, grâce à la généralisation, chaque classeur est valable pour plusieurs situations (dont le nombre est proportionnel au nombre de symboles #) et correspond donc à autant de *micro-classeurs* spécifiques, comme dans la figure suivante :

$$[01\#1]\ \uparrow\ = \begin{cases} [01\mathbf{0}1]\ \uparrow \\ [01\mathbf{1}1]\ \uparrow \end{cases}$$

FIG. 3.2 – Décomposition d'un classeur en sous-classeurs spécifiques.

Ainsi la modification d'un classeur (qui peut être au niveau de l'action ou de la condition) entraîne la modification d'un ou plusieurs sous-classeurs spécifiques. Il faut donc vérifier les propriétés l_i et r_i dans les micro-classeurs.

$$\boxed{[01\#1] \quad \uparrow \quad = \begin{cases} [0101] & \uparrow \\ [0111] & \uparrow \end{cases}} \longrightarrow \boxed{[01\#1] \quad \leftarrow \quad = \begin{cases} [0101] & \leftarrow \\ [0111] & \leftarrow \end{cases}}$$

FIG. 3.3 – Modification de l'action.

$$\boxed{[01\#1] \quad \uparrow \quad = \begin{cases} [0101] & \uparrow \\ [0111] & \uparrow \end{cases}} \longrightarrow \boxed{[0\#\#1] \quad \uparrow \quad = \begin{cases} [0001] & \uparrow \\ [0011] & \uparrow \\ [0101] & \uparrow \\ [0111] & \uparrow \end{cases}}$$

FIG. 3.4 – Modification de la condition.

Les deux figures précédentes montrent que la modification d'un seul attribut peut avoir des répercursions plus ou moins importantes sur les micro-classeurs. Pour chaque modification il faut calculer quel pourcentage de ces micro-classeurs satisfait la propriété.

$$[00\#\#0\#1\#] \quad \rightarrow \quad = \begin{cases} 1 & [00000010] & \rightarrow \\ 2 & [00000011] & \rightarrow \\ & \vdots & \\ 15 & [00110110] & \rightarrow \\ 16 & [00110111] & \rightarrow \end{cases}$$

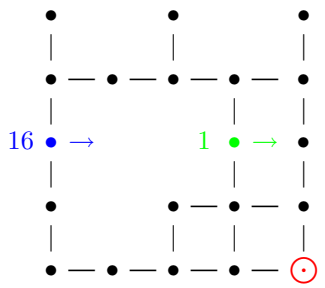

FIG. 3.5 – Exemple de classeurs corrects et incorrects.

Normalement, le taux de rétention r_i devrait être proche de 1, ce qui signifie que lorsque l'agent apprend un classeur correct, il ne l'oubliera pas dans le futur. Ceci implique qu'au fil du temps et des expériences réalisées la politique de l'agent doit se rapprocher de l'optimalité (c'est-à-dire $\delta_i^t \simeq \Delta_i^t$ ou $e(\delta_i^t) \simeq 0$).

Calcul de bornes

Signalons par ailleurs qu'il existe une méthode pour déterminer des bornes aux valeurs des paramètres c_i, l_i et r_i. Ceci est réalisé par une mesure de la complexité d'apprentissage appelée *sample complexity* tirée d'une théorie appelée "Probably Approximate Correct (PAC) theory". Cette PAC-theory impose deux importantes suppositions. La première est que les agents doivent avoir un apprentissage *consistant*, c'est-à-dire qu'une fois qu'un agent a appris un doublet $w \rightarrow a$ (dans notre cas un classeur) correct, il ne doit pas l'oublier. Ceci signifie simplement que l'on doit avoir $r_i = 1$. La deuxième supposition est que l'agent essaye d'apprendre une politique fixe, c'est-à-dire $\Delta_i^{t+1} = \Delta_i^t$ pour tout t.

Malheureusement, la première supposition n'est probablement pas faisable du fait que la population de classeurs évolue constamment. Quant à la deuxième, elle n'est pas possible dans notre cas. En effet, les agents étant eux-mêmes les prédateurs d'autres agents, la fonction cible Δ_i^t change après chaque déplacement de l'agent visé.

3.2.3 Expérimentations et résultats

L'implémentation de ce framework dans LCSTalk a été effectuée en deux phases. La première fut d'implanter le nécessaire afin de pouvoir obtenir expérimentalement l'erreur d'un agent, conformément à l'équation 3.1. La deuxième fut de trouver expérimentalement (de la même manière que Vidal et Durfee) la valeur des paramètres du framework.

