Proposition d'un modèle générique pour l'implémentation d'une famille de systèmes de classeurs

Cédric Buche — Cyril Septseault — Pierre De Loor

Laboratoire d'Informatique des Systèmes Complexes (LISyC)
Centre Européen de Réalité Virtuelle (CERV)
BP 38 F - 29280 Plouzané
[buche,septseault,deloor]@enib.fr

RÉSUMÉ. *Les systèmes de classeurs sont des outils performants pour l'apprentissage des interactions entre un agent autonome et son environnement, mais ceux-ci sont multiples et il n'existe pas de règles formelles de choix d'un modèle précis à partir d'un problème considéré. Cet article propose une analyse et une comparaison de la structure et de la dynamique des principaux modèles de systèmes de classeurs existants, afin d'aboutir à un modèle générique permettant de les supporter. Il présente l'implémentation de ce modèle, sa spécialisation pour certains systèmes de classeurs et son utilisation pour des applications diverses.*

ABSTRACT. *Classifiers systems are adapted tools to learn interactions between an autonomous agent and its environment. However, there are many kinds of classifiers systems which differ by numerous subtle technical features. There is no systematic rule to choose among them according to a given problem. This article analyzes the major kinds of classifiers systems in order to suggest a generic model common to all of them. This model and some of its specificities are presented. It has been developped for different applications which are also described.*

MOTS-CLÉS : *adaptabilité, apprentissage, dynamicité, systèmes de classeurs.*

KEYWORDS: *adaptability, learning, dynamicity, classifiers systems.*

RSTI - RIA. Volume 20 – n° 1/2006, pages 63 à 88

1. Introduction

La définition du comportement d'entités artificielles autonomes est confrontée au problème du choix d'un modèle qui permet de rendre compte de façon synthétique et efficace du lien entre les perceptions et les actions. Il existe de nombreuses propositions qui nécessitent une description exhaustive, difficile à élaborer, soit parce qu'elles demandent une définition basée sur des règles et des symboles *a priori* (Carver *et al.*, 1992; Mateas, 1999; Cavazza *et al.*, 2001), soit parce qu'elles sont sujettes à des problèmes de paramétrage et d'indéterminisme comportemental (Brooks, 1990; Maes, 1989). Une autre solution consiste à doter les entités d'un comportement initial approximatif qui va s'adapter par la suite à l'environnement. Cette solution est mise en œuvre par le biais des systèmes de classeurs. En effet, ceux-ci présentent l'avantage d'être composés d'un ensemble de règles mises en concurrences et d'incorporer des processus d'apprentissage du choix de ces règles et de leur amélioration. Une large littérature existe à leur sujet (Wilson, 1994; Wilson, 1995; Cliff *et al.*, 1995; Stolzmann, 1998; Lanzi *et al.*, 1999b; Tomlinson *et al.*, 1999b). De nombreux auteurs proposent des variantes offrant des mécanismes adaptés à des problèmes spécifiques. Notre objectif est de pouvoir aisément tester et enrichir ces mécanismes. C'est la raison pour laquelle nous nous sommes intéressés à l'élaboration d'un modèle support générique et à son implémentation.

Cet article est structuré de la façon suivante : dans un premier temps nous présentons les mécanismes généraux des systèmes de classeurs et nous montrons leur utilisation dans les systèmes ZCS et XCS. Ensuite, nous présentons un modèle générique intégrant ces mécanismes et permettant de tester facilement différentes versions. Puis, nous montrons comment nous avons utilisé ce modèle pour différents types d'application : le multiplexeur, l'environnement *Woods* et l'optimisation d'un algorithme de résolution distribuée et adaptative du problème de la génération d'emplois du temps. Enfin, nous envisageons nos perspectives concernant l'utilisation des systèmes de classeurs pour l'adaptation de stratégies pédagogiques dans les environnements virtuels de formation.

2. Les systèmes de classeurs

2.1. *Principes*

Un système de classeurs gère une base de règles appelées classeurs de la forme « condition-action », qui peuvent être pondérées par des attributs de qualité, lorsque l'on cherche à obtenir un système apprenant. Le système possède un cycle de fonctionnement au cours duquel il perçoit son environnement, en déduit les règles applicables, exécute une action issue de ces règles et peut percevoir une rétribution de l'environnement qu'il utilise alors pour modifier les règles ou leurs attributs de qualité. C'est la qualité d'une règle, associée à l'adéquation entre sa partie condition et la perception de l'environnement, qui conditionne son choix. Les systèmes de classeurs permettent donc d'apprendre, par expérimentation, l'association de conditions et d'actions maxi-

misant la récolte de rétributions. Pour éviter l'explosion combinatoire du nombre de règles, celles-ci sont généralisantes : elles s'appliquent à différentes perceptions de l'environnement. Il faut alors utiliser des mécanismes permettant de les créer, de les enrichir (spécialisation/généralisation) ou de les détruire. C'est souvent un algorithme évolutionniste qui s'en charge, bien qu'il existe d'autres heuristiques. Les qualités des règles sont modifiées dynamiquement par le biais d'un apprentissage par renforcement et les règles sont modifiées par les algorithmes génétiques.

La figure 1 synthétise les interactions entre un système de classeurs et son environnement. L'interface d'entrée permet de représenter une perception de l'état de l'environnement en un formalisme identifiable avec la représentation des règles. L'interface de sortie permet d'exécuter l'action choisie en faisant une opération analogue dans l'autre sens. La rétribution est considérée comme une valeur discrète dont l'occurrence n'est pas systématique, mais qui est associée à un état de l'environnement.

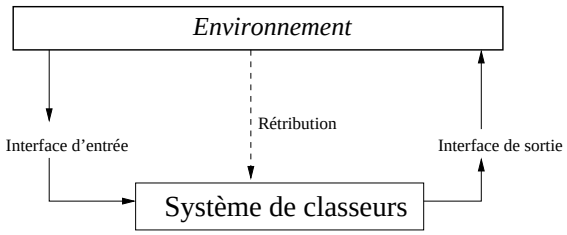


Figure 1. *Interactions entre l'environnement et le système de classeurs*

2.2. *Formalisation*

Dans cette partie, nous allons proposer une formalisation incrémentale et générique des systèmes de classeurs en introduisant progressivement les mécanismes d'apprentissage.

Structure de base

La structure globale d'un système de classeurs, présentée en figure 2, est un 7-uplet (`Ie`, `[P]`, `[M]`, `[A]`, `Comparaison`, `Sélection`, `Io`) :

– `Ie` est l'interface d'entrée, faisant correspondre à toute `Perception` de l'environnement un code binaire.

– `[P]`, appelé *population*, est l'ensemble des classeurs du système, codés par une succession de *n* bits[1]. Les représentations généralisantes contiennent des symboles $\#$ correspondant à une valeur indéterminée. Une règle est un couple (C, A) avec $C \cup A \in \{0, 1, \#\}^n$ avec :

1. Même si certains systèmes travaillent sur d'autres alphabets (Matteucci, 1999; Wilson, 2000; Heguy *et al.*, 2002).

- C : la condition d'application de la règle.

- A : la ou les actions associées à l'application de la règle.

Prenons l'exemple d'un robot qui possède quatre capteurs tout ou rien et une action. L'interface d'entrée transforme l'état des capteurs en une valeur binaire et l'interface de sortie déclenche l'action en fonction de la valeur du bit d'action. Ainsi, une règle $\{011\#, 1\}$ signifie que la règle est applicable si le premier capteur est inactif et les deux suivants actifs. L'état du quatrième capteur n'a pas d'influence et l'application de la règle déclenche l'action.

– `[M]` $\subseteq$ `[P]` est l'ensemble des classeurs dont la partie condition s'apparie avec les informations perçues de l'environnement pour un cycle de sélection. Il est appelé *Match-set*.

– `[A]` $\subseteq$ `[M]` est l'ensemble des classeurs représentant l'action sélectionnée. Il est appelé *Action-set*.

– `Comparaison` est le mécanisme permettant de passer de `[P]` à `[M]`. Il s'agit généralement d'une règle d'appariement entre C et l'information provenant de `Ie`. Cette règle sait interpréter les symboles de généralisation composant les conditions des classeurs.

– `Sélection` est le mécanisme permettant de passer de `[M]` à `[A]`. Il détermine, en fonction de critères spécifiques aux différentes versions de systèmes de classeurs, l'action choisie.

