THÈSE de DOCTORAT de L'UNIVERSITÉ PARIS 6

Spécialité : **Informatique**

Présentée par : **Pierre GÉRARD**

pour obtenir le grade de
DOCTEUR de L'UNIVERSITÉ PARIS 6

Systèmes de classeurs : étude de l'apprentissage latent

Soutenue le 07 décembre 2002

devant le jury composé de :

Dr. François CHARPILLET	(rapporteur)
Dr. Jean-Arcady MEYER	(directeur de thèse)
Pr. Patrice PERNY	(examinateur)
Dr. Bruno STOUFFLET	(examinateur)
Dr. Stewart WILSON	(rapporteur)
Pr. Jean-Daniel ZUCKER	(examinateur)

Résumé

Dans cette thèse, nous abordons la question de l'anticipation dans les systèmes de classeurs, pour résoudre des problèmes d'apprentissage par renforcement. Ces systèmes permettent d'apprendre incrémentalement des règles définissant la politique comportementale d'un agent, en utilisant la généralisation. Des qu'un système dispose d'un modèle de son environnement, il devient capable d'anticiper les conséquences de ses actions, et d'apprendre une politique plus rapidement. Nous examinons ici comment les systèmes de classeurs peuvent être utilises pour apprendre un tel modèle. En premier lieu, nous reprenons le formalisme du système ACS proposé par Stolzmann en 1998, nous en examinons les limites, puis nous en proposons un nouveau. Le système utilisant ce nouveau formalisme, MACS, utilise une architecture Dyna pour intégrer l'apprentissage d'un modèle de l'environnement et l'apprentissage d'une politique. Nous discutons de l'utilisation d'estimations et d'heuristiques dans les systèmes de classeurs, au lieu d'algorithmes génétiques. Nous concluons sur la possibilité de profiter de capacités d'anticipation sans utiliser des systèmes de classeurs spécifiquement dédies à ce problème.

Mots-clés: Systèmes de classeurs, anticipation, généralisation, apprentissage latent, apprentissage par renforcement

Abstract

In this thesis, we address the problem of anticipation in Learning Classifier Systems, for solving Reinforcement Learning problems. Such systems learn situation-action rules incrementally, by making use of generalization. These rules define the behavioral policy of the agent. When a system has built a model of its environment, it becomes able to anticipate the consequences of its actions, and to improve the learning speed of the policy. Here, we study how Learning Classifier Systems can be used to learn such a model. First, we study the limitations of the formalism of a previous system proposed by Stolzmann in 1998 : ACS. Then, we propose a new formalism and MACS, a new system, to use it. In MACS, we integrate the processes of learning a model and of learning a policy. We discuss the use of estimates and heuristics in Learning Classifier Systems, instead of Genetic Algorithms. We conclude about the possibility of drawing benefits from anticipation without using dedicated systems.

Keywords: Learning Classifier Systems, anticipation, generalization, latent learning, reinforcement learning

Remerciements

Sur le plan professionnel, je remercie en premier lieu Jean-Arcady Meyer pour m'avoir fait confiance en m'acceptant à l'AnimatLab, il y a maintenant quelques temps de cela. Je le remercie aussi pour la qualité de son encadrement, sa disponibilité et l'exigence stimulante dont il fait preuve à chaque instant. Mes remerciements les plus vifs vont aussi à Olivier Sigaud, à qui je dois en grande partie mon intégration à Dassault Aviation. Je le remercie mille fois pour les innombrables discussions passionnantes et enrichissantes que nous avons eues tout au long de cette thèse. Je sais aussi devoir beaucoup à ses qualités humaines, et au rôle qu'il a joué dans les moments de doute, inévitables dans le déroulement d'une thèse.

Je remercie aussi mes rapporteurs – François Charpillet et Stewart Wilson – qui ont accepté de travailler sur ma thèse, ainsi que les autres membres du jury – Patrice Perny qui le préside, Bruno Stoufflet et Jean Daniel Zucker – de porter un intérêt à mon travail, même un samedi matin. Plus particulièrement, je dois beaucoup à Stewart Wilson, dont l'exposé à GECCO2001 m'a fait comprendre beaucoup. Je le remercie aussi pour les multiples pages de courrier électronique que nous avons échangées, et pour le temps qu'il a bien voulu consacrer à la lecture d'une thèse écrite dans une langue qui n'est pas la sienne. J'ai déjà dit remercier Jean-Arcady Meyer pour la confiance qu'il m'a accordée, mais cette thèse n'aurait pas pu non plus avoir été effectuée sans que Bruno Stoufflet ne m'ait accueilli à la Direction Prospectives et Recherches de Dassault Aviation. À ce propos, je n'oublie pas Corinne Fournier.

Je remercie aussi Otsu san, qui m'a invité cet été à l'AIST de Tsukuba, au Japon. Avec lui et avec Kurita san, j'ai pu avoir pendant deux mois des échanges scientifiques remarquables, qui m'ont donné des perspectives de recherche très stimulantes.

Je remercie aussi l'ensemble de la communauté des systèmes de classeurs pour le dynamisme, la passion et la sympathie dont tous font preuve. Je remercie en particulier Martin Butz et Wolfgang Stolzmann, pour tous les échanges que nous avons eus.

Quelque part entre le plan professionnel et le plan amical, je remercie en bloc tous les membres de l'AnimatLab, dont Agnès, Benoît, David, Fab, Gildas, Gouri, JAM, Loïc, Olivier, Steph, Thierry, le Troll et Vinz. D'une part, leur sérieux mêlé de décontraction offre une ambiance de travail remarquable et, d'autre part, les affinités réelles que j'ai avec ces personnes autorisent cette détente salutaire et productive qui accompagne les relations d'amitié. Hors de l'Animat-Lab mais toujours au LIP6, je suis aussi ravi d'avoir pu rencontrer des personnes comme Samuel ou Nico. Côte Dassault Aviation, je cite à nouveau Fab, et je remercie en outre Farid et Fred pour les moments que nous avons pu partager.

D'un point de vue strictement personnel, je remercie en tout premier lieu mes parents Solange et Alain, et pas seulement pour leur attitude durant ces trois ans, mais pour toute une vie. L'affection qu'ils m'ont toujours portée a fait de moi quelqu'un d'assez équilibré pour supporter les déconvenues, doutes et déprimes légères qui jalonnent le parcours de n'importe quel thésard. Je ne les remercierai jamais assez pour la liberté et la confiance qu'ils m'ont toujours accordée, même quand je pouvais faire un peu n'importe quoi. Ma sœur Audrey ne se rend probablement pas compte de l'énergie qu'elle m'apporte à chaque fois que je la revoie, et je la remercie tout simplement d'être. J'ai également une pensée émue pour mes grands-parents, sans le soutien desquels mon parcours n'aurait probablement pas été le même. Je remercie aussi ma marraine Solange, mon parrain Martin, mes cousins comme Laurent, mes cousines comme Muriel, et mes oncles et tantes comme Hilde, Eliane ou Marcel.

Toujours sur le plan personnel, je remercie tout particulièrement Sophie. Son soutien a été vital pour moi, surtout dans les dernières semaines. C'est à elle plus qu'à toute autre chose que je devrai de ne jamais oublier la semaine précédant la soutenance. Parmi ceux avec qui je suis moins intime, je remercie mes vieux amis de Moselle et mes nouveaux amis de région parisienne. Même si je ne les vois pas tous à fréquence égale, je tiens à les remercier tous pour le soutien indéfectible qu'ils m'ont apporté, parfois sans même savoir à quel point. Je cite d'abord les bôs : le vieux Buch, Gub, Nono, Theo, Yann, Sebar et Vincent. Du côté parisien, je pense en particulier à Aurélien, Chris, Didier, Fred, Greg, Gaëlle, Hélène, Francky, Julie, Karim, Laurent, Malika, Patou, Steph et Sylvie.

Le lecteur attentif aura remarqué que dans chaque liste de noms, j'ai pris soin d'utiliser un ordre alphabétique ; c'est qu'il est tout à fait impossible d'en utiliser un autre. Par ailleurs, si certains se sentent oubliés, qu'ils ne ruminent pas de sombres pensées à mon encontre. Qu'ils m'en parlent et je leur dirai s'il s'agit d'un oubli malheureux ou bien d'une volonté délibérée.

Table des matières

Introduction

Pour résoudre un problème, on a parfois besoin d'effectuer plusieurs actions successives. La figure 1 illustre un problème de décision qui peut se poser à un clown de type « Auguste ». Ce problème se résume à une constatation simple : pour entarter le clown blanc alors qu'on n'a pas de tarte à la crème, il faut d'abord aller à la pâtisserie avant d'aller chez le clown blanc. À chaque étape de la résolution du problème, on connaît la situation dans laquelle on se trouve : on a une tarte à la crème ou pas, on se trouve à son domicile ou ailleurs... En fonction de cette situation, on doit être capable de décider quoi faire, pour se rapprocher un peu plus de l'objectif que l'on se fixe, en agissant sur le monde de façon à changer sa situation courante.

Systèmes de classeurs

Une façon de représenter une solution possible au problème posé consiste à définir un ensemble de règles de décision, qui spécifient chacune quelle action entreprendre, en fonction de la situation dans laquelle on se trouve. Un tel ensemble de règles définit une *politique*, c'est-à-dire qu'il spécifie la décision à prendre dans chaque situation.

L'ensemble de ces règles peut être exprimé sous la forme d'un *système de classeurs*, qui présentent l'intérêt de pouvoir apprendre automatiquement les règles qui définissent la politique. Dans ce cas, le système est d'abord confronté au problème réel, sans aucune connaissance préalable : il ne sait pas qu'aller à la pâtisserie permet d'obtenir une tarte à la crème, ni même qu'aller chez le clown blanc avec une tarte à la crème permet d'atteindre l'objectif consistant à entarter le clown blanc. Il apprendra ces faits par l'expérience.

Confronté à une situation, le système choisit une action, en fonction des règles de décision dont il dispose, et la nouvelle situation qui lui est présentée est celle qui résulte de sa dernière action. Par exemple, si la situation indique la possession de tarte à la crème et la présence au domicile, et si la décision est d'aller à la pâtisserie, alors la nouvelle situation est caractérisée par la possession de tarte à la crème et la localisation à la pâtisserie.

Lorsque les actions choisies ne permettent pas de remplir les objectifs assignés, c'est que la politique n'est pas bonne, et que les règles de décision doivent être modifiées. Un système de classeur apprend donc une politique par essais et erreurs, au cours de ses tentatives successives.

On peut alors voir le problème auquel est confronté le système comme un environnement, dans lequel un agent informatique est engagé. L'agent perçoit sa situation par l'intermédiaire de senseurs, et il dispose de diverses possibilités de réponse pour modifier sa situation, conformément

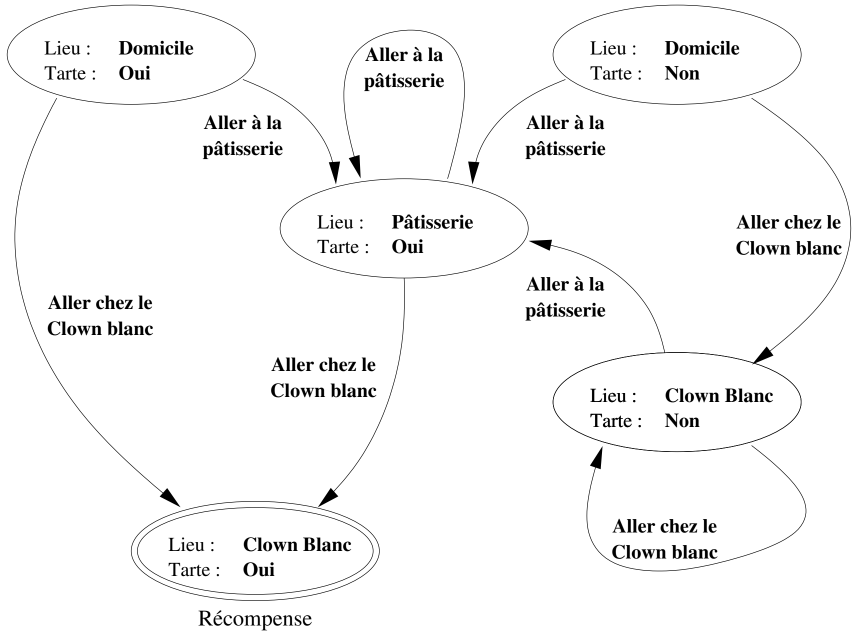

FIG. *1: Ce graphe décrit entièrement un problème de décision dans lequel on est soit à son domicile, soit à la pâtisserie, soit chez le clown blanc. Si on est chez le clown blanc et si on a une tarte à la crème, on peut entarter le clown blanc. Sans tarte à la crème, on ne peut pas entarter le clown blanc. On peut faire deux choses : soit aller chez le clown blanc, soit aller à la pâtisserie. Par souci de simplicité, nous considérons qu'aller à la pâtisserie permet systématiquement d'obtenir une tarte à la crème, et que l'action d'aller chez le clown blanc inclue l'entartage, source de récompense. Les situations possibles sont représentées par les nœuds du graphes. Elles sont identifiées par un lieu et par la possession ou non de tarte à la crème. Les actions permettent de modifier sa situation. Le problème posé est de déterminer les actions adéquates dans chaque situation, de manière à entarter le clown blanc le plus rapidement possible, quelle que soit la situation initiale.*

à la figure 2. Le problème de l'agent reste l'apprentissage d'une bonne politique. Dans l'exemple de la figure 1, l'agent connaît sa situation en percevant le lieu dans lequel il se trouve, et en déterminant s'il a ou non une tarte à la crème. Il peut changer de situation en allant chez le clown blanc ou en allant à la pâtisserie. Ainsi, l'agent perçoit sa situation, agit sur son environnement et perçoit sa nouvelle situation. Un nouveau cycle de la boucle sensori-motrice commence alors.