Expérimentations

Tout d'abord, il a fallu implémenter la fonction cible afin de la comparer avec les décisions prises par l'agent. Cette simple comparaison nous donnera directement le taux d'erreur commis par l'agent.

Calculer la fonction cible revient à créer une liste avec tous les doublets $w \rightarrow a$ qui sont optimaux pour un but donné. Une façon simple de le faire est de transformer l'environnement en graphe (comme montré dans la figure 3.6). Il suffit alors, pour chaque état w de l'environnement, de calculer quelles actions sont optimales grâce à un parcours en largeur. La figure 3.7 montre, pour une position de l'agent dans l'environnement, quelles sont les actions optimales et les mauvaises actions.

Résultats

En ce qui concerne le calcul de l'erreur théorique de l'agent, le problème était de trouver expérimentalement la valeur des paramètres du framework, de manière à ce que la courbe théorique se rapproche le plus possible de la courbe réelle.

Les tests ont été réalisés avec le système de classeurs XCS sur l'environnement appelé Woods1. Deux tests ont été effectués, l'un mono-agent, l'autre multi-agents. Les résultats sont exposés aux figures 3.8 et 3.9, page 30.

Ces résultats parlent d'eux-mêmes. Comme on peux le voir sur la première figure, l'erreur de l'agent tend rapidement vers une faible valeur (< 0.2). En

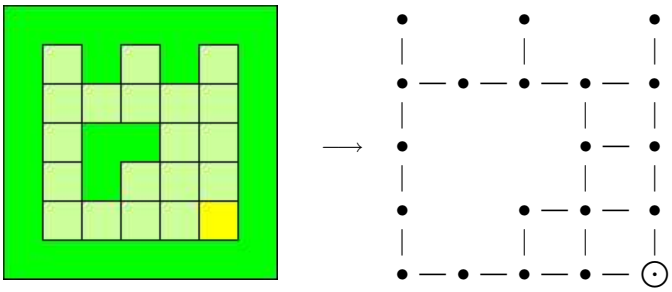

FIG. 3.6 – Transformation d'un environnement Maze228 en graphe non orienté où le symbole • représente un nœud, les symboles {|,—} un lien (uniquement les déplacements valides), et le symbole ⊙ le but à atteindre.

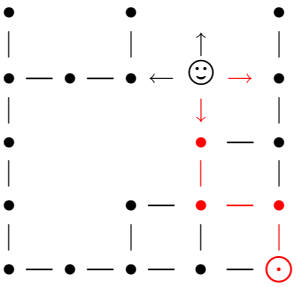

FIG. 3.7 – Fonction cible et fonction décision.
• — • représente un chemin optimal.

revanche, dans le test multi-agents, on remarque que cette erreur n'évolue pas et reste élevée (autour de 0.8), bien que les performances "nombre de pas / problème" s'améliorent au fil du temps (voir un aperçu figure 2.1, page 18) et sont en moyenne meilleures qu'en employant une stratégie de déplacement aléatoire. Cette erreur est en grande partie due au fait que la fonction cible change constamment (valeur de v_i élevée). De plus la structure même de ces problèmes impose que l'erreur initiale soit très élevée car, sur huit directions possibles, il n'y en a généralement qu'une seule d'optimale.

Ceci tendrait à montrer que les systèmes de classeurs tels que XCS ou ZCS ne seraient effectivement pas adaptés pour des problèmes multi-agents. Il faudrait alors pouvoir tester si d'autres systèmes de classeurs n'obtiendraient pas de meilleures performances, en particulier des systèmes de classeurs exploitant des notions d'anticipation tels que ACS ou MACS.