– `Io` est l'interface de sortie faisant correspondre à un code binaire l'activation d'`Action`.

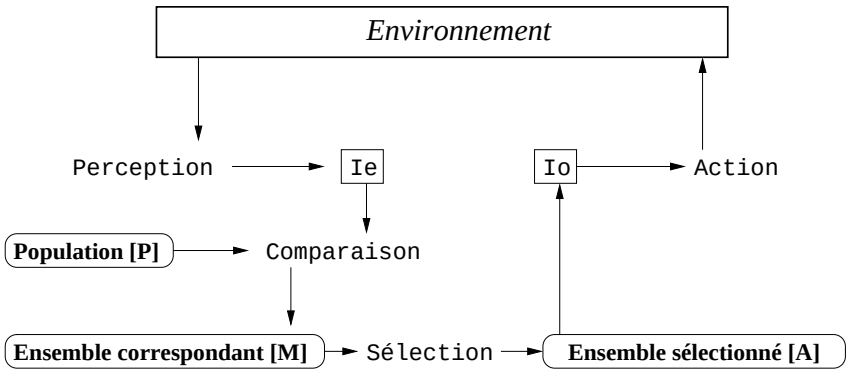


Figure 2. *Structure et dynamique d'un système de classeurs*

Apprentissage

L'apprentissage s'effectue grâce à une évaluation de la qualité des règles, représentée par un ou plusieurs paramètres supplémentaires. La définition d'un classeur est donc étendue à un triplet $R = (C, A, f)$ où f caractérise la qualité de ce dernier. L'apprentissage élaboré par une `Rétribution` des règles, modifiant leur qua-

lité grâce à des algorithmes d'apprentissage par renforcement et par `Génération` de règles à l'aide d'algorithmes évolutionnistes et d'heuristiques de recouvrement (*covering*). La dynamique des systèmes de classeurs apprenants est alors basée sur le cycle : `Perception` / `Comparaison` / `Génération` *(covering)* / `Sélection` / `Action` / `Rétribution` / `Génération` *(algorithme évolutionniste)* dont le pseudo code est représenté en figure 3.

```
t := 0 ;                                    // compteur d'itérations
initClassifierPopulation( P(t) ) ;          // population de classeurs initiale
while not done do                           // critère de terminaison (temps, etc.)
   t := t +1 ;
   // Perception : 1. codage de la perception
   Ie(t) := readDetectors(t) ;
   // Comparaison : 2a. comparaison de [P] et Ie et sauvegarde les appariements dans [M]
   M(t) := matchClassifiers( Ie(t),P(t),Comparaison ) ;
   // Génération (1) Covering : 2b. créer des règles s'appariant avec Ie (si M vide ou criterion1)
   if ( M.size < criterion0 || criterion1 ) then
                              M(t) := cover( Ie(t),P(t),Covering ) ;
   // Sélection : 3. sélection des règles dans [A]
   A(t) := selectClassifiers( M(t),Sélection ) ;
   // Actions : 4. envoie de l'action via Io
   Io(t) := sendEffectors( A(t) ) ;
   // Rétribution (1) : 5. réception de la rétribution de l'environnement
   r := receivePayoff(t) ;
   // Rétribution (2) : 6. distribution de la rétribution aux classeurs
   P(t) := distributeCredit( r,P(t),P(t-1),Rétribution ) ;
   // Génération (2) A.E. : 7. éventuellement (selon t) un Algorithme Evolutionniste est utilisé sur [P]
   if ( criterion2 ) then
         P(t+1) := reviseRules( P(t),Algorithme_Evolutionniste ) ;
```

Figure 3. *Pseudo code représentant les 7 étapes du cycle d'un système de classeurs apprenant*

Mécanisme de `Sélection`

La sélection est guidée par la qualité des règles qui sont regroupées selon leur partie `A`. Souvent, c'est un mécanisme de roue de la fortune[2] qui est appliqué, ce qui signifie que chaque paquet a une probabilité proportionnelle à sa qualité d'être sélectionné. Il existe différentes versions d'algorithmes de sélection pour favoriser, par exemple, la taille des paquets, la qualité de la meilleure règle, un compromis entre les deux, etc.

2. Le mécanisme de roue de la fortune consiste à effectuer un tirage aléatoire entre des éléments pour lesquels la probabilité d'être choisi est proportionnelle à leur sélectivité.

Mécanisme de *Rétribution*

Ce mécanisme (*credit assignment* en anglais) distribue les rétributions aux règles qui ont contribué à les obtenir. Il augmente la qualité des règles déclenchées précédement à l'obtention de la rétribution et diminue les autres. Sa définition influe sur la longueur pertinente d'une chaîne d'actions : nombre d'enchaînement de règles considérées nécessaires pour atteindre un but.

Mécanisme de *Génération*

Le mécanisme de génération doit à la fois minimiser le nombre de règles tout en préservant celles qui permettent d'atteindre les rétributions. Une bonne règle est donc une règle généralisante ayant une qualité importante (relativement aux autres). Les deux mécanismes de génération (*rules discovery*) utilisés sont le *covering* et les *algorithmes génétiques*.

– Le *covering* permet de créer des règles lorsque aucun classeur ne s'apparie à la perception de l'environnement. Cela signifie que la population `[P]` ne possède pas un nombre suffissant de règles permettant de proposer une action relative à la situation. Le *covering* peut également créer des règles lorsque les qualités des classeurs appariés sont considérées insuffissantes. Dans les deux cas, de nouvelles règles sont créées avec une partie condition adéquate plus ou moins généralisante. Dans ce cas, la probabilité d'obtenir un # est un paramètre à fixer. La partie action est choisie aléatoirement et la *Qualité f* est souvent la moyenne des qualités de `[P]`. Imaginons par exemple que le message d'entrée provenant de `Ie` soit 0111. La population `[P]` ne possède pas de règle correspondante. Le *covering* crée une règle dont la condition peut être 0111 ou $\#111$ ou $0\#\#1$. Cette méthode permet notamment d'initialiser le système avec une population `[P]` vide et évite la génération de règles ne correspondant à aucun état possible de l'environnement.

– Les *algorithmes génétiques* (Holland, 1975; Goldberg, 1989) sont utilisés pour générer de nouvelles règles à partir de celles existantes. Ils utilisent des opérations évolutionnistes telles que le croisement ou la mutation[3]. La sélection des règles servant de base pour les opérations génétiques est généralement effectuée par un mécanisme de roue de la fortune. Les algorithmes génétiques peuvent intervenir sur les règles se situant dans `[P]` ou dans `[A]` selon les versions de système de classeurs (Wilson, 1994; Wilson, 1995). Ce mécanisme permet au système de s'adapter plus rapidement aux environnements dynamiques. La fréquence d'utilisation des algorithmes génétiques par rapport au cycle d'un système de classeurs est un facteur important pouvant influencer sensiblement les performances d'apprentissage. La sélection par algorithmes génétiques peut encore être raffinée. Deux types de systèmes de classeurs se distinguent : le type *Michigan* (Smith, 1980) et le type *Pittsburgh* (Holland

3. Le croisement divise deux règles en un point de croisement. Le croisement génère une nouvelle règle qui est le résultat de la partie gauche d'une des règles et de la partie droite de l'autre règle. La mutation consiste à modifier un ou plusieurs bits d'une règle pour générer une nouvelle règle.

et al., 1978). La méthode *Michigan* applique les algorithmes génétiques en utilisant un unique système où chaque règle est un individu. La méthode *Pittsburgh* considère un individu comme une population entière (ensemble de règles). Dans ce dernier, l'opérateur de croisement mêle les ensembles de règles. Une valeur sélective est attribuée à chacune des populations. La méthode *Pittsburgh* permet alors de les mettre en compétition, néanmoins la gestion d'un aussi grand nombre de règles est plus lourde et convient difficilement à un apprentissage incrémental d'un environnement dynamique. Le mécanisme de suppression est simple puisqu'il s'agit d'éliminer les individus les « moins bons ».

3. Différentes versions

La complexité des premiers systèmes tels que le CS1 n'a pas donné de résultats concluants [4]. En effet, le CS1 peut s'envoyer lui-même des messages à l'aide de la liste des messages, ce qui crée des cycles d'inférences internes ayant tendance à développer et à maintenir des règles parasites. Contrairement à certains travaux qui essayent de corriger les déficiences de performance des systèmes originaux, Wilson choisit une approche qui consiste à les simplifier et propose le ZCS. En supprimant la liste des messages, il réduit les systèmes originaux aux éléments essentiels tout en conservant la même architecture.