Dans l'exemple de la figure 1, les quatre classeurs suivants définissent une politique qui permet de remplir l'objectif, c'est-à-dire d'entarter le clown blanc :

si on est à son domicile et si on a une tarte à la crème, aller chez le clown blanc
si on est à la pâtisserie et si on a une tarte à la crème, aller chez le clown blanc
si on est à son domicile et si on n'a pas de tarte à la crème, aller à la pâtisserie
si on est chez le clown blanc et si on n'a pas de tarte à la crème, aller à la pâtisserie

Dans les systèmes de classeurs, les règles de décision sont appelées *classeurs*. Dans le premier classeur de la liste ci-dessus, la partie *« si on est à son domicile et si on a une tarte à la crème »*

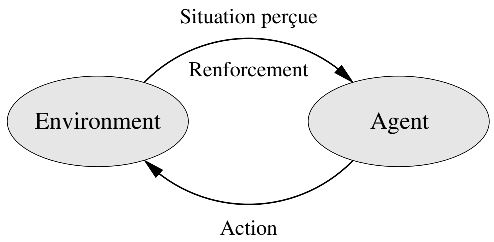

FIG. *2: Boucle sensori-motrice entre un agent est son environnement.*

définit les *conditions* dans lesquelles ce classeur peut s'appliquer. La partie *« aller chez le clown blanc »* est l'*action* proposée par le classeur.

Généralisation dans les systèmes de classeurs

Dans cet exemple, le lieu importe peu : pour peu que l'on ait une tarte à la crème, que l'on soit à son domicile, à la pâtisserie, ou encore en d'autres lieux comme chez la dresseuse, aller chez le clown blanc permet d'atteindre l'objectif d'entarter le clown blanc. Il en va de même qu'il pleuve, qu'il vente ou bien qu'il fasse beau, ou encore que l'on porte un noeud papillon à poids ou bien à rayures. Il n'est pas nécessaire de considérer indépendamment toutes les situations possibles pour prendre une décision. Ainsi, les deux premiers classeurs de la politique ci-dessus peuvent être rassemblés en un seul classeur :

si on a une tarte à la crème, aller chez le clown blanc

De même, les deux derniers peuvent être remplacés par le classeur :

si on n'a pas de tarte à la crème, aller à la pâtisserie

Ces règles utilisent une régularité du problème, qui permet d'ignorer le lieu pour prendre une décision. En ignorant certains aspects d'un problème, on effectue une *généralisation*. Au lieu de construire un classeur pour chacune des situations possibles, on peut alors rassembler les situations dans lesquelles il convient de prendre la même décision. De telles généralisations permettent de réduire le nombre de classeurs et d'améliorer l'intelligibilité du système. En effet, même si le problème était plus complexe et s'il pouvait concerner de nombreux lieux possibles, il suffirait toujours d'une seule règle pour décider d'aller chez le clown blanc si on a une tarte à la crème. De plus, si un système de décision est construit en ne prenant en compte que les lieux et la possession de tarte à la crème, mais que l'on ajoute, par la suite, d'autres aspects aux situations perçues, comme la météo, alors il n'est pas nécessaire de multiplier le nombre de classeurs pour prendre en compte le nouvel aspect : les anciens classeurs restent applicables.

Apprentissage par renforcement

Lorsque l'action choisie permet de résoudre le problème, on le signifie à l'agent en lui donnant un signal de *renforcement*. Un renforcement est un nombre positif ou négatif, qui signifie une punition ou une récompense. Par exemple, lorsque l'agent choisit d'aller chez le clown blanc avec une tarte à la crème, on lui donne une récompense de 1, mais lorsqu'il entreprend la même action sans avoir de tarte à la crème, la récompense est nulle ou négative. Les objectifs de l'agent peuvent être spécifiés par des récompenses.

Après chaque action entreprise, l'agent modifie éventuellement sa politique en créant de nouveaux classeurs, et il perçoit sa nouvelle situation. Un nouveau cycle de la boucle sensori motrice commence, dans lequel l'agent ajuste encore un peu plus sa politique, jusqu'à ce qu'elle soit bonne, après un certain temps. L'apprentissage s'effectue donc par des modifications successives de la solution : il est *incrémental*.

Le signal de renforcement permet à l'agent d'évaluer la qualité d'une solution envisagée, pour corriger ses erreurs. Ainsi, lorsque le système crée une règle qui ne lui permet pas de remplir ses objectifs, on finira par lui en préférer une autre, identifiée comme plus adéquate. Par exemple, dans le problème de la figure 1, les classeurs suivants sont conservés parce qu'ils permettent de résoudre le problème :

si on a une tarte à la crème, aller chez le clown blanc
si on n'a pas de tarte à la crème, aller à la pâtisserie

En revanche, un classeur comme *« si on n'a pas de tarte à la crème, aller chez le clown blanc »* ne permet pas d'atteindre l'objectif, et il finira par être écarté.

Lorsque l'agent entreprend une action et qu'il se trompe, il n'y a aucun superviseur pour lui indiquer quelle action il aurait dû entreprendre : le seul retour sur la qualité de la solution envisagée est la récompense. L'apprentissage est donc incrémental, et il n'est pas supervisé directement.

L'agent cherche à maximiser sa récompense, mais celle-ci peut n'intervenir qu'après plusieurs actions successives. Par exemple, quand l'agent choisit d'aller à la pâtisserie et qu'il n'a pas de tarte à la crème, c'est une bonne chose, mais la récompense n'intervient que plus tard, une fois qu'il est chez le clown blanc. Ainsi, l'agent doit apprendre une politique grâce à un signal de renforcement qui n'est pas délivré après chaque action.

Dans ces conditions, l'agent doit apprendre comment obtenir la récompense la plus importante possible, en un minimum de temps. Pour ce faire, il doit déterminer la meilleure action dans chacune des situations possibles, y compris celles qui n'autorisent pas une résolution immédiate du problème. Par exemple, le classeur suggérant que *« si on n'a pas de tarte à la crème, aller à la pâtisserie »* ne sera pas conservé parce qu'il permet de remplir directement les objectifs, mais plutôt parce qu'il permet à l'agent de se trouver dans une situation favorable à une résolution rapide du problème. C'est pourquoi il sera préféré au classeur *« si on n'a pas de tarte à la crème, aller chez le clown blanc »*. En effet, l'application de ce classeur n'apporterait pas non plus de renforcement, mais son application nécessiterait d'effectuer deux actions supplémentaires pour remplir l'objectif. Ces deux actions sont successivement *« aller à la pâtisserie »* pour obtenir une tarte à la crème, puis *« aller chez le clown blanc »*, pour enfin recevoir une récompense. En revanche, utiliser le classeur qui suggère que *« si on n'a pas de tarte à la crème, aller à la pâtisserie »* met l'agent dans une situation où il a une tarte à la crème, et où il n'a besoin que d'une seule action supplémentaire pour être récompensé, au lieu de deux. Toutes choses étant égales concernant l'action à effectuer une fois qu'il a une tarte à la crème, sur un certain nombre d'essais, l'agent constate que, lorsqu'il n'y a pas de tarte à la crème, aller à la pâtisserie permet de recevoir une récompense plus rapidement. Le premier classeur est donc conservé au détriment du second.

Anticipation

Pour apprendre l'action adéquate dans des situations ne menant à aucune récompense immédiate, il s'avère nécessaire de procéder à un certain nombre de tentatives pour apprendre quelle action choisir. En effet, l'agent n'a pas de modèle de son environnement qui lui permettrait de savoir que, lorsqu'il n'a pas de tarte à la crème, il peut en obtenir à la pâtisserie. Il ne connaît pas non plus l'action lui permettant de se trouver chez le clown blanc, juste après avoir été à la pâtisserie. S'il disposait de ces informations, et s'il n'avait pas de tarte à la crème, il pourrait, de son domicile, décider directement d'aller à la pâtisserie avant d'aller chez le clown blanc, sans procéder à aucun essai infructueux.

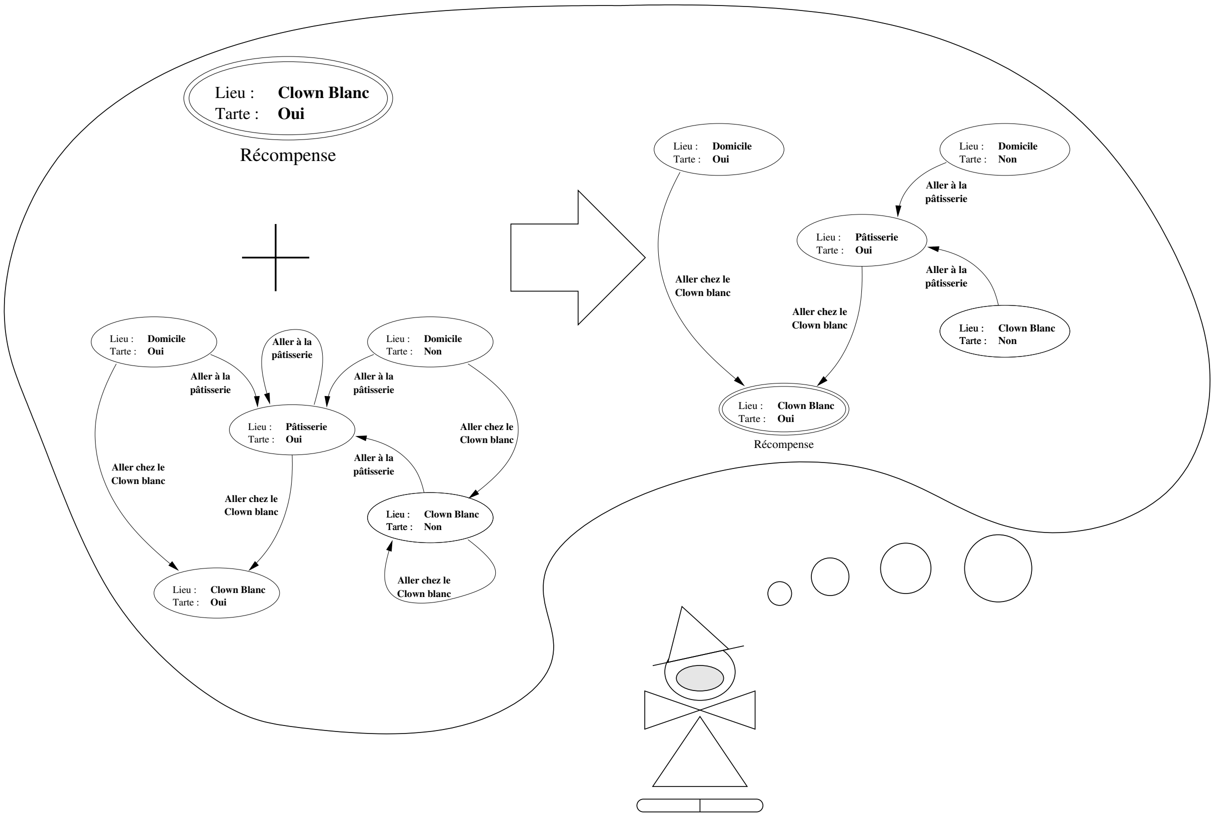

FIG. *3: Utilisation d'un modèle pour construire une politique.*

Avoir un modèle de l'environnement permet à l'agent d'*anticiper* les conséquences de ses actions, c'est-à-dire de prédire la nouvelle situation résultant d'une certaine action dans une situation particulière. Puisque les situations successives présentées à l'agent dépendent à chaque fois des actions entreprises, il est possible d'apprendre un tel modèle. Cet apprentissage est indépendant de la récompense : c'est un *apprentissage latent*. Pour ce faire, il suffit d'enregistrer la liste des conséquences de chaque action dans chacune des situations possibles. Par exemple, l'environnement de la figure 1 peut être représenté par l'ensemble de règles :

si on a une tarte à la crème, si on est au domicile et si on va chez le clown blanc, alors on aura une tarte à la crème et on sera chez le clown blanc
si on a une tarte à la crème, si on est au domicile et si on va à la pâtisserie, alors on aura une tarte à la crème et on sera à la pâtisserie
si on a une tarte à la crème, si on est à la pâtisserie et si on va chez le clown blanc, alors on aura une tarte à la crème et on sera chez le clown blanc
si on a une tarte à la crème, si on est à la pâtisserie et si on va à la pâtisserie, alors on aura une tarte à la crème et on sera à la pâtisserie
si on n'a pas de tarte à la crème, si on est au domicile et si on va chez le clown blanc, alors on n'aura pas de tarte à la crème et on sera chez le clown blanc
si on n'a pas de tarte à la crème, si on est au domicile et si on va à la pâtisserie, alors on aura une tarte à la crème et on sera à la pâtisserie
si on n'a pas de tarte à la crème, si on est chez le clown blanc et si on va chez le clown blanc, alors on n'aura pas de tarte à la crème et on sera chez le clown blanc
si on n'a pas de tarte à la crème, si on est chez le clown blanc et si on va à la pâtisserie, alors on aura une tarte à la crème et on sera à la pâtisserie

Dans la première règle, *« on aura une tarte à la crème et on sera chez le clown blanc »* est l'*effet* de l'action *« on va chez le clown blanc »*, à condition d'être dans la situation *« on a une tarte à la crème et on est au domicile »*. Dans ces règles, il y a donc trois parties : une pour la condition, une pour l'action et une pour l'effet de l'action. Si, par ailleurs, on sait que *« si on a une tarte à la crème et si on est chez le clown blanc, alors on sera récompensé »*, on peut en déduire immédiatement les actions à entreprendre dans chaque situation pour obtenir rapidement une récompense, comme illustré par la figure 3. Par exemple, on peut anticiper que, si on est au domicile sans tarte à la crème, il vaut mieux passer d'abord à la pâtisserie avant d'aller chez le clown blanc. Si on allait d'abord chez le clown blanc, il faudrait de toutes façons revenir à la pâtisserie pour obtenir une tarte à la crème.

Dans ce cas l'apprentissage par renforcement n'est pas direct, puisqu'il ne cherche pas d'abord à identifier les action optimales dans chaque situation. Il commence par la construction d'un modèle sous la forme d'anticipations.