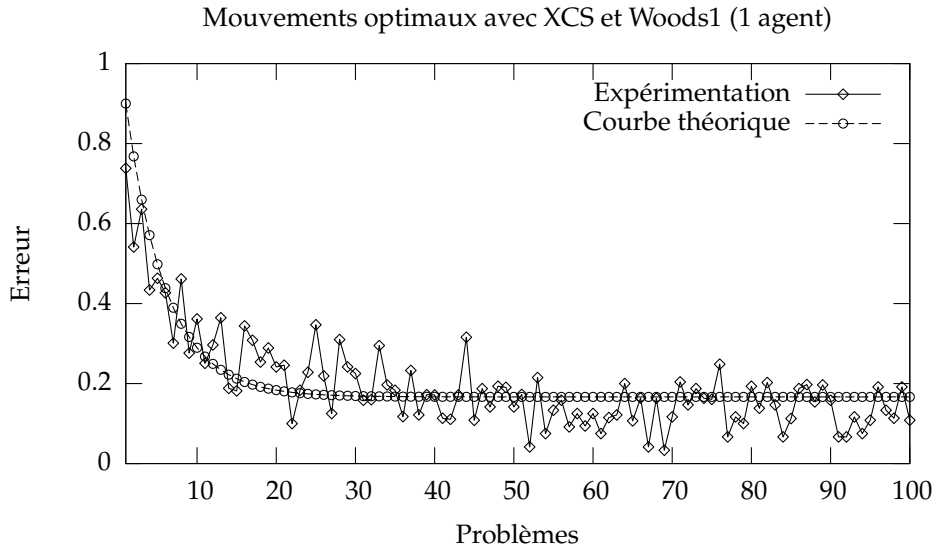

FIG. 3.8 – Courbes d'erreurs expérimentale et théorique lors d'un test mono-agent. Paramètres de la courbe théorique (*cf.* équation 3.6) : $l_i = 0.15,\ r_i = 0.97$ et erreur initiale $e(\delta_i^1) = 0.9$.

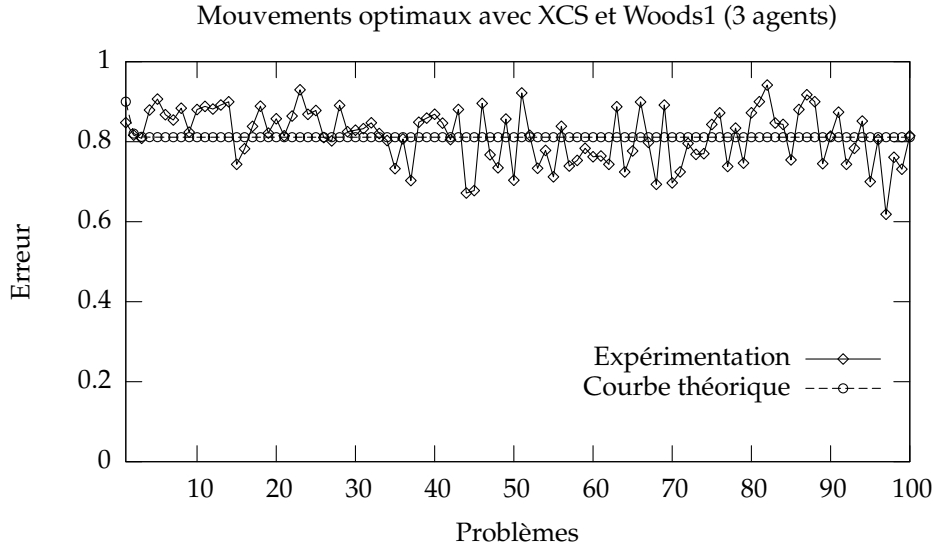

FIG. 3.9 – Courbes d'erreurs expérimentale et théorique lors d'un test multi-agents. Paramètres de la courbe théorique (*cf.* équation 3.5) : $c_i = 0.5,\ l_i = 0.2,\ r_i = 0.9,\ v_i = 0.74$, $|A_i| = 8$ et erreur initiale $e(\delta_i^1) = 0.9$.

Conclusion

Après avoir présenté ce qu'était le formalisme des systèmes de classeurs et en avoir décrit les trois plus connus, nous avons vu que le méta-apprentissage était, en l'état actuel des choses, un formalisme prometteur mais peut-être pas très adapté aux systèmes de classeurs du fait du temps de calcul important inhérent à son fonctionnement.

Puis j'ai présenté un framework modulaire pour les systèmes de classeurs nommé LCSTALK. Cette plateforme est déjà utilisée par quelques personnes de l'équipe et a même été nominé lors de la conférence de l'ESUG 2004. Néanmoins, elle n'est pas complètement terminée et nécessiterait l'ajout d'autres systèmes de classeurs tels que MACS par exemple, ainsi que l'ajout d'autres types de stratégies suivant les besoins.