3.1. *ZCS*

Le ZCS (Zeroth level Classifier System) a été présenté par Wilson comme un système de classeurs de type *Michigan* (Wilson, 1994). Son fonctionnement est représenté en figure 4. Ce type de système de classeurs utilise des règles sous la forme classique du triplet $R = (C, A, f)$. Il précise les mécanismes suivants :

Sélection

La `Sélection` des règles s'effectue en tenant compte de la *qualité* des classeurs de `[M]`. Lors de la phase d'exploitation elle est généralement soumise au mécanisme de roue de la fortune.

Rétribution

La `Rétribution` est soumise à un mécanisme proche du Q-Learning (Sutton, 1984; Sutton, 1988) : le *Bucket Brigade*[5] (Dorigo *et al.*, 1994). La rétribution de l'environnement est l'origine de la chaîne de rétribution, qui passe par tous les classeurs qui ont participé aux déclenchements de l'action, et se termine par les classeurs qui

4. Le CS1 apparaît en 1978 (Holland *et al.*, 1978) : une situation ou « état perçu » est stockée dans une mémoire perceptive de taille limitée appelée liste des messages, assimilée à la « mémoire à court terme » des sciences cognitives.
5. *Bucket Brigade* : chaîne de seaux.

sont à l'origine de toute l'action. Chaque règle prend la part qui lui revient et fait passer ce qu'il reste aux autres classeurs, via l'utilisation d'un facteur d'actualisation (*discount factor*). L'algorithme (Holland, 1985) intègre un système économique dans la population de classeurs [P]. L'idée est d'assimiler les classeurs et l'environnement à des agents financiers. La qualité d'une règle peut alors être vue comme le capital de l'agent. Le mécanisme de marché met en place une compétition entre les agents concernés par la proposition de l'environnement pour avoir le droit de participer aux enchères. Pour chaque classeur de [P], trois paramètres sont introduits : une *Enchère Cbid*, une *Taxe Ctaxe* et une *Rétribution Cr*. Nous pouvons considérer un classeur sous la forme $R = (C, A, f, s, Cbid, Ctaxe, Cr)$. La *Qualité f* de chaque classeur de [P] est mise à jour à chaque cycle de fonctionnement, suivant la formule : $f(t) = f(t-1) - Cbid * f(t-1) - Ctaxe * f(t-1) + Cr$.

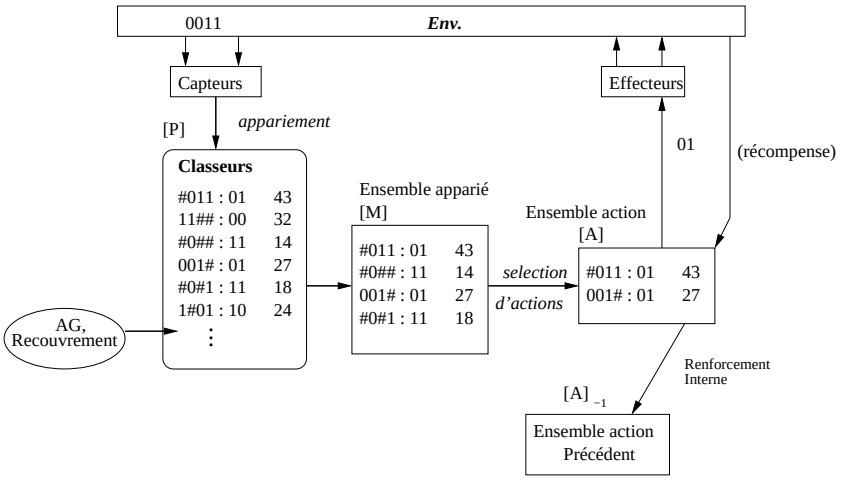


Figure 4. *Système de classeurs de type ZCS d'après (Wilson, 1994)*

Génération

La `Génération` s'effectue à l'aide du *covering* ou d'algorithmes génétiques s'appliquant au niveau de la population [P] en considérant la qualité des classeurs comme valeur sélective. La `Génération` agit à fréquence constante. L'opérateur de mutation utilise des « parents » issus de la méthode de la roue de la fortune sur [P]. Les nouveaux classeurs remplacent les règles les plus faibles afin de maintenir une population constante. La suppression sélectionne des règles en utilisant la méthode de la roue de la fortune sur [P] en considérant la valeur inverse de la qualité. La qualité des nouvelles règles est initialisée par la valeur moyenne des qualités des « parents ». Dans le cas du *covering* elle est initialisée par la valeur moyenne des qualités sur [P].

Paramètres

Différents paramètres sont nécessaires pour utiliser ce système (Wilson, 1994). Nous avons relevé la taille de la population, la qualité initiale des classeurs, le taux d'enchère, le taux de récompense, le taux de *covering*, la fréquence d'appel des algorithmes génétiques, le taux de croisement et le taux de mutation.

Extension du ZCS

De façon superficielle, on peut indiquer que la capacité de perception de l'environnement d'une entité se scinde selon deux types :

1) les informations extraites de la perception immédiate de l'entité fournissent toutes les informations nécessaires pour choisir la meilleure action dans toutes les situations. Dans ce cas, l'environnement est *markovien*. L'entité n'a pas besoin de mémoriser l'histoire du système pour connaître l'état de l'environnement à un instant donné, seule sa perception immédiate y suffit ;

2) les informations extraites de la perception immédiate de l'entité fournissent des informations partielles sur l'environnement. Dans ce cas, il est possible qu'il existe différentes situations qui apparaissent comme identiques pour l'entité, nécessitant des actions différentes optimales. Par conséquent, l'entité ne peut pas choisir la meilleure action en ne considérant que ses informations sensorielles immédiates. L'environnement est *non markovien*. Afin de pouvoir différencier ces états, l'entité doit prendre en compte son historique et peut utiliser une mémoire explicite (registre interne) ou implicite (chaînage entre les décisions).

Certains systèmes proposent d'étendre les capacités du ZCS à des environnements non markoviens :

– le ZCSM (Mémoire) (Cliff *et al.*, 1995) ajoute une mémoire temporaire au ZCS. Chacune des parties (condition et action) des règles est étendue avec une sous-chaîne de bits internes : la mémoire. Le choix des règles dépend donc de la valeur de la mémoire qui est également modifiée par les règles. Cette méthode permet d'augmenter les capacités des ZCS à des environnements non markoviens ; la difficulté étant de définir une taille pertinente pour cette mémoire ;

– le ZCCS (Corporation) (Tomlinson *et al.*, 1999b) définit des liens entre les règles. Chacune possède un lien vers la règle précédente et un lien vers la règle suivante. Une corporation est un ensemble de classeurs liés entre eux. La méthode de croisement utilise des règles sélectionnées de la corporation.

Limites

Comme le souligne Gérard (Gérard, 2002), le ZCS est sujet au problème de la maintenance des longues chaînes d'actions. En effet, le mécanisme d'enchère permet de rétribuer la chaîne des règles qui ont permis d'arriver à l'action proposée. Les classeurs appariés à des situations éloignées de toute récompense sont donc rétribués, mais d'une façon moindre par rapport aux classeurs qui sont en fin de chaîne menant

à la rétribution (*discount factor*). Comme il utilise la qualité des classeurs comme valeur sélective pour l'algorithme génétique, les classeurs intégrant une chaîne d'action mais étant éloignés de la rétribution sont rapidement éliminés. Or, s'ils sont les seuls à proposer une action adéquate pour une perception donnée, ils sont pertinents et ne devraient pas être éliminés. Ce problème augmente bien sûr avec la taille de l'environnement.

3.2. *XCS*

Le XCS (Wilson, 1995) est une version de système de classeurs basée sur le ZCS. Son fonctionnement est représenté en figure 5.