Modèle de l'environnement et généralisation

En vue de la construction d'un modèle, on peut aussi mettre en œuvre un processus de généralisation comme celui évoqué précédemment, et rendre plus intelligible l'ensemble des classeurs. Par exemple, on peut construire l'ensemble de classeurs suivants :

si on a une tarte à la crème, où que l'on soit et si on va chez le clown blanc, alors on aura toujours une tarte à la crème et on sera chez le clown blanc
si on n'a pas de tarte à la crème, où que l'on soit et si on va chez le clown blanc, alors on n'aura toujours pas de tarte à la crème mais on sera chez le clown blanc[1]
qu'on ait une tarte à la crème ou non, où que l'on soit et si on va à la pâtisserie, alors on aura une tarte à la crème et on sera à la pâtisserie

Dans ce cas, les effets rendent compte de ce qui change en effectuant l'action, et ils proposent chacun une information suffisante à l'anticipation d'une situation. On pourrait aussi produire l'ensemble de classeurs suivants :

quelle que soit la situation, si on va à la pâtisserie, alors on sera à la pâtisserie
quelle que soit la situation, si on va chez le clown blanc, alors on sera chez le clown blanc
quelle que soit la situation, si on va à la pâtisserie, alors on aura une tarte à la crème
si on a une tarte à la crème, où que l'on soit et si on va chez le clown blanc, alors on aura une tarte à la crème
si on n'a pas de tarte à la crème, où que l'on soit et si on va chez le clown blanc, alors on n'aura pas de tarte à la crème

Ici, les effets ne renseignent que sur un seul aspect de la nouvelle situation. Il s'ensuit que, pour anticiper les conséquences de l'action *« aller à la pâtisserie »* dans la situation *« on a une tarte à la crème et on est à la pâtisserie »*, il faut avoir recours aux deux premiers classeurs. Le premier donne le nouveau lieu, et le second indique si on aura une tarte à la crème. Le nombre de classeurs est plus important ici, mais le système reste intelligible : il est possible de séparer les classeurs anticipant partiellement des aspects différents des situations, de manière à proposer plusieurs groupes de classeurs, chacun très simple.

Problématique

Dans les pages qui précèdent, nous avons introduit l'apprentissage par renforcement : c'est un apprentissage non supervisé, incrémental, et qui repose sur un signal de renforcement éventuellement retardé. Nous avons également introduit la méthode de généralisation dans les systèmes de classeurs. Nous avons indiqué comment les mécanismes correspondant peuvent être utilisés dans la construction d'un modèle de l'environnement. Nous avons enfin montré comment il est possible d'utiliser le modèle pour anticiper les conséquences de ses actions, de manière à diminuer le nombre d'essais infructueux avant de trouver une bonne politique.

Cette thèse – centrée sur les systèmes de classeurs anticipatifs – est organisée en deux parties. Dans la première, nous étudierons comment il est possible de combiner l'apprentissage d'un modèle de l'environnement et des mécanismes de généralisation. Dans la deuxième, nous étudierons comment il est possible de combiner anticipation et apprentissage par renforcement.

[1] Cette règle est vraie parce qu'il est impossible d'être à la pâtisserie sans tarte à la crème.

La question de l'apprentissage d'un modèle de l'environnement a déjà été étudiée par Sutton (1988). Son système, *DynaQ+* apprend un modèle de l'environnement indépendamment de la récompense. Ce système utilise une architecture, appelée « *Dyna* », qui permet de séparer les processus d'apprentissage par renforcement de ceux d'apprentissage latent du modèle de l'environnement. Ce système est détaillé dans le chapitre 1. Sutton montre que l'apprentissage d'une politique est beaucoup plus rapide lorsque l'on utilise les capacités d'anticipation offertes par le modèle. Toutefois, *DynaQ+* n'utilise la généralisation ni pour l'apprentissage latent, ni pour l'apprentissage par renforcement : l'apprentissage latent construit des règles comme :

> *si on a une tarte à la crème, si on est au domicile et si on va chez le clown blanc, alors on aura une tarte à la crème et on sera chez le clown blanc*

et l'apprentissage par renforcement utilise l'algorithme *Q-learning* (Watkins, 1989), également décrit au chapitre 1, qui doit considérer toutes les situations et actions possibles, de manière exhaustive.

Nous trouvons donc dans la littérature d'une part, un système efficace profitant de l'anticipation et, d'autre part, des systèmes de classeurs traditionnels comme XCS (Wilson, 1995), qui permettent de généraliser, mais sans utiliser l'anticipation. Une des contributions de ce travail porte sur la manière dont les systèmes de classeurs peuvent utiliser la généralisation dans l'apprentissage latent d'un modèle de l'environnement. Ce problème a déjà été abordé par Stolzmann (1998) avec ACS, voir chapitre 2, un système qui utilise des classeurs à trois parties, comme par exemple :

> *si on a une tarte à la crème, où que l'on soit, si on va chez le clown blanc, alors on aura toujours une tarte à la crème et on sera chez le clown blanc*

Chacun des classeurs utilisés par ACS anticipe donc tous les aspect des situations (le lieu, la possession de tarte à la crème...). Dans cette thèse, nous proposons un premier système, YACS[2], qui utilise le même type de classeurs, mais des mécanismes différents pour leur création. Nous montrons que ces nouveaux mécanismes sont plus efficaces, tout en insistant sur le fait que, chercher à anticiper simultanément tous les aspect des situations rend difficile l'exploitation d'un certain nombre de régularités pour la généralisation.

Nous proposons alors un nouveau formalisme, et un nouveau système de classeurs pour l'apprentissage latent, MACS[3], qui utilise des classeurs de type :

> *quelle que soit la situation, si on va à la pâtisserie, alors on sera à la pâtisserie*

À la différence de YACS, les classeurs de MACS ne proposent donc que des anticipations partielles, qui ne prédisent qu'un seul aspect des situations. Nous montrerons plus loin comment ces anticipations partielles permettent de reconsidérer l'apprentissage latent dans les systèmes de classeurs. En particulier, nous montrons comment il est possible d'utiliser, pour l'apprentissage d'un modèle de l'environnement, des systèmes de classeurs pourtant non spécifiques à ce problème, comme le sont ACS, YACS et MACS, et nous discutons leurs avantages et limitations.

---

[2] Yet Another Classifier System

[3] Modular Anticipatory Classifier System

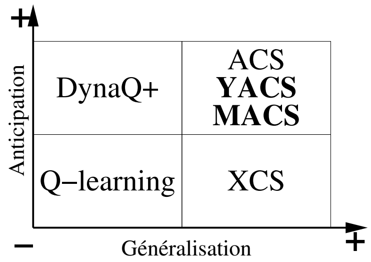

FIG. 4: *Positionnement de YACS et de MACS par rapport à d'autres systèmes, selon les seuls critères de la généralisation et de l'anticipation.*

Une deuxième contribution de ce travail porte sur l'utilisation du modèle de l'environnement pour appuyer l'apprentissage d'une politique. Pour ce faire, nous utilisons une architecture de type *Dyna* pour construire plusieurs politiques. Certaines d'entre elles permettent de choisir des actions pour accélérer l'apprentissage du modèle, et une autre permet d'atteindre les objectifs spécifiés par les récompenses.

Plan du mémoire

Dans le premier chapitre, nous revenons sur l'apprentissage par renforcement. Nous présentons les théories psychologiques qui ont inspiré l'apprentissage par renforcement, et nous décrivons les processus de décision markoviens comme cadre formel. Nous décrivons également l'architecture *Dyna* (Sutton, 1988), qui propose d'apprendre le modèle d'un environnement, indépendamment des récompenses, et qui utilise ce modèle par ailleurs, pour accélérer l'apprentissage d'une politique.

Dans le second chapitre, nous détaillons les systèmes de classeurs dévolus à l'apprentissage par renforcement direct, c'est-à-dire sans l'utilisation d'un modèle de l'environnement pour anticiper. Nous introduisons aussi ACS (Stolzmann, 1998), un système de classeurs permettant de tirer parti de la généralisation dans la construction d'un modèle de l'environnement.

Dans le troisième chapitre, nous décrivons YACS, notre premier système de classeurs dévolu à l'anticipation. Il reprend le formalisme d'ACS et permet de représenter le même type de régularités. Par rapport à ACS, son originalité réside dans les mécanismes utilisés pour construire incrémentalement l'ensemble des classeurs.

Dans le quatrième chapitre, nous identifions une classe de régularités qui ne peut être exploitée ni par ACS, ni par YACS. Nous décrivons un nouveau formalisme pour permettre des anticipations partielles. Nous proposons un nouveau système, MACS, pour utiliser ce formalisme dans la construction d'un modèle de l'environnement, sans qu'il soit encore question d'apprentissage d'une politique.

Dans le cinquième chapitre, nous montrons comment MACS s'intègre dans une architecture de type *Dyna*, de manière à utiliser l'apprentissage du modèle de l'environnement pour accélérer l'apprentissage d'une politique.

Dans le sixième chapitre, nous discutons les principales limitations de MACS et de YACS, qui sont liées à la prise en compte de problèmes dans lesquels les conséquences des actions sont incertaines, c'est-à-dire qu'elles fluctuent pour des actions et des situations pourtant identiques.

Dans le septième et dernier chapitre, nous revenons sur l'approche originale de MACS et sur le problème de l'anticipation dans les systèmes de classeurs. Cette approche est fondée sur la construction de plusieurs modèles indépendants, un pour anticiper chaque aspect des situations. Nous assimilons le problème de l'anticipation à un problème d'approximation de fonctions. Nous proposons alors des solutions pour la prise en compte de situations continues, et pour l'unification des systèmes de classeurs avec anticipation et des systèmes qui réalisent apprentissage par renforcement direct. Nous discutons alors de l'utilisation possible de statistiques descriptives dans les systèmes de classeurs.

Chapitre 1

Apprentissage par renforcement et apprentissage latent

Dans ce chapitre, nous présentons les théories psychologiques qui ont inspiré l'apprentissage par renforcement, et nous indiquons comment l'étude de l'apprentissage animal conduit à considérer un apprentissage indépendant de la récompense : un apprentissage latent. Apprendre un modèle de la dynamique de ses interactions entre un agent et son environnement peut se faire de cette manière.

En premier lieu, nous présentons le conditionnement classique, le conditionnement opérant et l'apprentissage latent chez l'animal. Nous introduisons alors l'apprentissage par renforcement dans le cadre informatique, et nous indiquons dans quelle mesure les modèles développés en psychologie ont été utilisés pour concevoir des agents adaptatifs. Enfin, nous présentons une architecture logicielle permettant d'accélérer l'apprentissage par renforcement d'un agent grâce à un apprentissage latent du modèle de l'environnement.

1.1 L'apprentissage animal

1.1.1 Le conditionnement classique de Pavlov

En psychologie animale, le début du vingtième siècle a été dominé par les théories béhavioristes. Les deux paradigmes prédominants et complémentaires de cette approche sont le conditionnement classique et le conditionnement opérant.

Le conditionnement classique a été introduit par Pavlov (1927). Il modélise le processus d'apprentissage des associations entre stimuli, de manière à modifier leur caractère attractif ou répulsif. On le nomme souvent conditionnement pavlovien. Il rend compte de la manière dont les animaux apprennent à associer des stimuli conditionnels et des stimuli inconditionnels. Par exemple, dans l'expérience célèbre du chien de Pavlov illustrée par la figure 1.1, le stimulus inconditionnel (l'odeur de la nourriture) entraîne un réflexe inconditionnel automatique (la salivation). Le chien est ensuite sujet à un certain nombre d'expériences durant lesquelles on fait

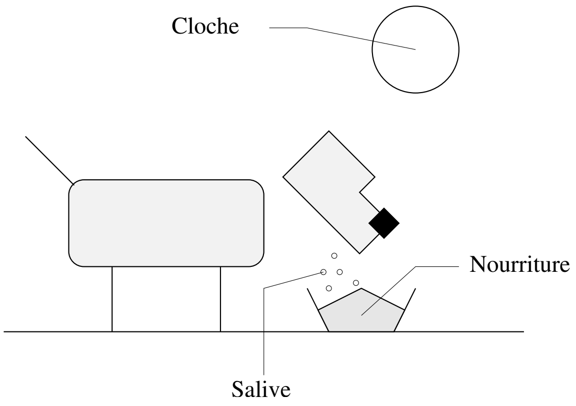

FIG. *1.1: Expérience de conditionnement classique pavlovien.*

immédiatement suivre la présentation d'un stimulus neutre (le son d'une cloche) par la présentation du stimulus inconditionnel. Le stimulus neutre ne provoque initialement aucune réaction particulière de la part du chien. Après la répétition de telles présentations, le chien commence à saliver dès qu'il perçoit le stimulus neutre, sans qu'il ne soit plus nécessaire de présenter le stimulus inconditionnel. Le stimulus neutre a été associé au stimulus inconditionnel pour devenir attractif.

1.1.2 Le conditionnement opérant de Skinner

Le conditionnement classique s'intéresse à la formation d'associations entre plusieurs stimuli. Il ne prend donc pas en compte les actions de l'animal. Le conditionnement opérant (Skinner, 1938), au contraire, concerne l'apprentissage de contingences entre actions et stimuli.

On distingue les contingences à deux termes et les contingences à trois termes. Les contingences à deux termes associent une action de l'animal à sa conséquence. Cette conséquence est un stimulus attractif ou répulsif. Lorsqu'une contingence associant une réponse à un tel stimulus n'est vérifiée qu'en présence d'un stimulus particulier, ce dernier est ajouté à la contingence pour former une contingence à trois termes. Le stimulus en question est dit « discriminant », il permet à l'animal de déterminer dans quelles situations l'action entreprise mène effectivement à la satisfaction considérée. On appelle « différenciation» le fait d'ajouter un stimulus discriminant à une contingence.

La figure 1.2 montre un dispositif classique permettant de conditionner un animal. Il s'agit d'une boîte de Skinner. Dans cette expérience, la boîte contient un pigeon et un levier délivrant de la nourriture lorsqu'il est pressé. Au début de l'expérience, le pigeon n'a aucune tendance particulière à presser le levier. Lorsqu'il agit ainsi, et il crée une contingence entre cette action

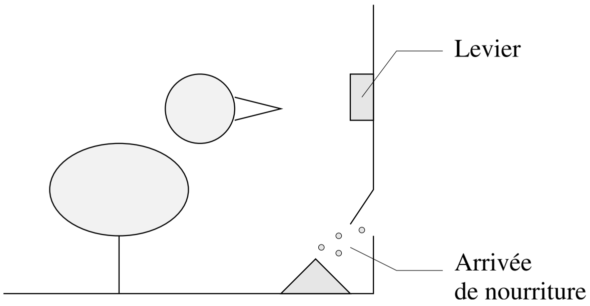

FIG. *1.2: Expérience de conditionnement opérant.*

et l'apparition de nourriture. Cette contingence est renforcée et le pigeon a une tendance de plus en plus nette à presser le levier pour obtenir de la nourriture. La contingence apprise ici est une contingence à deux termes. Pour que le pigeon apprenne une contingence à trois termes, on peut imaginer un dispositif dans lequel la nourriture n'est délivrée que lorsqu'une lampe est allumée. Dans ce cas, le pigeon apprend à presser le levier pour obtenir de la nourriture à condition que la lampe soit allumée. L'état de la lampe forme le troisième terme de la nouvelle contingence apprise.