Enfin, on a vu que l'adaptation du framework CLRI aux systèmes de classeurs était possible mais pas toujours aisée. L'implantation et la comparaison expérimentales des fonctions cible et de décision est assez simple. En revanche, le calcul de tous les paramètres directement à l'intérieur des algorithmes d'apprentissage n'a pas toujours été possible du fait de leur structure et de l'architecture du système. Cependant, il pourrait être intéressant dans une étude ultérieure, d'utiliser des processus réflexifs et la programmation par aspects (AOP) pour résoudre ce problème. En effet, l'agent n'a pas connaissance de sa fonction cible mais doit pourtant s'efforcer de faire tendre sa fonction de décision le plus près possible de celle-ci. Et de même que les paramètres CLRI agissent plutôt *sur* les classeurs, indépendamment de ceux-ci, que *dans* les classeurs, tous les concepts de ce framework seraient plus facilement implémentables s'ils étaient considérés comme des aspects du système de classeurs.

Bibliographie

[Hoffmann, 1993] Hoffmann, J. (1993). *Vorhersage und Erkenntnis*. Hogrefe.

[Holland, 1985] Holland, J. H. (1985). Properties of the bucket brigade. In Grefenstette, J. J., editor, *Proceedings of the 1st International Conference on Genetic Algorithms and their Applications (ICGA85)*, pages 1–7. Lawrence Erlbaum Associates : Pittsburgh, PA.

[Kovacs, 1997] Kovacs, T. (1997). XCS Classifier System Reliably Evolves Accurate, Complete, and Minimal Representations for Boolean Functions. In Roy, Chawdhry, and Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 59–68. Springer-Verlag, London. ftp ://ftp.cs.bham.ac.uk/pub/authors/T.Kovacs/index.html.

[Livolant, 2003] Livolant, E. (2003). *Apprentissage multi-agents par systèmes de classeurs : étude préliminaire*. Stage de DEA, Université de Caen.

[Sagit, 2004] Sagit, O. (2004). *Robotique Reconfigurable et Apprentissage par Renforcement*. Stage de DEA, Université de Caen.

[Schmidhuber, 1996] Schmidhuber, J. (1996). A general method for incremental self-improvement and multi-agent learning. In Yao, X., editor, *Evolutionary Computation : Theory and applications*, Singapore. Scientific Publ. Co.

[Seward, 1949] Seward, J. P. (1949). An experimental analysis of latent learning. *Journal of Experimental Psychology*, 39 :177–186.

[Stinckwich et al., 2004] Stinckwich, S., Livolant, E., and Piel, M. (2004). Apprendre à anticiper dans un système multi-agents. In *Le temps dans les systèmes complexes. 11èmes journées de Rochebrune*, pages 173–186, Paris. ENST.

[Stolzmann, 1999] Stolzmann, W. (1999). Latent Learning in Khepera Robots with Anticipatory Classifier Systems. In Wu, A. S., editor, *Proceedings of the 1999 Genetic and Evolutionary Computation Conference Workshop Program*, pages 290–297.

[Vidal and Durfee, 2003] Vidal, J. M. and Durfee, E. H. (2003). Predicting the expected behavior of agents that learn about agents : the CLRI framework. *Autonomous Agents and Multi-Agent Systems*, 6(1) :77–107.

[Wilson, 1994] Wilson, S. W. (1994). ZCS : A zeroth level classifier system. *Evolutionary Computation*, 2(1) :1–18. http ://prediction-dynamics.com/.

[Wilson, 1995] Wilson, S. W. (1995). Classifier Fitness Based on Accuracy. *Evolutionary Computation*, 3(2) :149–175. http ://prediction-dynamics.com/.

[Wilson, 1996] Wilson, S. W. (1996). Explore/exploit strategies in autonomy. In Maes, P., Mataric, M. J., Meyer, J.-A., Pollack, J., and Wilson, S. W., editors,

From Animals to Animats 4. Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (SAB96), pages 325–332. A Bradford Book. MIT Press.

[Zhao and Schmidhuber, 1996] Zhao, J. and Schmidhuber, J. (1996). Incremental self-improvement for life-time multi-agent reinforcement learning. In Maes, P., Mataric, M., Meyer, J.-A., Pollack, J., and Wilson, S. W., editors, *From Animats to Animats 4 : Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, Cambridge, MA. MIT Press, Bradford Books.