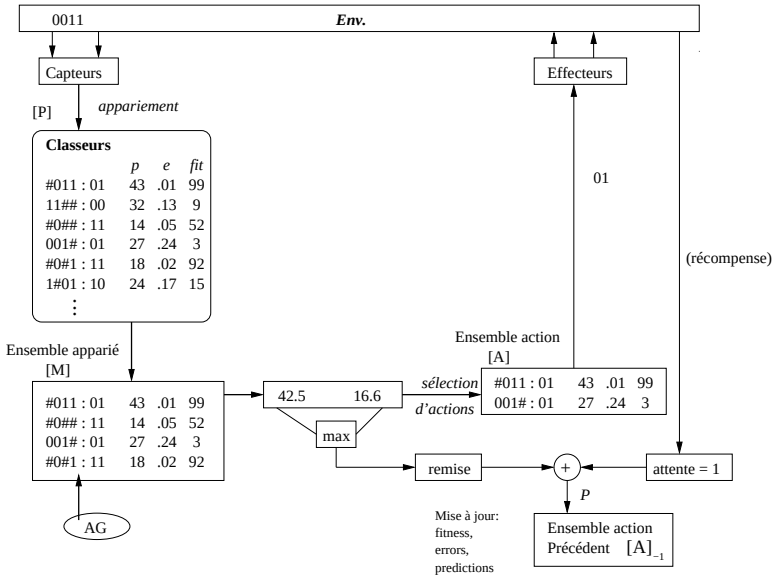


Figure 5. *Système de classeurs de type XCS d'après (Wilson, 1995)*

Ces spécificités sont les suivantes :

Règles

Les règles sont composées de trois paramètres venant compléter les parties condition et action. La *Qualité f* du ZCS est ainsi décomposée :

– la *Prédiction de paiement p* représente la récompense attendue en effectuant l'action A dans les situations appariées par C,

– l'*Erreur sur la prédiction e* représente l'écart entre la *Prédiction de paiement p* et le paiement réel,

– la *Fitness fit* est la fonction inverse de *e*, elle représente donc l'exactitude de la *prédiction de paiement p*.

Les classeurs peuvent être définis par $R = (C, A, p, e, fit)$ avec C et $A \in \{0, 1, \#\}^n$ et $p, e, fit \in \Re$.

Sélection

La `Sélection` se base non plus sur la *Qualité f* du classeur du ZCS mais sur la précision de la *Prédiction de paiement p*. Cette méthode permet de sélectionner les classeurs n'ayant pas seulement une action optimale mais plutôt ceux qui permettent de donner la qualité de l'action proposée avec précision quelle que soit sa position dans une chaîne d'action. Elle nécessite un calcul des prédictions (*Prediction Array* `PA`). Pour chaque action a_i présente dans `[M]`, une prédiction $p(a_i)$ est calculée. (Wilson, 1996) propose de séparer explicitement les stratégies d'exploration et d'exploitation :

– pour l'exploitation, la sélection est déterministe et basée sur la plus haute prédiction ; les algorithmes génétiques sont inhibés ;

– pour l'exploration, la sélection est soumise à une probabilité permettant de sélectionner aléatoirement ou de sélectionner la plus haute prédiction.

Rétribution

La `Rétribution` n'est plus un *Bucket Brigade* suite à l'article de (Dorigo *et al.*, 1994). Elle est soumise à un mécanisme de « Q-Learning dérivé », la *Prédiction de paiement p* représente la fonction d'utilité $Q(s, a)$ du Q-Learning et la rétropropagation de la récompense utilise les équations de Bellman. Une fois la `Sélection` effectuée, l'ensemble des actions précédentes $\texttt{[A]}_{-1}$ (l'ensemble `[A]` lors du dernier cycle) est modifié en utilisant une combinaison de la dernière rétribution de l'environnement et la prédiction maximale de la `PA` actuelle (voir figure 5). Le système ne cherche plus à trouver des classeurs adaptés au problème mais à trouver une approximation de la fonction d'utilité $Q(s, a)$ sans se limiter aux classeurs proposant une action optimale.

Génération

La `Génération` est effectuée à l'aide du *covering*. Il permet d'initialiser `[P]` avec un ensemble vide ou très réduit. Le processus d'algorithme génétique ne se situe plus au niveau de la population `[P]` mais au niveau de `[M]` et se base sur l'erreur de prédiction. La *Fitness fit* est la valeur sélective. Si l'erreur est grande alors la valeur sélective est faible et inversement. Le détail des opérations est présenté dans (Butz *et al.*, 2002). (Butz *et al.*, 2001) a montré que cette méthode favorise le maintien dans `[P]` des classeurs les plus généraux. Le classeur est conservé tant que son erreur est faible.

Paramètres

Différents paramètres sont nécessaires pour utiliser ce système. Le lecteur pourra se référer à (Butz, 1999; Butz, 2000; Butz *et al.*, 2002) pour plus de détails concernant leur description et leur utilisation.

Le problème de la maintenance de la longueur de chaîne d'actions

Ce système résout le problème de la maintenance de la longueur de chaînes d'actions en ne cherchant pas directement à résoudre le problème de la maximisation de la récompense attendue. Le mécanisme sépare l'apprentissage de la sélection (voir figure 6). Il apprend un modèle de la fonction de récompense en utilisant l'*Erreur sur la prédiction e* et sélectionne les classeurs en utilisant la *Prédiction de paiement p*.

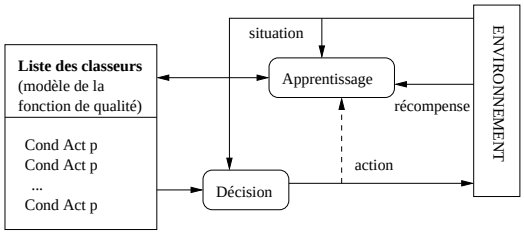


Figure 6. *Architecture de XCS d'après (Gérard, 2002)*

Améliorations

Des modifications ont été proposées afin d'améliorer le système dans des cas précis :

– le POP-XCS (optimisé) (Kovacs, 1996) propose d'étendre la généralisation au maximum. La méthode permet de converger vers une population réduite pour des problèmes markoviens ;

– une classification des règles (Kovacs, 1997) selon la capacité de généralisation (nombre de #) : surgénéralisée, généralisée au maximum et généralisée sous optimal. Cette classification oriente la généralisation des règles ;

– l'opérateur de « subsumption » (Wilson, 1998) permet de réduire la taille de la population afin de la restreindre à un ensemble de classeurs généraux. Un classeur subsume un autre classeur si sa partie condition est plus générale et si leurs parties action sont identiques. Le classeur subsumé est effacé. Un classeur n'est capable de subsumer que si son expérience[6] est suffisante et s'il est suffisamment précis dans ses prédictions ;

– le XCSS (spécifié) (Lanzi, 1997) est utilisé pour des environnements qui permettent peu de généralisation ;

6. Paramètre représentant le nombre d'activations du classeur.

– le XCSm (*messy*) (Lanzi *et al.*, 1999a) propose une modification de la partie condition des règles permettant d'avoir une longueur variable ;

– le XCSR (réel) (Wilson, 2000) permet la prise en compte des données réelles ;

– le CXCS (corporation) (Tomlinson *et al.*, 1999a) introduit le principe de corporation dans les XCS ;

– le XCSTS (*tournament selection*) (Butz *et al.*, 2003) propose de répondre au problème de performance dû à la sélection proportionnelle des parents de l'algorithme génétique. Les parents sont sélectionnés en considérant deux sous populations choisies aléatoirement. Les vainqueurs sont ceux qui ont la plus grande *fitness*. Cette modification rend les XCS plus efficaces.

Notons que cette énumération n'est pas exhaustive.

3.3. *Systèmes connexes*

Il existe différentes versions de systèmes de classeurs qui tendent à améliorer le modèle et/ou l'étendue de ses applications. Nous avons distingué les systèmes à *anticipation* et les systèmes *hiérarchiques*.

Anticipation

Les systèmes à anticipation se basent sur l'hypothèse que l'anticipation des conséquences engendrées par une action est primordiale pour définir un comportement. Nous montrons ici les systèmes principaux :

– le ACS (Anticipatory Classifier System) (Stolzmann, 1998) ajoute une partie *effet* pour chaque règle, anticipant les changements du monde causés par la partie action sous la partie condition. La qualité d'un classeur est alors calculée sur la qualité de l'anticipation des *effets* de la règle sur le monde, ainsi le système se crée un modèle du monde permettant de planifier ses actions. Il propose l'utilisation d'heuristiques à la place des algorithmes génétiques pour la construction des règles ;

– le YACS (Yet Another Classifier System) (Gérard, 2002) utilise le même formalisme que le ACS avec des heuristiques différentes permettant d'améliorer la vitesse de l'apprentissage latent ;

– le MACS (Modular Anticipatory Classifier System) (Gérard, 2002) est un formalisme pour l'apprentissage latent qui introduit les anticipations partielles, ses règles n'anticipent pas les valeurs de tous les senseurs en une seule fois. Ce système offre de nouvelles possibilités de généralisation et de performances.