1.1.3 L'apprentissage latent de Tolman

Les théories du conditionnement ne prennent en compte la formation d'associations ou de contingences entre actions et stimuli que dans la mesure ou une satisfaction est en jeu. Dans le conditionnement classique, un stimulus neutre est seulement associé à un autre stimulus si ce dernier a un caractère attractif ou répulsif. De même, dans le conditionnement opérant, une contingence n'est formée que si elle donne lieu à une récompense ou à une punition.

Pourtant, Tolman et Honzik (1930) montrent que l'animal dispose de capacités d'apprentissage latent, c'est-à-dire qu'il réalise un apprentissage qui aurait lieu indépendamment de toute récompense, en rupture avec les théories béhavioristes de Skinner. Ce type d'apprentissage formerait des associations permettant de prédire les conséquences des actions, même si ces conséquence ne sont associées à aucune récompense ou punition. Ces prédictions forment un modèle de la dynamique des interactions entre l'animal et son environnement.

Une expérience de Tolman et Honzik sur l'apprentissage latent est illustrée par la figure 1.3. Cette expérience fait intervenir un rat dans un labyrinthe dont les blocs A et B sont amovibles par l'expérimentateur. Si aucun bloc n'est présent, le plus court moyen pour le rat de rejoindre la nourriture à partir de son point de départ est d'emprunter le chemin 1 en ligne droite. Si le bloc A est présent, le chemin le plus court vers la nourriture est le chemin 2. Si le bloc B est présent, alors le chemin le plus court est le chemin 3. Au début de l'expérience, aucun bloc n'est positionné et on laisse le rat libre de ses mouvements dans le labyrinthe.

Suivant une procédure de conditionnement opérant, le rat apprend à rejoindre la nourriture par le chemin 1. Selon les théories béhavioristes, l'animal renforce le chemin 1 plus que le chemin

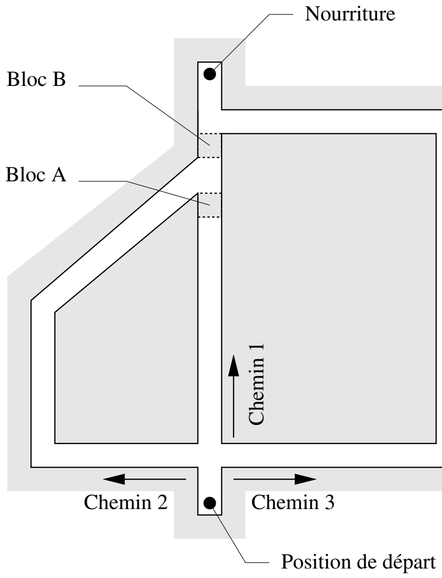

FIG. *1.3: Expérience de Tolman concernant l'apprentissage latent.*

2, et celui-ci plus que le chemin 3. Dès que le comportement consistant à utiliser le chemin 1 a été appris, l'expérimentateur obstrue celui-ci en positionnant le bloc A. Conformément aux théories du conditionnement opérant, le rat rejoint désormais la nourriture en empruntant le chemin 2, deuxième dans l'ordre des renforcements associés aux chemins. Par contre, lorsque le chemin direct 1 est bloqué en B au lieu de A, le rat choisit directement le chemin 3. Pourtant, selon les théories béhavioristes, le rat aurait d'abord dû emprunter le chemin 2 et ce n'est qu'après l'avoir essayé qu'il aurait dû emprunter le chemin 3.

Pour interpréter les résultats d'une telle expérience, on est nécessairement conduit à supposer que, pour emprunter directement le chemin 3, le rat utilise un modèle de son environnement, dont il se sert pour déduire, sans avoir à emprunter le chemin 2, que le bloc B obstrue à la fois le chemin 1 et le chemin 2. Ainsi, durant ses premiers essais dans le labyrinthe, en plus d'apprendre à emprunter le chemin direct pour rejoindre la nourriture, le rat forme un modèle de son environnement. L'apprentissage d'un tel modèle est indépendant de la récompense, il est dit « latent ». Le rat utilise ensuite le modèle appris pour anticiper les conséquences de ses actions. En particulier, grâce au modèle, il peut anticiper qu'emprunter le chemin 2 ne mènera à rien alors que le chemin 3 lui permettra de rejoindre la nourriture.

Seward (1949) propose une autre expérience confortant l'hypothèse d'un tel apprentissage latent. Elle est illustrée par la figure 1.4. Durant la première partie de cette expérience, un rat est laissé libre de ses mouvements dans un labyrinthe en forme de T dont une des extrémités est

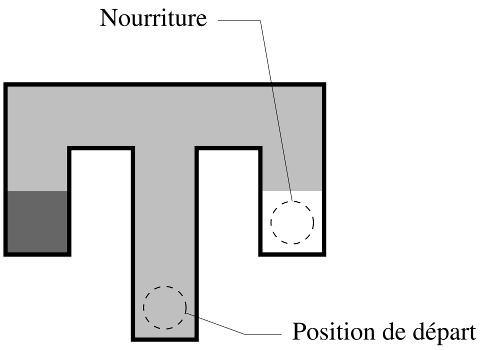

FIG. *1.4: Expérience de Seward concernant l'apprentissage latent.*

peinte en blanc, et l'autre en noir. Selon les théories béhavioristes, aucun apprentissage ne devrait avoir lieu dans ces conditions puisqu'aucune source de renforcement n'est disponible. Le rat est ensuite retiré du labyrinthe et il est soumis à un conditionnement classique par la présentation répétée de nourriture sur une case noire. Quand le rat est à nouveau positionné à l'entrée du labyrinthe, il se dirige directement vers la case noire.

Pour expliquer ce résultat, il faut à nouveau admettre que le rat a formé un modèle de son environnement. Grâce à ce modèle, il est capable d'anticiper les conséquences de ses actions, de manière à choisir directement celles qui le mènent à la nourriture. Ce modèle a été appris alors qu'il n'y avait aucune source de renforcement dans le labyrinthe. C'est donc un apprentissage latent. Tolman propose que ce modèle soit constitué de prédictions explicites :

> [...] *a condition in the organism which is equivalent to what in ordinary parlance we call a « belief », a readiness or disposition, to the effect that an instance of this sort of stimulus situation, if reacted to by an instance of that sort of response, will lead to an instance of that sort of further stimulus situation, or else, simply by itself be accompanied, or followed, by an instance of that sort of stimulus situation.* (Tolman, 1959)

Cette idée est reprise par Hoffmann (1993) qui propose une théorie du Contrôle Anticipatif du Comportement[4]. Cette théorie stipule que l'animal forme constamment une prédiction des conséquences de ses actions. Ces prédictions sont sans cesse révisées en fonction de ce qui advient réellement. Cette confrontation des prédictions avec l'expérience permet à l'animal d'améliorer son modèle de l'environnement en le corrigeant dès qu'une prédiction n'est pas vérifiée. Grâce à ce modèle, il est capable d'anticiper les conséquences de ses actions. Le contrôle du comportement est affecté par cette capacité d'anticipation conformément aux expériences décrites plus haut :

- dans l'expérience de Tolman et Honzik, l'anticipation permet à l'animal d'éviter d'emprunter le chemin 2 juste après que le chemin 1 ait été obstrué en B.
- dans l'expérience de Seward, elle permet de rejoindre directement la nourriture.

La théorie de Hoffmann a été utilisée par Stolzmann (1998) pour concevoir ACS, un système

[4] Anticipatory Behavioral Control

de classeurs qui apprend des règles `Condition Action Effet`. Grâce à leur partie `Effet`, ces règles spécifient les conséquences d'une action sous certaines conditions. L'ensemble des règles produites par l'apprentissage permet à l'agent d'anticiper dans n'importe quelle situation. Grâce à cette capacité d'anticipation, le système devient capable de déterminer à l'avance la séquence d'actions qui lui permettra d'atteindre son but. Dans cette thèse, nous proposons des systèmes de classeurs alternatifs à ACS. Dans les chapitres 3 et 4, nous proposons de nouveaux mécanismes d'apprentissage latent et dans le chapitre 5, nous montrons comment exploiter le modèle de l'environnement pour décider de l'action.

1.2 L'apprentissage par renforcement

1.2.1 Le modèle de Rescorla-Wagner

L'apprentissage par renforcement dans le cadre informatique est inspiré du modèle de Rescorla et Wagner (1972) en psychologie. Ce modèle psychologique rend compte de l'apprentissage des association dans le conditionnement classique. Il décrit comment et à quelle vitesse les stimuli conditionnels sont renforcés.

Dans ce modèle, une valeur $V(sc)$ est associée à chaque stimulus conditionnel sc. Dans une expérience classique de conditionnement pavlovien, on procède à plusieurs essais successifs au cours desquels la perception d'un ou plusieurs stimuli conditionnels est immédiatement suivie de la présentation d'un stimulus inconditionnel (comme la présentation de nourriture). Le stimulus inconditionnel fait office de récompense ou de punition. L'intensité du stimulus inconditionnel est notée $R(si)$.

D'après le modèle de Rescorla-Wagner, à chaque essai, la valeur associative des n stimuli conditionnels présents est mise à jour par l'équation suivante, dite équation de Rescorla-Wagner :

$$V(sc_i) \leftarrow V(sc_i) + \beta \left(R(si) - \sum_{i=1}^{n} V(sc_i) \right) \tag{1.1}$$

Le paramètre β représente la vitesse d'apprentissage, qui peut varier selon les espèces.

Ce modèle est l'un des plus importants de l'apprentissage animal parce qu'il permet de rendre compte d'un grand nombre des phénomènes observés dans le cadre du conditionnement classique, comme le blocage (Kamin, 1969), le super-conditionnement (Rescorla et Wagner, 1972) ou la surévaluation (Kremer, 1978).

Le modèle de Rescorla-Wagner a été repris et adapté au formalisme des processus de décision markoviens par Sutton et Barto (1988), de manière à proposer des algorithmes d'apprentissage par renforcement.

1.2.2 Les processus de décision markoviens et la programmation dynamique

Certains problèmes nécessitent souvent plusieurs actions successives pour être résolus, chacune conditionnée par une situation perçue. Pour représenter de tels problèmes, les recherches en informatique ont conduit à définir le cadre formel des processus de décision markoviens (MDP[5]). Selon ce formalisme, étant donné un automate à états finis spécifiant les conséquences des actions entreprises et une fonction de récompense, les algorithmes de la programmation dynamique (Bellman, 1957) permettent de décider de la meilleure action dans chaque situation. L'utilisation de ce type de techniques suppose donc une connaissance parfaite de son environnement de la part de l'agent. L'objet de la programmation dynamique est la prise de décision. Elle relève donc du domaine de la recherche opérationnelle plutôt que de celui de l'apprentissage.

Un modèle de Markov spécifie les relations entre l'agent et son environnement avec les éléments suivants :

- un ensemble fini d'états S ;
- un ensemble fini d'actions A ;
- une fonction de transition $T \ : S \times A \times S \rightarrow [0,1]$ qui donne la probabilité d'occurrence d'un état en fonction de l'état précédent et de l'action entreprise .
- une fonction de récompense $R \ : S \times A \rightarrow \Re$ qui associe une récompense immédiate à chaque couple état-action.

Les actions représentent les moyens de réponse de l'agent sur son environnement. Les états identifient les perceptions possibles de l'agent. La fonction de transition spécifie la probabilité pour l'agent d'être confronté à un état particulier juste après qu'il a entrepris une action dans un état donné. Disposer de cette fonction permet d'anticiper les conséquences des actions. La fonction de récompense permet de spécifier les objectifs de l'agent. Une récompense négative est une punition et l'agent évitera d'effectuer les actions qui y conduisent. Une récompense immédiate importante favorise la reproduction des actions correspondantes dans les mêmes conditions.

La fonction R, par le biais des récompenses immédiates, permet donc d'assigner des buts à l'agent. Si l'objectif d'un agent nécessite plusieurs actions successives pour être rempli, l'agent ne reçoit généralement de récompense qu'une seule fois, juste après avoir effectué la dernière action lui permettant d'atteindre son but. Dans ce cas, les actions intermédiaires sont créditées d'une récompense nulle. Pourtant, l'agent doit être capable de décider de l'action optimale dans n'importe quelle situation, y compris quand il n'a aucune récompense immédiate. Il faut donc être capable de rétro-propager l'information concernant les récompenses immédiates jusqu'aux états où il n'y en a aucune. Ces informations doivent permettre de décider de l'optimalité de chaque action dans n'importe quel état. Pour ce faire, on associe une qualité $Q(s,a)$ à chaque couple (s,a). Les qualités relatives permettent d'identifier les actions optimales. De plus, on associe à chaque état s une valeur $V(s)$ qui représente le cumul de toutes les récompenses immédiates attendues dans le futur à partir de cet état. Ces récompenses sont dépréciées à mesure qu'elles sont retardées. Ainsi, il est préférable de recevoir les récompenses rapidement. À valeurs égales, les récompenses immédiates sont privilégiées par rapport aux récompenses futures. La partie gauche de la figure 1.5 illustre l'atténuation de la récompense reçue en rejoignant la case marquée `F`.

Pour rétro-propager les récompenses immédiates $R(s,a)$ et calculer une fonction qui associe

[5] *Markov Decision Process*

une qualité $Q(s,a)$ à chaque couple état-action, on définit deux fonctions :

$$V \ : S \rightarrow \Re$$

$$Q \ : S \times A \rightarrow \Re$$

On définit la fonction de valeurs V de la manière suivante :

$$V(s) = \max_{a \in A} Q(s,a) \tag{1.2}$$

La valeur d'un état est donc la récompense cumulée maximale que l'on peut espérer obtenir à partir de cet état. Quant aux qualités, elles se calculent de la manière suivante :

$$Q(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V(s') \tag{1.3}$$

De ces qualités, on peut déduire une politique optimale en stipulant que, dans un état donné s, l'action décidée a est telle que $Q(s,a)$ soit maximale. Une politique est un ensemble de couples de $S \times A$ qui spécifie directement quelle action choisir dans chaque état. Ainsi, bien que l'ensemble de données $V(s)$ et $Q(s,a)$ ne constitue pas une politique, il est possible de les utiliser pour en dériver une. Par souci de simplification, il nous arrivera souvent dans la suite de ce mémoire d'utiliser les termes « calculer une politique » au lieu de « calculer l'ensemble des valeurs $V(s)$ et $Q(s,a)$ nécessaires à la définition d'une politique ».

Dans l'équation 1.3, $R(s,a)$ est la récompense immédiate associée à l'état s et l'action a. La valeur $\sum T(s,a,s').V(s')$ représente la somme des récompenses attendues à partir des différents états futurs possibles, la récompense étant pondérée par la probabilité d'atteindre l'état en question. La qualité associée à un état et à une action est donc la somme de la récompense immédiate correspondante et de la récompense future.

La récompense future est multipliée par un facteur d'amortissement $\gamma \in [0,1]$. Ce facteur permet de ne pas accorder la même importance à la récompense future et à la récompense immédiate. La figure 1.5 illustre l'effet du facteur d'amortissement sur les valeurs des états. Si $\gamma = 0$, alors les qualités ne sont calculées qu'en fonction des récompenses immédiates, et si $\gamma = 1$, alors la récompense future est aussi importante que la récompense immédiate et toutes les valeurs seront égales.

Dans les équations de Bellman, il y a une équation de ce type par état et par action, l'ensemble formant un système d'équations. Chaque valeur et chaque qualité dépend de la valeur et des qualités associées à tous les couples état-action. Pour résoudre ce système d'équations, *Value Iteration* est l'algorithme le plus simple (Bellman, 1957; Bertsekas, 1987). Il est itératif et met directement à jour les valeurs les unes après les autres, en fonction des valeurs déjà calculées. Cet algorithme est donné ci-dessous. Il stocke les valeurs $Q(s,a)$ dans une table et les met à jour de manière itérative. Il s'arrête lorsque les modifications apportées à la dernière itération sont inférieures à un seuil ϵ.

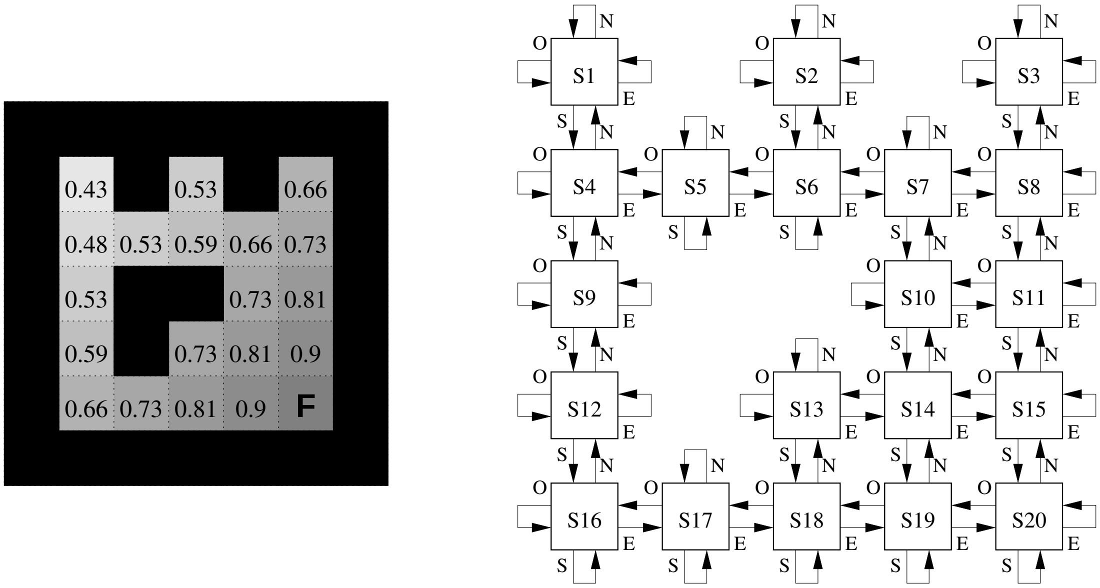

FIG. *1.5: Effet du facteur d'amortissement sur les valeurs associées aux états.* L'environnement présenté à gauche correspond à l'automate présenté à droite. Chaque case de l'environnement correspond à un état différent et l'agent dispose de quatre actions, chacune provoquant un mouvement d'une case vers une des quatre directions cardinales (a_N, a_E, a_S et a_O, respectivement notées N, S, E et O dans l'automate de la partie droite). Les actions provoquent des transitions d'état en état. L'agent reçoit une récompense de 0.9 lorsqu'il rejoint la case marquée `F`, c'est-à-dire s_{20}. Si la fonction de transition spécifie que choisir l'action a_S dans l'état s_{11} mène à la situation s_{15} avec une probabilité de 1 (dans ce cas $T(s_{11}, a_S, s_{15}) = 1$), si la valeur associée à s_{15} est 0.9 (voir figure de gauche) et si la fonction de récompense immédiate spécifie une valeur nulle (dans ce cas $R(s_{11}, a_S) = 0$), alors l'application de l'équation de Bellman donne $Q(s_{11}, a_S) = 0 + \gamma \times 0.9$. Si $\gamma = 0.9$, alors $Q(s_{11}, a_S) = 0.81$. Comme cette valeur est la plus grande parmi $Q(s_{11}, a_N)$, $Q(s_{11}, a_E)$, $Q(s_{11}, a_S)$ et $Q(s_{11}, a_O)$, alors la valeur de s_{11}, $V(s_{11})$, devient égale à 0.81 (voir figure gauche).

La figure 1.6 illustre les itérations successives de l'algorithme *Value Iteration*[6]. Les premières itérations mettent à jour les qualités et les valeurs des états proches des sources de récompense. Ces valeurs mises à jour sont utilisées aux itérations suivantes pour modifier de nouvelles valeurs, de plus en plus éloignées de la récompense, jusqu'à la terminaison de l'algorithme. Dans ce cas, *Value Iteration* converge vers les valeurs illustrées par la figure 1.5.

Il est à noter que ces mécanismes ne fonctionnent que lorsque le problème abordé vérifie l'hypothèse de Markov. Dans ce cas, il n'y a aucune ambiguïté sur les états, c'est-à-dire que les états perçus par le système sont clairement discriminés et contiennent toute l'information nécessaire à la prise de décision. Un problème est dit « non-markovien » lorsque certains états perçus sont ambigus et que la décision nécessite l'adjonction d'informations concernant les expériences

[6]Selon l'ordre dans lequel sont considérés les états à chaque itération, la convergence peut-être plus rapide. La figure 1.6 présente le pire des cas.

Algorithme 1 *Value Iteration*

Répéter
  Pour tout $s \in S$ **Faire**
    Pour tout $a \in A$ **Faire**
      $q \leftarrow Q(s,a)$
      $Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') max_{a' \in A} Q(s',a')$
      $\Delta \leftarrow max(\Delta, |q - Q(s,a)|)$
    Fin Pour
  Fin Pour
Jusqu'à $\Delta < \epsilon$

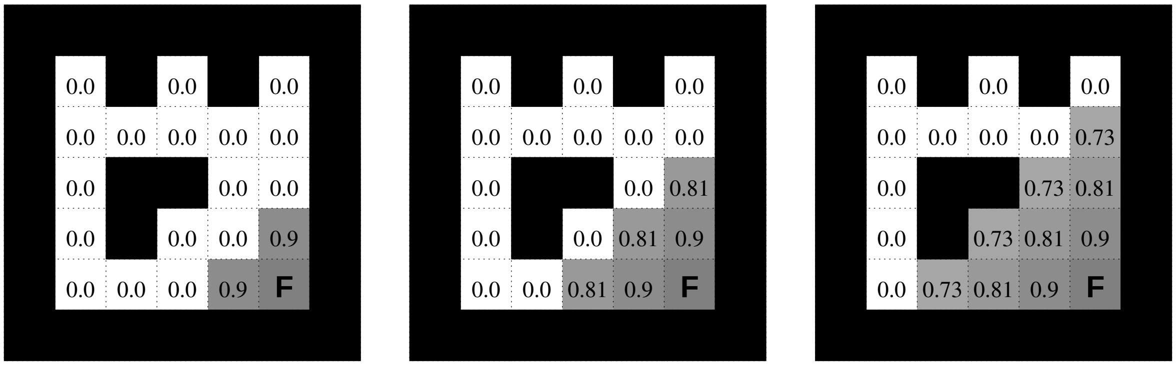

FIG. *1.6: Itérations successives de* Value Iteration.

passées de l'agent. La figure 1.7 illustre les problèmes d'ambiguïté qui peuvent survenir. Elle montre comment la prise en compte des états et des actions passées permet de lever l'ambiguïté sur un état qui, sans ces informations supplémentaires, ne permettrait pas de décider de l'action optimale.

1.2.3 L'algorithme Q-learning

En représentant un problème comme un processus de décision markovien, c'est-à-dire en spécifiant complètement les fonctions R et T, il est possible de calculer itérativement des valeurs dont on peut dériver une politique, c'est-à-dire stipuler pour chaque état la meilleure action à entreprendre. Mais pour utiliser des algorithmes de décision comme *Value Iteration*, il faut que le modèle de Markov sous jacent au problème soit connu. Dans ce cas, il est possible de calculer une politique à n'importe quel moment, en itérant l'algorithme jusqu'à ce qu'il converge.

Lorsqu'un agent est plongé dans un environnement inconnu pour réaliser un apprentissage, les fonctions R et T sont *a priori* inconnues et cet algorithme de décision n'est d'aucun secours. Il faut donc un mécanisme d'apprentissage capable de modéliser les fonctions R et T, ou alors apprendre directement les qualités associées aux couples état-action. Dans ce cas, les qualités sont ajustées progressivement, tout au long de l'expérience de l'agent.

L'algorithme *Q-learning* (Watkins, 1989) permet de calculer incrémentalement et directement

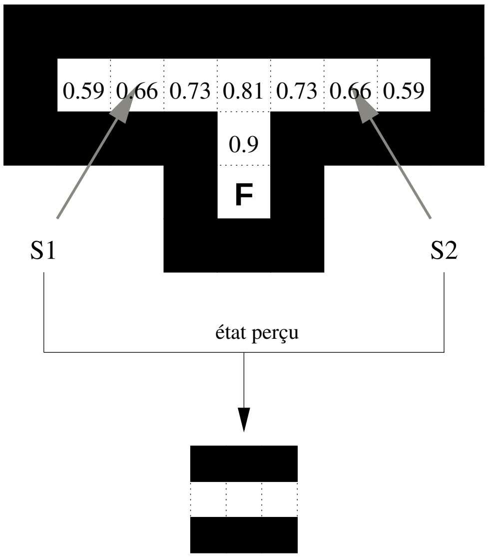

FIG. *1.7: Exemple de problème non-markovien.* Dans cet environnement, l'agent perçoit les huit cases immédiatement autour de lui. Les états sont alors identifiés par l'absence ou la présence de murs dans chacune des cases adjacentes. Or, dans les cases s_1 et s_2, l'agent perçoit la même chose. Ces deux cases sont ambiguës. Or, aller vers l'est à partir de la case s_1 éloigne de la nourriture alors que la même action entreprise à partir de s_2 en rapproche. Il est impossible de calculer une qualité unique associée à l'état et à l'action en question. Pour différencier les états ambigus, on doit avoir recours à une mémoire du passé. Par exemple, bien que s_1 et s_2 soient semblables, savoir si l'on vient du coin gauche ou du coin droit permet de lever l'ambiguïté.

la fonction de qualité en fonction de l'expérience. L'apprentissage est direct puisqu'il ne nécessite de connaissance préalable ni de la fonction de transition, ni de la fonction de récompense. Cet algorithme emprunte au formalisme des processus de décision markoviens la possibilité de prendre en compte les actions, et il met à jour les qualités conformément au modèle de Rescorla-Wagner.

Les qualités associées à chaque couple état-action sont initialisées avec des valeurs nulles. L'équation de mise à jour des qualités considère successivement les quadruplets $(s_{t-1}, a_{t-1}, r_t, s_t)$. En fonction de ces informations, à chaque pas de temps, une qualité est modifiée de la manière suivante[7] :

$$Q(s_{t-1}, a_{t-1}) \leftarrow Q(s_{t-1}, a_{t-1}) + \beta \left(r_t + \gamma \max_{a \in A} [Q(s_t, a) - Q(s_{t-1}, a_{t-1})] \right) \tag{1.4}$$

Une seule qualité est donc mise à jour à chaque pas de temps, en fonction des qualités précédemment calculées. L'apprentissage n'utilise pas la fonction de transition $T : S \times A \times S \rightarrow [0, 1]$ qui permet d'anticiper les conséquences des actions. Ce n'est que lorsqu'une transition $(s_{t-1}, a_{t-1}, r_t, s_t)$ survient qu'elle est utilisée pour mettre à jour $Q(s_{t-1}, a_{t-1})$ en fonction des qualités associées à l'état suivant s_t.

[7] Cette formulation de l'équation du *Q-learning* est empruntée à Josefowiez (2001)

Contrairement au modèle de Rescorla-Wagner qui décrit le conditionnement classique, elle permet de prendre en compte l'action. En s'inspirant de l'apprentissage animal et du conditionnement classique, on a donc conçu des algorithmes d'apprentissage par renforcement qui incluent l'action.

1.3 L'apprentissage latent pour les ordinateurs

1.3.1 Les problèmes du Q-learning

Mise à jour d'une seule qualité par pas de temps

L'algorithme *Q-learning* permet d'apprendre une approximation de la fonction de qualité. Il en construit un modèle constitué de triplets $(s, a, Q(s, a))$, en ajustant les valeurs $Q(s, a)$ progressivement, tout au long de son expérience. Si cet algorithme converge avec une probabilité de 1 vers des qualités optimales, il se révèle en pratique assez inefficace dès que le nombre d'états et de transitions augmente.