Hiérarchique

Les systèmes hiérarchiques proposent l'utilisation de plusieurs systèmes de classeurs s'échangeant des informations. Nous exposons les deux principaux modèles :

– le ALECSYS (A LEarning Classifier SYStem) (Dorigo, 1995) est une structure hiérarchique permettant de décomposer une tâche complexe en un ensemble de tâches simples. Un système de classeurs joue le rôle d'arbitre dans le choix de l'activation d'un des autres systèmes qui apprennent les comportements de bases ;

– le OCS (Organizational Classifier System) (Takadama *et al.*, 1999) est composé de différents agents possédant chacun un système de classeurs devant satisfaire une tâche commune. Le mécanisme de spécialisation et la communication entre agents permettent d'augmenter l'efficacité du système.

Les systèmes à anticipation et hiérarchiques proposent d'autres systèmes non énumérés ici.

4. Un modèle générique pour une famille de classeurs

Les systèmes que nous avons présentés permettent de trouver des solutions optimales dans des environnements markovien ou non markovien. Néanmoins, comme le fait remarquer Sanza (Sanza, 2001) les améliorations et systèmes connexes sont adaptés à des cas précis et aucun de ces modèles n'a vocation à apporter une solution globale pour tous les problèmes (le XCS n'est efficace que si les rétributions sont discrètes et en nombre fixe, le ACS n'est utile que si chaque action engendre une modification dans la perception du monde...).

Les systèmes de classeurs sont donc multiples. Dans le cadre de nos travaux, il nous est nécessaire de pouvoir développer et tester facilement une grande variété de tels systèmes. L'analyse de la structure et de la dynamique effectuée en section 2 nous permet d'aboutir à un modèle générique support pour une famille de systèmes de classeurs.

Notre architecture se veut générique, dans le sens où elle permet d'implémenter les systèmes de la famille des ZCS et XCS (ZCS, XCS, ZCSM, XCSM). Nous illustrons plus loin son utilisation par différentes expérimentations. Nous décrivons tout d'abord l'architecture ; nous montrons ensuite son utilisation.

4.1. *Architecture*

Notre architecture s'articule autour d'un assemblage de deux composantes, chacune (*interface avec l'environnement* et *système*) étant détaillée par la suite. Elle est représentée en figure 7 sous la forme d'un diagramme de classe UML, auquel nous avons ajouté un système de classeurs de type ZCS par héritage.

Interface avec l'environnement

L'*interface avec l'environnement* détermine les interactions entre le système et l'environnement communes aux différents systèmes de classeurs, suivant la figure 1. Dans notre modèle, les différentes interfaces sont implémentées à l'aide de trois

classes : *CS_II*, *CS_IO* et *CS_R* (respectivement interface d'entrée, interface de sortie et rétribution). La communication entre les interfaces et l'environnement se fait sous la forme de messages, ce qui permet au système de classeurs d'avoir une exécution parallèle à celle de l'environnement (ce qui est nécessaire si on veut pouvoir l'utiliser autant dans des environnements simulés que réels).

Système

La partie *système* définit, de façon réifiée, les éléments et les mécanismes de notre système de classeurs.

Nous considérons les éléments suivants :

– un classeur (*CS_Classifier*) possède une partie condition (*CS_Condition*), une partie action (*CS_Action*) et une partie paramètre (*CS_Parameter*) ;

– les ensembles `[P]`,`[M]`,`[A]` et $[A]_{-1}$ sont des listes de classeurs, de type *CS_ClassifierList* ;

Nous proposons les mécanismes suivants :

– le mécanisme de `Comparaison`, qui permet d'extraire les classeurs dont la condition s'apparie aux informations provenant de l'environnement. Il est intégré dans *CS_ClassifierList* par la méthode *match()* ;

– le mécanisme de `Génération` par *covering*, qui crée des règles suivant le contenu de `[M]` après `Comparaison`. Il est intégré dans *CS_ClassifierList* par la méthode *cover()* qui peut être paramétrée (nombre de # notamment) ;

– le mécanisme général (*CS_System*) représente le fonctionnement sur un cycle défini (méthode *step()*) par le pseudo code figure 3 ;

– le mécanisme de `Sélection` des actions gagnantes (*CS_SelectorAlgo*) qui doit pouvoir être différent suivant l'apprentissage souhaité ;

– le mécanisme de `Rétribution` (*CS_AOCAlgo*) modifiant la partie paramètre des classeurs ;

– l'algorithme génétique de `Génération` (*CS_GeneticAlgo*), où différents opérateurs doivent être précisés tels que le croisement ou la mutation.

4.2. *Utilisation*

Nous pouvons spécialiser notre architecture[7] afin d'obtenir un ZCS (figure 7). Par héritage, nous définissons :

– les `Régles` (*ZCS_Classifier* fils de *CS_Classifier*) ayant un paramètre force (*ZCS_Power* fils de *CS_Parameter*) ;

7. L'implémentation de notre architecture se présente sous la forme d'une API *C++* basée sur *ARéVi* (Harrouet *et al.*, 2002), un atelier de réalité virtuelle fondé sur une approche multi-agents.

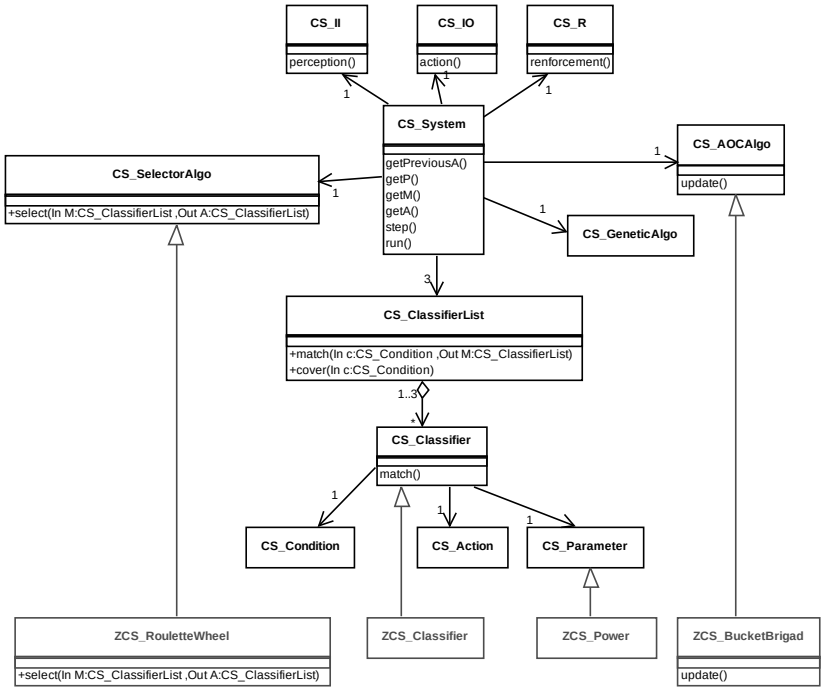


Figure 7. *Diagramme de classe UML de notre architecture, augmenté par un ZCS obtenu par héritage*

– le mécanisme de `Sélection` de type roue de la fortune (*ZCS_RouletteWheel* fils de *CS_SelectorAlgo*) ;

– le mécanisme de `Rétribution` de type *Bucket Brigade* (*ZCS_BucketBrigad* fils de *CS_AOCAlgo*) ;

– l'algorithme génétique de `Génération` (*ZCS_GeneticAlgo* fils de *ZCS_GeneticAlgo*) spécifiant notamment que la valeur sélective est la force de la règle.

Le reste du système utilise les mécanismes par défaut déjà présents, tel que le *covering*. Nous avons mis en place le système de classeurs XCS en utilisant les mêmes techniques. Pour cela, il faut surdéfinir *CS_Parameter*.

Extensions

Nous pouvons également très facilement ajouter de la mémoire au ZCS pour obtenir un ZCSM. Dans ce cas, il faut augmenter le résultat de la perception par un registre

interne au système de classeurs qui est modifié par une partie de la dernière action effectuée. Utilisant notre architecture, il suffit d'hériter de *CS_System* pour redéfinir la méthode *step()* (voir figure 8). Nous avons mis en place le système de classeurs XCSM en utilisant les mêmes principes.

```
t := t +1 ;
// Perception : 1a. codage de la perception
Ie(t) := readDetectors(t) ;
// Perception : 1b. ajout du registre interne
Ie(t) := Ie(t) + Ir(t) ;
...
// Sélection : 3. sélection des règles dans [A]
A(t) := selectClassifiers( M(t),Sélection ) ;
...
// Mémoire : 8. Registre interne
Ir(t+1) := extractPart( getActionPart( A(t) ) ) ;
```

Figure 8. *Pseudo code, représentant la modification de la figure 3, qui met en place un registre interne afin d'ajouter de la mémoire à un système classique*

Systèmes connexes

En utilisant ces mécanismes de spécialisation et d'extension nous avons pu utiliser notre architecture pour implémenter et tester les systèmes de classeurs de la famille des ZCS et XCS (ZCS, ZCSM, XCS, XCSM). Nos perspectives se portent sur l'implémentation de systèmes connexes classiques à anticipation ou hiérarchique (ACS pour l'anticipation et ALECSYS (Dorigo, 1995) pour la hiérarchie). Leurs mises en place pourraient être moins aisées que les systèmes de la famille des ZCS et XCS, et nécessiter des modifications de l'architecture.

4.3. *Validation*

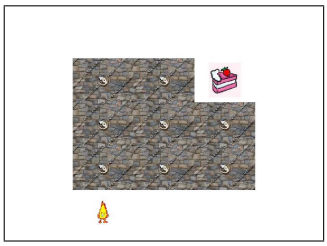
Un environnement d'évaluation simple et fréquemment utilisé est un multiplexeur. Considérons un multiplexeur avec pour entrée 6 bits $a_0, a_1, d_0, d_1, d_2, d_3$ et pour $sortie$ un bit. a_0 et a_1 correspondent aux bits d'adresse. L'équation du multiplexeur est $sortie = \overline{a_0}.\overline{a_1}.d_0 + \overline{a_0}.a_1.d_1 + a_0.\overline{a_1}.d_2 + a_0.a_1.d_3$. La sortie sera la valeur de d_0, d_1, d_2 ou d_3 selon l'adresse. Le but est de retrouver cette fonction de multiplexage.

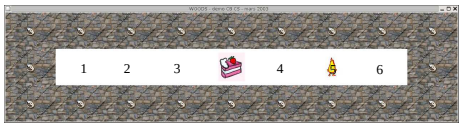
Utilisant notre architecture, il suffit de déterminer les détecteurs et les effecteurs s'interfaçant avec l'environnement, le renforcement à distribuer, et d'instancier un *CS_System* (voir figure 9). Les conditions et les actions des classeurs correspondent ici respectivement aux entrées et aux sorties du multiplexeur. Une récompense est fournie au système de classeurs lorsque la règle sélectionnée correspond à la fonction de multiplexage.

```
/* Environnement */
      env = new EnvironmentMultiplexer(nbEntries,nbOut) ;
/* Relation avec l'environnement */
      detector = new CS_II_Boolean(env) ;
      effector = new CS_IO_Boolean(env) ;
      reinforcement = new CS_R(env) ;
/* Système */
      system = new CS_System() ;
      system->setDetector(detector) ;
      system->setEffector(effector) ;
      system->setReinforcement(reinforcement) ;
/* Activité */
      system->run() ;
```

Figure 9. *Mise en œuvre de notre architecture générique pour un environnement de type multiplexeur*

Un autre environnement d'évaluation intéressant est l'environnement *Woods*. Il correspond à une représentation graphique basée sur un motif qui est répété à l'infini. Il représente une forêt où le but est de trouver de la nourriture. Dans cette forêt il y a des obstacles infranchissables (arbres) et des cases de libre circulation. La perception de l'environnement correspond à une représentation des 8 cases autour de la position occupée. Le plus simple de ces environnements est *Woods1* (voir figure 10a). C'est un environnement déterministe markovien. Le *Woods100* (voir figure 10b) quant à lui est non markovien. En effet, le déplacement optimum de la case numéro 2 est dirigé vers la droite alors qu'il est vers la gauche pour le numéro 5 bien que ces deux cases sont perçues identiquement.


(a)



(b)

Figure 10. *Environnements markovien Woods1 (a) et non markovien Woods100 (b)*

Le système apprend en alternant une itération en mode exploration (sélection de l'action par mécanisme de roue de la fortune) et une iération en mode exploitation (meilleure action choisie). Les courbes ne prennent en compte que les résultats en mode exploitation, en considérant la moyenne des dix dernières itérations.

Notre système converge pour le multiplexeur (figure 11a) et pour *Woods1* (figure 11b). Pour le multiplexeur, nous obtenons 100 % de réussite. En ce qui concerne *Woods1*, nous obtenons des solutions proches du nombre minimum de déplacements. Les résultats sont concluants : dans les deux cas nous obtenons des performances comparables aux résultats décrits par (Wilson, 1994).

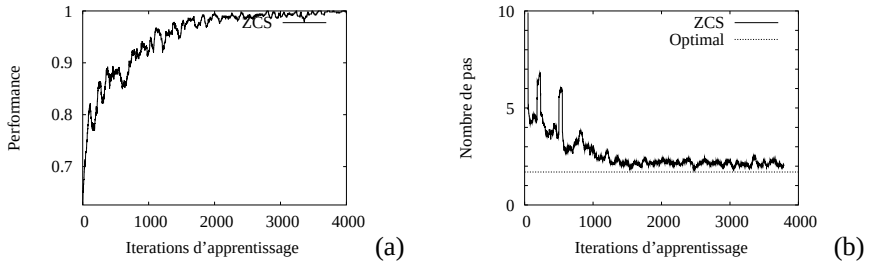


Figure 11. *Apprentissage d'un multiplexeur (a) et dans Woods1 (b) d'un ZCS. A partir d'un nombre suffisant d'itérations, notre système réalise la fonction de multiplexage et obtient le minimum de déplacements dans le cas de Woods1*

De plus, nous avons comparé les résultats du ZCS et du ZCSM dans l'environnement non markovien *Woods100* (figure 12). Nos résultats mettent en évidence les difficultés du ZCS pour obtenir des règles optimales en environnement non markovien. Ils confirment également que notre architecture permet d'étendre les capacités du ZCS à des environnements non markoviens.

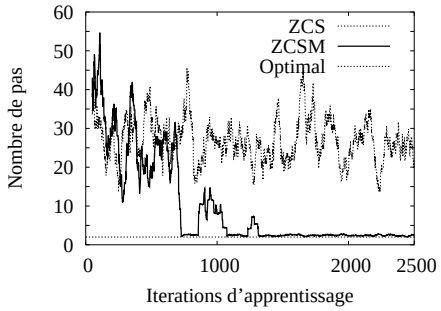


Figure 12. *Apprentissage dans Woods100 d'un ZCS et d'un ZCSM*

5. Exemple d'application : les agents stressés

5.1. *Présentation*

Les agents stressés constituent un algorithme distribué de résolution de problèmes basé sur une métaphore comportementale de groupes d'humains. L'algorithme, détaillé en (De Loor *et al.*, 2003), a été testé sur le problème de résolution d'emplois du temps. C'est une technique de résolution distribuée constituant une heuristique de recherche proche de l'éco-résolution (Ferber, 1995). Une communauté d'agent tente de trouver une solution au problème en envoyant des requêtes, des propositions, des refus ou des annulations. L'émission de ces messages dépend d'une variable appelée *stress* dont l'évolution est elle-même définie par la perception des messages et deux variables : la sensibilité personnelle et la sensibilité collective. Quand le stress passe par des seuils critiques (appelés seuils de crise et seuils de requête), différents mécanismes de décisions sont déclenchés pour modifier partiellement la solution en cours de calcul afin d'orienter la recherche globale. Les seuils de crise et de requête ainsi que les sensibilités personnelles et collectives de chaque agent sont fixés empiriquement. La figure 13 montre leur influence sur les performances de l'algorithme pour un cas donné. Les valeurs optimales sont dépendantes du problème et une étude expérimentale des scénarios de résolution indique que leur modification, durant cette résolution, pourrait améliorer les performances de l'algorithme. Nous avons voulu tester une telle hypothèse grâce aux systèmes de classeurs qui devaient apprendre à modifier ces paramètres, afin d'adapter ceux-ci aux différents problèmes et aux différents stades de résolution d'un problème.