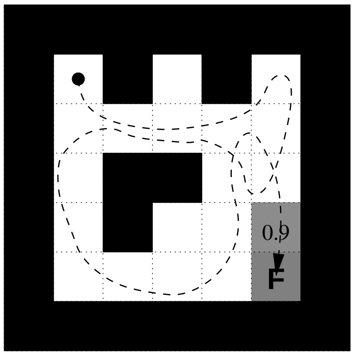

FIG. *1.8: Q-learning (1er essai).*

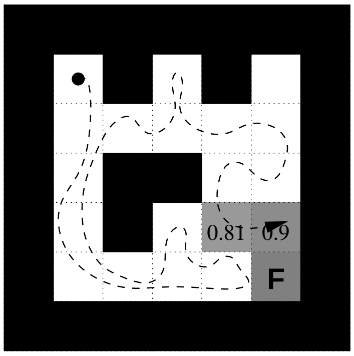

FIG. *1.9: Q-learning (2e essai).*

En effet, le *Q-learning* ne dispose pas de la fonction de transition $T : S \times A \times S \to [0,1]$. À chaque instant, il n'utilise que le quadruplet $(s_{t-1}, a_{t-1}, r_t, s_t)$ pour mettre à jour la fonction de qualité. Cet algorithme n'apprend en modifiant les qualités que lorsqu'il effectue réellement une action.

Il en résulte une certaine lenteur d'apprentissage illustrée par les figures 1.8 et 1.9. Dans cet exemple, l'expérience est composée d'essais successifs. Chaque essai commence dans la case nord-ouest. Dès que l'agent rejoint la case contenant de la nourriture (marquée F), il reçoit une récompense scalaire et un nouvel essai commence.

Au début de la phase d'apprentissage, l'agent n'a aucune connaissance de son environnement et toutes les qualités sont nulles. Entre plusieurs actions chacune créditées d'une qualité nulle, étant donné une situation, l'agent en choisit une au hasard. La politique initiale est donc une politique aléatoire. Durant sa marche aléatoire, il ne reçoit aucun renforcement et n'apprend rien jusqu'à ce qu'il atteigne la case marquée F et qu'il reçoive une récompense (voir figure 1.8). Il met alors à jour la qualité correspondante au couple état-action qui lui a permis de recevoir cette récompense.

Un nouvel essai débute alors et l'agent recommence une marche aléatoire puisqu'il n'a toujours rien appris concernant les états éloignés de la source de récompense. Ce n'est que lorsqu'il arrivera par hasard dans l'état précédemment mis à jour qu'il pourra modifier son modèle en mettant à jour une nouvelle qualité en fonction de celle calculée précédemment (voir figure 1.9). Ce n'est qu'après un grand nombre d'essais commençant chacun par une marche aléatoire que les valeurs optimales seront trouvées (voir figure 1.5).

L'algorithme *Q-learning* n'utilisant pas de modèle de la fonction de transition, il ne peut mettre à jour qu'une seule qualité à chaque pas de temps : celle associée à (s_{t-1}, a_{t-1}). S'il était capable de mettre à jour plusieurs qualités à chaque pas de temps, sa vitesse d'apprentissage serait accrue.

Nécessité d'une politique stochastique

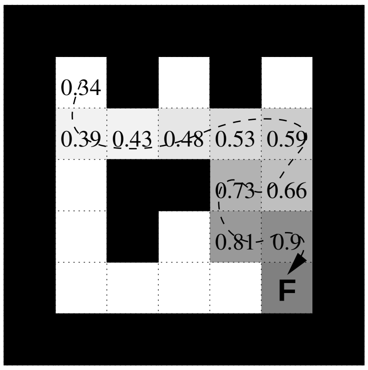

FIG. *1.10: Q-learning (nécessité d'une politique stochastique).*

En outre, il se peut que l'agent apprenne une fonction Q comme celle illustrée par la figure 1.10. Dans cet exemple, les explorations aléatoires successives auront conduit l'agent à ne mettre à jour que certaines qualités associées aux cases grisées. Ces qualités conduisent l'agent à répéter sans cesse la politique sous-optimale conduisant à la trajectoire indiquée en pointillés. L'agent exploite ce qu'il a appris et ne visite plus d'autres états que ceux qu'il a déjà visités. Or explorer de nouveaux états est le seul moyen de découvrir la politique optimale. Par exemple, bien que les qualités calculées lui dictent d'aller à l'ouest à partir de la case de valeur 0.66, un mouvement vers le sud lui permettrait de découvrir la politique optimale et de créditer l'état en question d'une valeur de 0.81.

En fait, l'algorithme *Q-learning* ne converge avec une probabilité de 1 que si tous les couples (s, a) sont visités suffisamment souvent. Il y a là un dilemme entre exploration et exploitation. L'exploration permet de visiter régulièrement tous les états mais éloigne des sources de récompenses, et l'exploitation permet de maximiser la récompense mais pas de visiter tous les états suffisamment souvent. La façon la plus courante d'aborder ce dilemme est de définir une politique stochastique en fonction des qualités calculées par l'algorithme *Q-learning*. La politique n'est alors pas déduite des qualités en choisissant systématiquement la meilleure action, mais en choisissant parfois une action identifiée comme sous-optimale. Donc les critères permettant d'apprendre des qualités adéquates sont en contradiction avec la volonté d'obtenir une politique optimale.

1.3.2 L'architecture Dyna

Les problèmes d'efficacité de l'algorithme *Q-learning* viennent surtout de ce que l'agent ne met pas à profit toute l'information offerte par sa boucle sensori-motrice. De la même manière que dans les théories du conditionnement opérant en psychologie animale, il n'apprend que ce qui relève de la récompense. Pour mettre à jour plusieurs qualités à chaque pas de temps, l'algorithme *Q-learning* devrait être capable de considérer des transitions hypothétiques, indépendamment de la dernière transition observée. Mais pour cela, il faut être capable d'anticiper les conséquences de ses actions, et donc de disposer de la fonction de transition. Dans un environnement *a priori* inconnu, cette fonction est inconnue et il faut en apprendre un modèle. Or la boucle sensori-motrice offre aussi l'information nécessaire à l'apprentissage latent de la fonction de transition, puisque les conséquences d'une action ne consistent pas seulement en une récompense, mais aussi en une nouvelle situation.

Sutton (1992) propose l'architecture *Dyna* pour permettre d'apprendre un modèle de la fonction de transition, et pour l'utiliser afin d'accélérer l'apprentissage des qualités associées aux couples état-action.

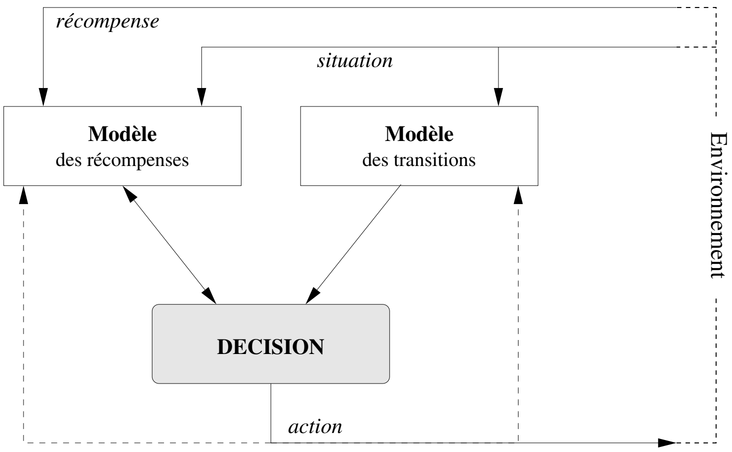

FIG. *1.11: L'architecture* Dyna.

Dans *DynaQ+*, le modèle des récompenses immédiates spécifie les récompenses associées aux couples (s_t, a_t). Le modèle de l'environnement est constitué d'un ensemble de triplets (s_t, a_t, s_{t+1}) décrivant les transitions rencontrées. À chacun de ces triplets est associée la probabilité p

de la transition. Cette probabilité est mise à jour en fonction de l'expérience réelle de l'agent. Ce modèle de l'environnement est appris de manière latente, indépendamment de la récompense.

Comme dans l'algorithme *Q-learning*, les qualités associées au dernier couple état-action sont mises à jour à chaque pas de temps. Mais, en plus, le modèle environnemental est utilisé pour simuler des actions. À chaque pas de temps, contrairement à *Q-learning*, *DynaQ+* est donc capable de mettre à jour les qualités associées à plusieurs couples état-action. La rapidité de l'apprentissage des qualités s'en trouve nécessairement améliorée. À n'importe quel moment, *DynaQ+* peut ainsi utiliser son modèle de l'environnement et appliquer un algorithme semblable à *Value Iteration*, en simulant autant d'actions qu'il est nécessaire pour mettre à jour toutes les qualités. *DynaQ+* utilise donc des techniques issues de la recherche opérationnelle pour des problèmes d'apprentissage (Gérard *et al.*, 2003).

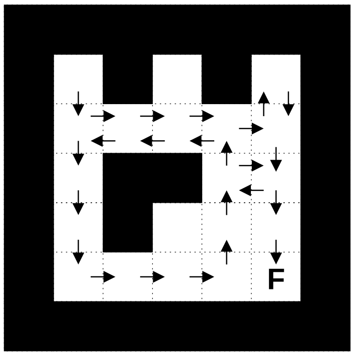

FIG. *1.12: DynaQ+ (1er essai) : construction du modèle.*

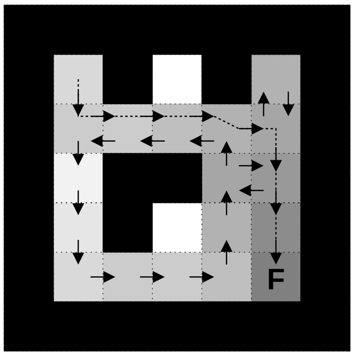

FIG. *1.13: DynaQ+ (fin du 1er essai) : identification d'une récompense immédiate, et utilisation du modèle de l'environnement pour mettre à jour toutes les qualités possibles. Dans cet exemple, le nombre d'actions simulées est plus grand que celui utilisé par Sutton, et le résultat est identique à ce que permet d'obtenir une application de l'algorithme* Value Iteration *en fonction du modèle de l'environnement.*

Dans l'exemple de la figure 1.8 développé dans la section 1.3.1 pour illustrer le *Q-learning*, lorsque l'agent est contraint de procéder à une marche aléatoire faute de connaissances sur son environnement, il peut néanmoins construire un modèle de la fonction de transition. Dans cet exemple, au cours du premier essai, il peut intégrer à son modèle de l'environnement toutes les transitions représentées par des flèches dans la figure 1.12. Dès que l'agent reçoit un renforcement et qu'un nouvel essai commence, plutôt que de procéder à une nouvelle marche aléatoire, il peut utiliser son modèle pour mettre à jour plusieurs qualités. Dans l'exemple de la figure 1.13, toutes les transitions ont été simulées plusieurs fois et un grand nombre de qualités ont pu être mises à jour avant même que le nouvel essai commence. Ici, la politique résultante des qualités calculées (en pointillés) est optimale à partir de la case nord-ouest. En général, les qualités ainsi mises à jour ne permettent pas de définir immédiatement une politique optimale puisque le modèle de l'environnement n'est pas encore complet. Néanmoins, la simulation d'actions grâce à un modèle de l'environnement permet d'accélérer l'apprentissage des qualités.

Dans ce chapitre, nous avons indiqué comment la mise au point d'algorithmes d'apprentissage par renforcement a été influencée par la littérature sur l'apprentissage animal. Nous avons montré comment les limitations de ces algorithmes ont été dépassées par l'introduction d'un mécanisme d'apprentissage latent pour construire un modèle prédictif de la dynamique des interactions entre l'agent et son environnement.

L'architecture *Dyna* utilise un tel apprentissage latent dans le cadre de l'apprentissage par renforcement. Pour mettre au point une telle architecture, il est nécessaire de construire un modèle de l'environnement, d'une part, et de pouvoir l'utiliser pour accélérer l'apprentissage d'une politique, d'autre part. Nous nous posons maintenant le problème de la construction du modèle produit par cet apprentissage latent en cherchant à améliorer l'approche de *DynaQ+*, qui élabore une liste exhaustive de toutes les transitions possibles. Dans le chapitre suivant, nous décrivons les systèmes de classeurs, qui vont nous servir à construire des modèles compacts et intelligibles.

Chapitre 2

Les systèmes de classeurs

Dans ce chapitre, nous introduisons les systèmes de classeurs. Ces systèmes à base de règles permettent de résoudre des problèmes d'apprentissage par renforcement et procèdent à une généralisation qui exploite les régularités de la dynamique des interactions entre l'agent et son environnement. Nous introduisons en particulier le système ACS (Stolzmann, 1998), qui permet d'exploiter ces capacités de généralisation pour réaliser l'apprentissage latent d'un modèle de l'environnement.

2.1 Le paradigme originel des systèmes de classeurs

2.1.1 Généralisation dans les systèmes de classeurs

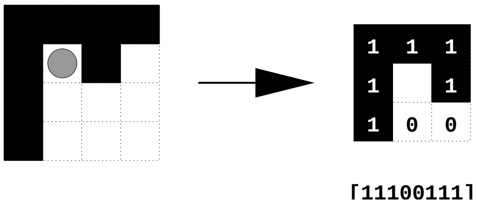

FIG. *2.1: Attributs et situation.*

Dans les problèmes abordés par les systèmes de classeurs (Holland, 1976), les états sont caractérisés par plusieurs attributs représentant autant de propriétés perceptibles de l'environnement. Nous préférons alors parler de situations perçues plutôt que d'états. Une situation est un vecteur de plusieurs valeurs discrètes, une pour chacun des attributs perçus. Par exemple, un agent dans un monde de cases comme celui illustré par la figure 2.1 peut percevoir huit attributs, chacun exprimant la présence ou la présence d'un obstacle sur une des huit cases adjacentes. La présence d'un mur dans une direction donnée est identifiée par un symbole `1` et l'absence de mur par un symbole `0`. Les attributs sont considérés dans l'ordre des aiguilles d'une montre, en commençant par l'attribut au nord. Ainsi, l'agent représenté par le cercle grisé perçoit sa situation localement comme un vecteur d'attributs. Ici, l'agent perçoit `[11100111]`.