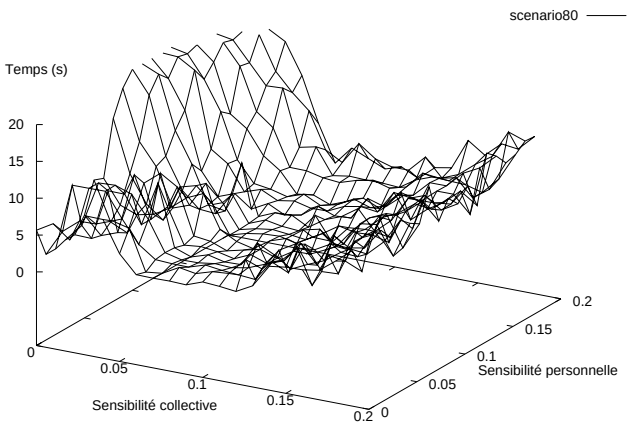


Figure 13. *Influence des paramètres de sensibilité personnelle et collective sur la vitesse moyenne de résolution*

5.2. *Apprentissage*

Pour cet exemple, l'environnement est assurément non markovien et fortement dynamique. En effet, le stress d'un agent synthétise, en une valeur, l'historique de ses interactions. A une valeur de stress donnée peuvent très probablement correspondre de nombreux historiques différents et autant d'emplois du temps possibles. La dynamicité est quant à elle implicitée par l'agentification du problème. Dans le contexte de cette étude, l'aspect le plus intéressant constitue la définition de l'interface d'entrée et de la taille des classeurs. Grâce à l'outil générique, nous avons pu tester différents critères : des *hypothèses sur l'interface d'entrée*, *différents types de renforcement* et *différents types de systèmes de classeurs*.

Hypothèses sur l'interface d'entrée

La difficulté était de rechercher une perception pertinente améliorant l'apprentissage. Les entrées suivantes ont été utilisées successivement :

1) le stress des agents,

2) la dérivée du stress des agents,

3) le nombre de *crises* (passage du stress par le seuil de crise) piquées dernièrement par l'agent.

Profitant des possibilités de notre architecture, plusieurs hypothèses ont été testées par héritage de *CS_II* :

– différentes tailles pour le codage de ces entrées (les informations de l'environnement sont transformées par l'interface d'entrée en une représentation symbolique binaire),

– différentes combinaisons, permettant de prendre en compte une partie ou la totalité des informations perçues.

La solution la plus intéressante a été de prendre en compte les trois entrées, chacune étant codée sur 3 bits (8 seuils).

Différents types de renforcement

Deux types de renforcements ont été identifiés :

1) soit renforcer pendant la résolution d'un problème, en donnant une récompense dépendant de la proximité d'une solution ;

2) soit renforcer lorsque le problème est résolu, en prenant en compte le temps de résolution du problème pour la récompense.

C'est ce dernier type de renforcement que nous avons pris en compte, l'utilisation de renforcement pendant la résolution posant premièrement des difficultés sur le choix des moments où l'on donnera une récompense, et deuxièmement, le renforcement des chemins menant à des solutions partielles ne renforce pas obligatoirement le chemin vers une solution complète.

Différents types de systèmes de classeurs

Nous avons implémenté les systèmes suivants :

– des XCS simples, qui ont eu beaucoup de mal à se stabiliser. En effet, l'environnement est non markovien, et par conséquent provoque l'instabilité du système ;

– des XCS avec mémoire intermédiaires (Lanzi *et al.*, 1999b), destinés aux environnements non markovien, sont apparus décevants. Nous avons identifié le bruit présent au niveau de l'interface d'entrée, consécutif à l'aspect asynchrone et non déterministe de l'interaction entre agents, comme facteur possible débouchant sur ces résultats ;

– des XCS dédiés aux environnements bruités (Lanzi *et al.*, 1999a). Cette solution nous a donné des résultats plus satisfaisants en termes de vitesse d'apprentissage. Cependant, le nombre de paramètres caractérisant un XCS est apparu comme un véritable problème et un inconvénient certain quant à leur utilisation ;

– des ZCS avec ou sans mémoire qui finalement nous ont donné les meilleurs résultats. Notons que ceux-ci sont plus simples et que cette simplicité n'est pas allée à l'encontre de la performance.

Résultats

Les courbes de la figure 14 (a) illustrent l'apprentissage d'un ZCS (en utilisant les paramètres de (Wilson, 1994)) et la figure 14 (b) montre les vitesses d'apprentissage relatives d'un ZCS et d'un XCS sur ce problème. Ainsi nous avons pu tester la souplesse de notre outil en termes d'adaptabilité et sa robustesse dans le cadre d'une utilisation au sein d'un système complexe et ouvert.

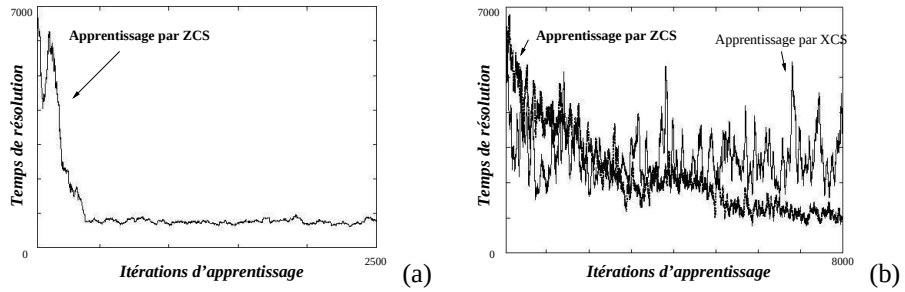


Figure 14. *Résolution du problème des emplois du temps par les agents stressés avec de l'apprentissage par ZCS dans un des agents (a) et comparaison entre l'apprentissage par XCS et par ZCS dans un problème plus complexe (b)*

6. Conclusion et perspectives

Après avoir proposé une description des systèmes de classeurs existants, nous avons décrit un modèle plus général de système de classeurs qui recouvre les systèmes classiques. Nous en avons proposé une implémentation qui permet de modéliser et d'étendre les systèmes classiques ainsi que leurs variantes. Cette implémentation est assez souple pour être utilisée pour différents problèmes puisqu'elle propose une interface entre l'environnement et le système de classeurs (entrées/sorties/renforcement). Elle a pu être utilisée sur différents problèmes et a permis de tester rapidement plusieurs types de systèmes de classeurs, différentes hypothèses conceptuelles, ainsi que d'obtenir des résultats comparatifs intéressants. Ceux-ci nous ont montré, entre autre, l'intérêt de pouvoir accéder à une librairie de systèmes de classeurs qui devrait permettre de définir une méthodologie de choix d'algorithmes d'apprentissage basée sur des étapes de tests.

Actuellement, nous étudions l'utilisation de systèmes de classeurs au sein d'une plate-forme de formation axée sur la mise en situation de l'apprenant et utilisant la réalité virtuelle comme médium. Nous utilisons le modèle *MASCARET* (Buche *et al.*, 2004) qui propose l'utilisation des systèmes multi-agents pour simuler les environnements réalistes, collaboratifs et adaptatifs pour la formation. Un point-clé de cette plate-forme réside dans l'utilisation *d'agents pédagogiques* qui sont des entités virtuelles contribuant à l'apprentissage par le biais de rôles didactiques particuliers (gêneurs, compagnons, conseillers...) en interaction avec l'apprenant. L'idée directrice du travail est de fournir aux formateurs humains un outil leur permettant de spécifier le comportement des agents pédagogiques de sorte qu'ils adaptent leurs interventions par rapport au contexte de la simulation, aux erreurs de l'apprenant et à ses caractéristiques personnelles (aptitudes, cursus...). Le formateur expert d'un domaine mais non informaticien agit de façon intuitive et difficilement formalisable. Nous proposons qu'il « apprenne » aux agents pédagogiques, par le biais de systèmes de classeurs, les réactions et les actions pertinentes permettant d'améliorer les performances du sujet d'apprentissage.

7. Bibliographie

Brooks R., « Elephants Don't Play Chess », *Robotics and Autonomous Systems*, vol. 6, n° 1&2, p. 3-15, June, 1990.

Buche C., Querrec R., De Loor P., Chevaillier P., « MASCARET : A Pedagogical Multi-Agent System for Virtual Environment for Training », *International Journal of Distance Education Technologies (JDET)*, vol. 2, n° 4, p. 41-61, Novembre, 2004.

Butz M., An Implementation of the XCS classifier system in C, Technical report, The Illinois Genetic Algorithms Laboratory, 1999.

Butz M., XCSJava 1.0 : An Implementation of the XCS classifier system in Java, Technical report, Illinois Genetic Algorithms Laboratory, 2000.

Butz M., Pelikan M., « Analyzing the Evolutionary Pressures in XCS », *in* L. Spector, E. D. Goodman, A. Wu, W. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, E. Burke (eds), *Genetic and Evolutionary Computation Conference (GECCO'01)*, Morgan Kaufmann, San Francisco, California, USA, p. 935-942, 7-11 July, 2001.

Butz M., Sastry K., Goldberg D., « Tournament selection : stable fitness pressure in XCS », *in* E. C.-P. et al. (ed.), *Genetic and Evolutionary Computation Conference (GECCO'03)*, Springer, p. 1857-1869, 2003.

Butz M., Wilson W., « An Algorithmic Description of XCS », *Journal of Soft Computing*, vol. 6, n° 3–4, p. 144-153, 2002.

Carver N., Lesser V., The Evolution of Blackboard Control Architectures, Technical Report n° UM-CS-1992-071, Department of Computer Science, University Massachusetts, 1992.

Cavazza M., Charles F., Mead S., « Characters in Search of an Author : AI-Based Virtual Staorytelling », *Virtual Storytelling, International Conference ICVS 2001*, n° 2197 in *LNCS*, Springer, p. 145-154, 2001.

Cliff D., Ross S., Adding Temporary Memory to ZCS, Technical Report n° CSRP347, School of Cognitive and Computing Sciences, University of Sussex, 1995.

De Loor P., Sepseault C., Chevaillier P., « Les émotions : Une métaphore pour la résolution de problèmes dynamiques distribués », *in* J.-P. Briot, K. Ghédira (eds), *Déploiement des systèmes multi-agents, vers un passage à l'échelle, JFSMA'2003, Revue des sciences et technologies de l'information*, vol. hors série, Hermes, p. 331-344, November, 2003. ISBN : 2-7462-0810-5.

Dorigo M., « Alecsys and the AutonoMouse : Learning to Control a Real Robot by Distributed Classifier Systems », *Machine Learning*, vol. 19, p. 209-240, 1995.

Dorigo M., Bersini H., « A Comparison of Q-Learning and Classifier Systems », *From Animals to Animats 3 : Third International Conference on Simulation of Adaptive Behavior (SAB94)*, p. 248-255, 1994.

Ferber J., *Les systemes multi-agents, vers une intelligence collective*, InterEditions, Paris, 1995.

Goldberg D., *Search, Optimization and Machine Learning*, Reading MA Addison Wesley, chapter Genetic Algorithms, 1989.

Gérard P., Systèmes de classeurs : étude de l'apprentissage latent., PhD thesis, Université Paris VI, 2002.

Harrouet F., Tisseau J., Reignier P., Chevaillier P., « oRis : un environnement de simulation interactive multi-agents », *Revue des sciences et technologie de l'information, série Technique et science informatiques (RSTI–TSI)*, vol. 21, n° 4, p. 499-524, 2002.

Heguy O., Sanza C., Berro A., Duthen Y., « GXCS : A generic Classifier System and its application in a Real Time Cooperative Behavior Simulations », *International Symposium and School on Advanced Distribued System (ISADS'02)*, 2002.

Holland J. H., *Adaptation in Natural and Artifcial Systems*, university of Michigan Press, 1975.

Holland J. H., « Properties of the Bucket Brigade Algorithm », *Proc. of the International Conference on Genetic Algorithms and Their Applications*, Pittsburgh, PA, p. 1-7, 1985.

Holland J., Reitman J., « Cognitive systems based on adaptive algorithms », *in* D. A. Waterman, F. Hayes-Roth (eds), *Pattern-directed inference systems*, New York : Academic Press, 1978. Reprinted in : Evolutionary Computation. The Fossil Record. David B. Fogel (Ed.) IEEE Press, 1998. ISBN : 0-7803-3481-7.

Kovacs T., « *Evolving Optimal Populations with XCS Classifier Systems* », Master's thesis, School of Computer Science, University of Birmingham, Birmingham, U.K., 1996.

Kovacs T., XCS Classifier System Reliably Evolves Accurate, Complete, and Minimal Representations for Boolean Functions, Technical Report n° CSRP-97-19, School of Computer Science, University of Birmingham, Birmingham, U.K., 1997.

Lanzi P., « A Study of the Generalization Capabilities of XCS », *in* T. Bäck (ed.), *Proc. of the Seventh Int. Conf. on Genetic Algorithms*, Morgan Kaufmann, San Francisco, CA, p. 418-425, 1997.

Lanzi P. L., Colombetti M., « An Extension to the XCS Classifier System for Stochastic Environments », *in* W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, R. E. Smith (eds), *Genetic and Evolutionary Computation Conference*, vol. 1, Morgan Kaufmann, Orlando, Florida, USA, p. 353-360, 13-17 July, 1999a.

Lanzi P. L., Wilson S. W., Optimal classifier system performance in non-Markov environments, Technical Report n° 99.36, Département d'Electronique et d'Information - Polytechnique de Milan, 1999b.

Maes P., « The dynamics of action selection », *Proceedings of the international Joint Conference on Artificial Intelligence, IJCAI'89*, 1989.

Mateas M., « An Oz-Centric Review of Interactive Drama and Believable Agents », *Lecture Notes in Computer Science*, vol. 1600, p. 297-343, 1999.

Matteucci M., Fuzzy Learning Classifier System : Issues and Architecture, Technical Report n° 99.71, Dipartimento di Elettronica e Informazione - Politecnico di Milano, 1999.

Sanza C., Evolution d'Entités Virtuelles Coopératives par Système de Classifieurs, PhD thesis, Université Paul Sabatier, 2001.

Smith S. F., A Learning System Based on Genetic Adaptive Algorithms, PhD thesis, University of Pittsburgh, 1980.

Stolzmann W., « Anticipatory classifier systems », *Third Annual Genetic Programming Conference*, Morgan Kaufmann, p. 658-664, 1998.

Sutton R. S., Temporal Credit Assignement in Reinforcement Learning, PhD thesis, University of Massachusetts, 1984.

Sutton R. S., « Learning to Predict by the Methods of Temporal Differences », *Machine Learning*, vol. 3, p. 9-44, 1988.

Takadama K., Terano T., Shimohara k., « Can Multiagents Learn in Organization ? – Analyzing Organizational Learning-Oriented Classifier System », *IJCAI'99 Workshop on Agents Learning about, from and other Agents*, 1999.

Tomlinson A., Bull L., « A Corporate XCS », *Proceedings of International Workshop on Learning Classifier Systems*, p. 298-305, 1999a.

Tomlinson A., Bull L., « A Zeroth Level Corporate Classifier System », *in* W. S. P-L Lanzi, S. W. Wilson (eds), *Second International Workshop on Learning Classifier Systems*, Springer, 1999b.

Wilson S., « Generalization in the XCS classifier system », *Genetic Programming 1998 : Proceedings of the Third Annual Conference*, p. 665-674, 1998.

Wilson S. W., « Get Real ! XCS with Continuous-Valued Inputs », *Lecture Notes in Computer Science*, vol. 1813, p. 209-219, 2000.

Wilson W. S., « ZCS : A zeroth level classifier system », *Evolutionary Computation*, vol. 2, n° 1, p. 1-18, 1994.

Wilson W. S., « Classifier Fitness Based on Accuracy », *Evolutionary Computation*, vol. 3, n° 2, p. 149-175, 1995.

Wilson W. S., « Explore/exploit strategies in autonomy », *From Animals to Animats 4 : Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (SAB96)*, p. 325-332, 1996.