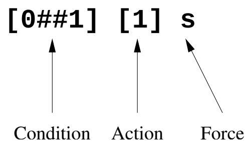

FIG. *2.2: Exemple de classeur.*

appariement	0	1	#
0	V	F	V
1	F	V	V
#	V	V	V

TAB. *2.1: Table de vérité de l'opérateur d'appariement*

Un système de classeurs est caractérisé par un ensemble de règles de décision appelées classeurs. Chacun de ces classeurs est caractérisé par une partie `Condition`, une partie `Action` et une force `s` (voir figure 2.2). Les parties `Condition` spécifient dans quelles situations le classeur est applicable, et les parties `Action` spécifient les réponses du système à ces situations. Ainsi, le classeur `[11100111] [01]` propose l'action `[01]` dans la situation `[11100111]`. Si l'action `[01]` représente un mouvement vers le sud, alors le classeur en question propose un déplacement vers le sud dans la situation correspondant à la figure 2.1. Si, dans une certaine situation, plusieurs classeurs proposent chacun une action, leurs forces respectives sont utilisées pour sélectionner le classeur dont l'action est exécutée dans l'environnement.

Les attributs des classeurs peuvent prendre une valeur discrète ou une valeur générale `#`. Un vecteur d'attributs est apparié avec un autre lorsque leurs attributs de même rang sont appariés deux à deux. La table de vérité de l'opérateur d'appariement pour les attributs est donnée par le tableau 2.1.

Un attribut de valeur `#` est donc apparié avec n'importe quelle valeur. Dans une partie `Condition`, il opère donc comme un joker permettant d'ignorer la valeur d'un attribut. Par contre, un attribut avec une valeur particulière (`0` ou `1`) spécifie une restriction pour l'ensemble des vecteurs qui sont appariés avec la condition. À titre d'exemple, la condition `[#0]` n'est appariée qu'avec les vecteurs `[00]`, `[10]`, `[0#]`, `[1#]` et `[##]`, elle ne l'est pas avec `[01]`, `[11]` et `[#1]`. La transformation d'un attribut en symbole `#` rend la condition plus générale (elle est appariée avec un plus grand nombre de situations). La condition `[#1]`, par exemple, est plus générale que la condition `[01]`, mais elle n'est pas plus générale que la condition `[1#]` bien qu'elle lui soit appariée. Remplacer un attribut `#` par une valeur particulière introduit une distinction pour réaliser une spécialisation. La nouvelle condition est alors appariée à moins de situations que l'ancienne.

En utilisant ces symboles `#` dans les parties `Condition`, il est possible de spécifier des domaines de validité englobant plusieurs situations. Il devient alors possible d'exhiber des régularités ne fai-

[10100011]	[01]
[10100111]	[01]
[11100011]	[01]
[11100111]	[01]
[1#100#11]	[01]

TAB. *2.2: Exemple de généralisation*

sant intervenir qu'une partie de ces attributs. Exhiber de telles régularités permet de généraliser, c'est-à-dire de représenter plusieurs couples situation-action avec une une seule règle.

Par exemple, un seul classeur `[1#100#11] [01]` permet de proposer l'action `[01]` dans quatre situations (`[10100011]`, `[10100111]`, `[11100011]` et `[11100111]`, voir tableau 2.2). Si l'action `[01]` est effectivement optimale dans chacune de ces quatre situations, alors l'utilisation d'un tel classeur permet de réduire la taille du système. En outre, si le système n'avait jamais perçu que les trois premières situations avant de généraliser, il serait tout de même capable d'agir de manière adéquate dans la quatrième situation, sans jamais l'avoir perçue.

2.1.2 Architecture d'un système de classeurs

L'idée originale des systèmes de classeurs est présentée par Holland (1976) et le premier système, CS1, est décrit par Holland et Reitman (1978). Ces systèmes permettent d'utiliser des algorithmes génétiques (Goldberg, 1989) pour des tâches d'apprentissage par renforcement.

L'architecture générale de tels systèmes est présentée dans la figure 2.3. Ils se composent d'une interface d'entrée, d'une interface de sortie, d'une liste de classeurs et d'une liste de messages. L'interface d'entrée traduit les perceptions de l'agent en messages d'entrée qui peuvent être appariés avec les conditions des classeurs. L'interface de sortie traduit les messages de sortie en actions effectives de l'agent sur l'environnement. Ainsi, à chaque pas de temps, le système reçoit un message correspondant à une situation perçue, et décide quel message renvoyer et donc quelle action entreprendre. De plus, le système reçoit un signal de renforcement numérique qu'il exploite pour apprendre quelle action choisir dans chaque situation, pour maximiser sa récompense.

À chaque pas de temps, les messages d'entrée sont postés sur la liste de messages du système de classeurs. Les messages de la liste sont appariés avec les conditions des classeurs caractérisant le système. Si la condition d'un classeur est appariée avec un message de la liste, le classeur est activé. Les messages ayant provoqué une telle activation disparaissent alors de la liste. Chaque classeur activé poste ensuite le message correspondant à sa partie `Action` sur la liste de messages. À ce point, certains messages ont été supprimés de la liste, d'autres y ont été ajoutés. Les messages qui peuvent être interprétés par l'interface de sortie sont traduits en actions effectives sur l'environnement et supprimés de la liste des messages.

Il est à noter que certains messages postés sur la liste ne correspondent à aucune action effective du système. Ces messages ne sont donc pas supprimés par l'interface de sortie et il

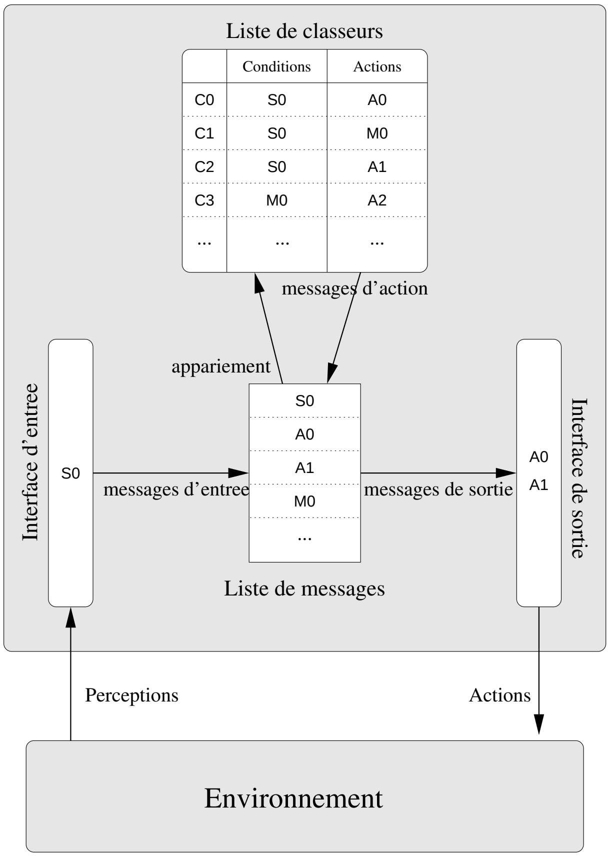

FIG. *2.3: Architecture d'un système de classeurs.*

restent sur la liste jusqu'au prochain pas de temps. Ils seront alors éventuellement appariés à des classeurs. De tels messages sont dits « internes ». Ils permettent de produire une information qui n'est pas directement interprétable, ni en tant que situation perçue, ni en tant qu'action. Cette information est persistante d'un pas de temps à l'autre et peut être utilisée plus tard par le système. L'ensemble de ces messages définit l'état interne du système. Sans eux, le comportement de l'agent ne disposerait d'aucune mémoire et il serait purement réactif : les actions ne pourraient être choisies qu'en réaction à la situation courante, sans prendre en compte les situations passées.

2.1.3 Apprendre avec des algorithmes génétiques

Algorithmes génétiques

Dans la plupart des systèmes de classeurs, la création et la sélection des classeurs fiables est réalisée par des algorithmes génétiques (Goldberg, 1989). De tels algorithmes utilisent un génome pour caractériser une solution à un problème posé. Ce génome est une liste de symboles permettant de construire la solution représentée[8]. Chaque solution est un individu au sein de la

[8]Dans le cas des systèmes de classeurs, les symboles 0, 1 ou # caractérisant les classeurs forment directement le génome. L'opération de traduction d'un génome en classeur consiste donc simplement à délimiter les parties

population. L'algorithme attribue une note à chacun de ces individus en fonction de sa capacité à résoudre le problème posé. Cette note est appelée « valeur sélective », elle est attribuée lors d'une évaluation, en confrontant l'individu au problème.

Un algorithme génétique est initialisé avec une population d'individus dont les génomes sont tirés au hasard. Le déroulement d'un tel algorithme est composé de générations successives. À chaque génération, les moins bons individus sont supprimés et d'autres sont créés. Pour créer un nouvel individu, on construit généralement un génome à partir des génomes d'individus crédités d'une bonne valeur sélective. Pour ce faire, on a recours à des opérateurs de croisement et de mutations. L'opérateur de croisement, illustré par la figure 2.4, croise deux solutions pour en créer deux nouvelles. Les points de croisement sont aléatoires. L'opérateur de mutation, illustré par la figure 2.5, modifie aléatoirement un symbole du génome. À chaque génération, les opérateurs de croisement et de mutation permettent de créer de nouveaux individus à partir des meilleurs individus de la population, et un mécanisme de sélection supprime les moins bons. Ainsi, de génération en génération, les individus sont de plus en plus adaptés à la résolution du problème, jusqu'à ce qu'une solution satisfaisante soit trouvée.

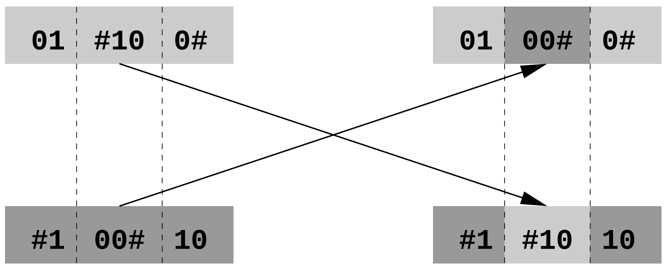

FIG. *2.4: Opérateur génétique de croisement appliqué à un classeur.* Ici, deux points de croisement sont choisis aléatoirement. Ces points de croisement spécifient trois parties du génome, qui sont mélangées de manière à créer deux nouveaux individus.

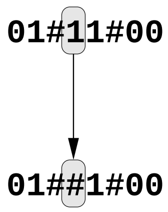

FIG. *2.5: Opérateur génétique de mutation appliqué à un classeur.* Un point de mutation et une nouvelle valeur sont choisis aléatoirement. Ici, le quatrième symbole est changé en symbole `#`.

Systèmes de classeurs de type Pittsburgh

On distingue deux types de systèmes de classeurs, qui diffèrent quant aux modalités d'application des algorithmes génétiques : le type « Pittsburgh » (Smith, 1980) et le type « Michigan » (Holland et Reitman, 1978). Dans le type « Pittsburgh » illustré par la figure 2.6, un génome représente une base de règles complète. Un tel système de classeur gère donc une population d'individus qui sont chacun un système à base de règles. L'opérateur de croisement mélange

`Condition` et `Action`. Par exemple, le classeur `[0#1#] [1010]` est caractérisé par le génome `0#1#1010`.

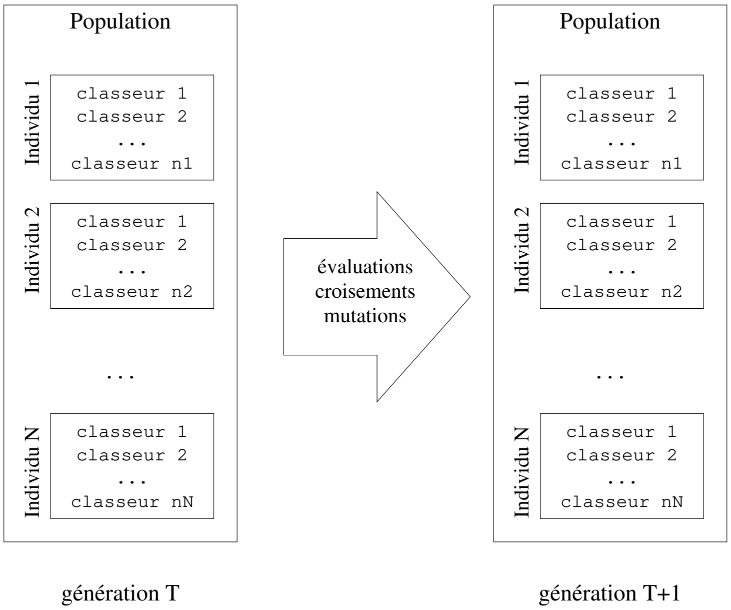

FIG. *2.6: Système de classeurs de type « Pittsburgh »*

les bases de règles de manière à explorer de nouvelles solutions. Chaque ensemble de règles est confronté au problème posé et se voit attribuer une valeur sélective. Le mécanisme de sélection ne conserve que les meilleures bases de règles.

Pour réaliser un agent adaptatif avec de tels systèmes, il faut mettre en compétition un grand nombre de bases de règles à l'intérieur de l'agent. Chaque base de règles doit avoir été évaluée avant qu'il soit possible de passer à la génération suivante. Le nombre d'évaluations nécessaires à un apprentissage est donc très important et les évaluations doivent pouvoir être réalisées très rapidement. C'est pourquoi ce type de système est surtout utilisé lorsqu'il est possible de procéder rapidement aux évaluations, grâce à une simulation.

Ce type de système est surtout utilisé lorsque l'on veut trouver une solution dont on sait qu'elle n'aura pas besoin d'être modifiée ultérieurement, ce qui n'est pas le cas pour l'apprentissage incrémental. Lorsqu'un système à base de règles adéquat a été trouvé, il est en effet possible de l'utiliser indépendamment, mais dans ce cas, il ne change plus pendant son utilisation. Pour garder une adaptation pendant l'exploitation du système, il faut conserver tous les systèmes, de manière à appliquer l'algorithme génétique. Dans ce cas, le nombre de règles à gérer est très important et on préférerait avoir un système permettant, par apprentissage, de modifier incrémentalement une unique base de règles.

Systèmes de classeurs de type Michigan

Dans les systèmes de classeurs de type « Michigan », illustrés par la figure 2.7, l'algorithme génétique opère sur une population de classeurs. Cette population forme un unique système à base de règles et les classeurs sont les individus. L'algorithme génétique cherche à évaluer la

FIG. *2.7: Système de classeurs de type « Michigan »*

participation de chacun des classeurs à la résolution du problème général, et sélectionne les classeurs qui participent le mieux à la résolution du problème. En procédant ainsi, les systèmes de classeurs de type « Michigan » permettent une adaptation en ligne en n'utilisant qu'une seule base de règles.

À chaque génération, l'algorithme génétique crée de nouveaux classeurs en opérant à des croisements (voir figure 2.4) et des mutations (voir figure 2.5) sur les meilleurs d'entre eux. L'opérateur de croisement permet de mélanger les attributs de deux classeurs et la mutation introduit du bruit dans la réplication des attributs. Certains des classeurs ainsi créés ne seront pas du tout adaptés à la résolution du problème, mais d'autres permettront d'améliorer les performances du système. Les premiers seront écartés, et les seconds seront conservés et utilisés pour créer de nouveaux classeurs. Si l'algorithme génétique converge, la liste de classeurs contient surtout des classeurs adaptés à la résolution du problème. Dans cette thèse, nous portons notre attention sur les systèmes de type « Michigan », puisqu'ils sont mieux adaptés aux problèmes d'apprentissage.

Pour utiliser un algorithme génétique, il faut être en mesure d'évaluer les classeurs et de leur attribuer à chacun une valeur sélective. Dans les systèmes de classeurs, on associe une force à chacun des classeurs. Cette force est modifiée par l'algorithme *Bucket brigade*, qui rétro-propage[9] la récompense environnementale de classeur en classeur, en appliquant un mécanisme de taxe à chaque étape.

Ainsi, les classeurs qui sont souvent impliqués dans un succès rapide sont crédités d'une force importante. Cette force est liée à la récompense environnementale et elle est aussi utilisée comme valeur sélective pour l'algorithme génétique. Ainsi, seuls les classeurs permettant effectivement de maximiser le cumul au long terme des récompenses environnementales sont conservés.

[9] La rétro-propagation amortit les récompenses à mesure qu'elles sont prises en compte par des classeurs éloignés loin des sources de récompenses, comme dans l'algorithme *Q-learning*, section 1.2.3

2.2 ZCS : un système de classeurs sans liste de messages

2.2.1 Architecture de ZCS

Bien que l'architecture des systèmes de classeurs présentée plus haut soit très générale, elle peut induire des problèmes de rapidité d'apprentissage. En particulier, la capacité à poster des messages internes augmente considérablement l'espace de recherche des algorithmes génétiques.

Wilson (1994) a proposé ZCS (Zeroth-level Classifier System), un système de classeurs « de niveau zéro » qui ne conserve que le strict nécessaire au fonctionnement d'un système de classeurs. En particulier, ZCS supprime la liste de messages : les messages d'entrée sont directement confrontés aux parties `Condition` des classeurs, et les parties `Action` sont directement traduites en actions effectives. La structure des classeurs reste la même : les classeurs sont caractérisés par une partie `Condition` et une partie `Action`, et une force leur est associée (voir figure 2.2).

Ainsi, dès qu'une situation est perçue, les classeurs dont la condition est appariée avec cette situation sont tous activés directement. Chacun de ces classeurs proposant une action, le système opère à un arbitrage en fonction de la force de ces classeurs, et l'action proposée par le classeur avec la plus grande force est immédiatement envoyée aux effecteurs, sans passer par une liste de messages.

En se privant de liste de messages, ZCS se prive de messages internes et des capacités de mémoire qui y sont attachés. Le spectre des problèmes abordés est dès lors réduit aux problèmes markoviens (voir section 1.2.2) : le système est réactif puisqu'il ne fonde ses actions que sur la dernière situation perçue et ne peut plus avoir recours à des états internes. Toutefois, appliqué aux problèmes markoviens, le système résultant est plus simple et mieux analysable que les systèmes à liste de messages.

Dans ZCS, la valeur sélective des classeurs est toujours leur force, qui est ajustée grâce à l'algorithme de la *Bucket brigade*. Les problèmes résultant de cette assimilation de la valeur sélective d'un classeur à sa force sont discutés à la section suivante.

2.2.2 Le problème de la maintenance des longues chaînes d'actions

La simplicité de ZCS a permis d'identifier un problème majeur lié à l'utilisation de la force des classeurs comme valeur sélective pour l'algorithme génétique.

Dorigo (1994) montre l'équivalence formelle entre l'algorithme de la *Bucket brigade* et le *Q-learning* présenté dans la section 1.2.3. Le mécanisme de taxes de la *Bucket brigade* produit le même effet sur les forces que le facteur d'amortissement utilisé dans le *Q-learning* (voir section 1.2.3). Les classeurs appariés à des situations éloignées de toute récompense[10] sont donc crédités d'une force moins importante. En conséquence, la qualité d'une action sous-optimale, mais associée à une situation proche d'une source de récompense, peut très bien être plus grande que la qualité d'une action optimale, mais associée à une situation qui se trouve loin de toute source de récompense (voir figure 2.8). Or dans certains systèmes de classeurs, la force est utilisée comme

[10] loin en termes de nombre d'action nécessaires pour rejoindre le but

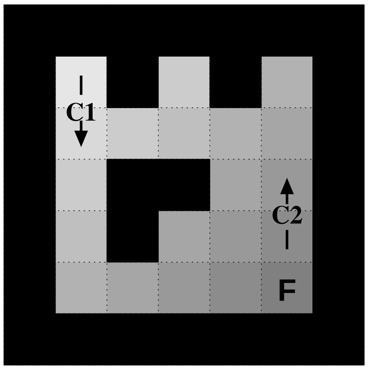

Fig. *2.8: Problème lié à l'assimilation de valeur sélective d'un classeur à sa force.* La force des classeurs correspond à la valeur – au sens de l'équation 1.2 – des situations dans lesquelles mène l'action proposée. Dans cet exemple, le classeur C1 propose une action optimale pour rejoindre le but. Pourtant, puisqu'il est applicable loin de toute source de récompense, il est crédité d'une force moins importante que le classeur C2, qui propose portant une action sous-optimale. En conséquence, C1 risque d'être éliminé alors qu'il est optimal.

valeur sélective. Certains classeurs, qui proposent pourtant une action optimale pour les situations appariées avec leur partie `Condition`, peuvent avoir une force moindre que d'autres, qui proposent une action sous-optimale pour des situations plus proches d'une source de récompense. Ils peuvent donc être éliminés.

Ce problème devient plus important à mesure que l'environnement grandit, que la chaîne d'action pour atteindre le but s'allonge, et que certaines situations ont une valeur faible. Le problème de la maintenance des longues chaînes d'actions consiste donc à ne pas perdre les classeurs optimaux qui doivent intervenir dans une longue séquence d'actions qui n'est récompensée qu'à la fin.

2.3 XCS : un système de classeurs pour modéliser la fonction de récompense

2.3.1 Résolution du problème des longues chaînes d'actions

Donnart et Meyer (1996) proposent une architecture hiérarchique pour découper une tâche complexe en plusieurs opérations plus simples, et éviter ainsi ces problèmes de maintenance de longue chaîne d'actions. Sans utiliser d'architecture hiérarchique et en conservant une liste de classeurs unique, Wilson (1995) propose de ne plus considérer la qualité d'un classeur comme sa valeur sélective.

Dans le système correspondant, appelé XCS, une prédiction `p` est associée à chaque classeur `Condition Action`. Cette prédiction correspond aux qualités de *Q-learning* (voir section 1.2.3) et aux forces d'autres systèmes de classeurs (voir section 2.1.1) : elle représente la récompense attendue en effectuant l'action de la partie `Action` dans toutes les situations appariées à la

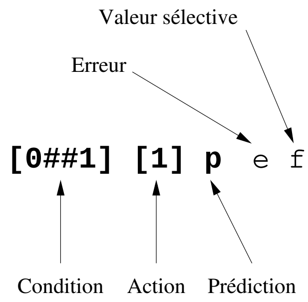

FIG. *2.9: Exemple de classeur dans XCS.*

partie `Condition`. Cette prédiction correspond aux qualités $Q(s,a)$ de l'équation 1.3. Pour rétro-propager la récompense environnementale et mettre à jour les prédictions, XCS applique les équations de Bellman aux classeurs `Condition Action` de la même manière que cela est fait pour les couples (s,a) dans le *Q-learning*.

Contrairement à la force des classeurs de ZCS, la prédiction de la récompense associée à chaque classeur n'est pas utilisée directement comme valeur sélective pour l'algorithme génétique. En plus de cette prédiction, XCS associe en effet à chaque classeur une mesure de l'erreur commise sur la prédiction. Cette erreur estime dans quelle mesure la prédiction est fiable ou non. La valeur sélective est alors calculée sur la base de cette erreur plutôt que sur la prédiction elle-même.

Dès lors, XCS ne cherche plus simplement à trouver des classeurs adaptés au problème, mais à trouver une approximation de la fonction de qualité $Q(s,a)$, sans se limiter aux classeurs proposant une action optimale. XCS ne sélectionne pas seulement les classeurs proposant une action optimale, mais sélectionne tous les classeurs permettant de répondre avec précision à la question « quelle est la qualité associée à une situation et une action quelconque ? ». En cela, XCS apprend un modèle de la fonction de qualité définie dans la section 1.2.3.

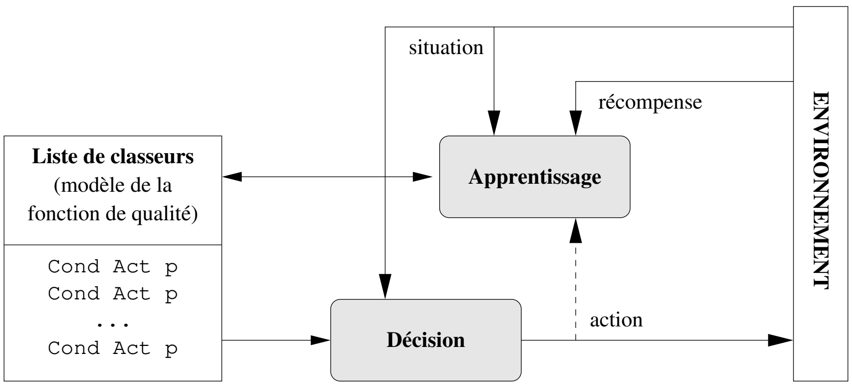

FIG. *2.10: Architecture de XCS.*

Ainsi, XCS apprend un modèle des récompenses attendues et la sélection de l'action adéquate consiste à utiliser ce modèle pour choisir, dans chaque situation, l'action proposée par le classeur

	Action		
Situation	[1]	[2]	[3]
[101110]	0.81	...	...
[001100]	0.81	...	...
[101110]	0.81	...	...
[110001]	0.72	...	...
[010011]	0.72	...	...

TAB. *2.3: Qualités associées aux couples situation-action dans une représentation tabulaire*

apparié prédisant la meilleure récompense. Par contre, l'algorithme génétique utilise l'erreur sur la qualité prédite par les classeurs comme valeur sélective, indépendamment de la prédiction elle-même. Donc, que son action soit optimale ou non, que sa prédiction soit importante ou non, un classeur est conservé tant que son erreur est faible. XCS résout le problème de la maintenance de longues chaînes d'actions en ne cherchant pas directement à résoudre le problème de la maximisation de la récompense attendue, mais en apprenant un modèle de la fonction de récompense, et en utilisant un mécanisme de décision séparé pour utiliser ce modèle, conformément à la figure 2.10.

Il en résulte une reconsidération de la généralisation. Ici, elle est utilisée pour diminuer la taille du modèle des récompenses attendues, alors que les systèmes de classeurs précédents ne cherchaient à approximer aucune fonction. Ils cherchaient directement une solution au problème de la maximisation des récompenses attendues[11].

2.3.2 De Q-learning à XCS

Dans cette section, nous montrons comment il est possible de passer d'une représentation tabulaire de la fonction de qualité (comme dans le cas de l'algorithme *Q-learning*) à une représentation à base de classeurs offrant la possibilité de généraliser. Cette présentation est inspirée de Lanzi (2000).

Le tableau 2.3 montre un exemple de représentation tabulaire d'une fonction de qualité. Une telle représentation tabulaire peut être traduite en un ensemble de triplets situation-action-qualité.

Dans cette représentation, les règles correspondant à ces triplets situation-action-qualité sont représentées de manière exhaustive, comme dans le tableau 2.4. Dans cet exemple, nous considérons qu'il n'y a pas d'autres situations que celle qui sont présentées.

Pourtant, les systèmes de classeurs sont dévolus à des problèmes dans lesquels les situations se composent de plusieurs attributs représentant autant de propriétés perceptibles de l'environ-

[11]Les premiers systèmes de classeurs, ZCS inclus, sont dits « strength based » alors que ceux dérivant de XCS sont dits « accuracy based ».

Situation	Action	Qualité
[000111]	[1]	0.81
[101110]	[1]	0.81
[001100]	[1]	0.81
[110001]	[1]	0.72
[010011]	[1]	0.72

TAB. *2.4: Règles sans généralisation correspondant à la représentation tabulaire de la table 2.3. Seules les règles concernant l'action* [1] *sont indiquées.*

Condition	Action	Qualité
[#0####]	[1]	0.81
[#1####]	[1]	0.72

TAB. *2.5: Généralisation des trois premières règles de la table 2.4*

nement. Il est alors possible de diminuer le nombre de règles en utilisant des symboles #[12] ne faisant porter aucune contrainte sur la valeur de certains attributs.

Dans l'exemple de la table 2.4, trois règles proposant l'action [1] sont créditées d'une qualité de 0.81. Deux autres ont une qualité de 0.72 et font intervenir la même action. Dans chacune des trois premières règles, celles de qualité 0.81, le deuxième attribut vaut 0. Or, dans les deux dernières règles proposant la même action mais avec une qualité différente, cet attribut vaut 1. Le deuxième attribut permet donc à lui seul de discriminer les situations dans lesquelles l'action [1] a une qualité de 0.81 des situations dans lesquelles cette action a une qualité de 0.72.

En identifiant les attributs non pertinents par des symboles #[13], il est possible de représenter chaque groupe de situations avec une seule condition. Ainsi, dans l'exemple de la figure 2.5, les trois premières règles sont remplacées par un unique classeur `[#0####] [1] 0.81`, et les deux autres par le classeur `[#1####] [1] 0.72`.

En résumé, dans XCS, une prédiction `p` est associée à chaque classeur `Condition Action`. Elle prédit la récompense attendue quand l'action spécifiée par le classeur est entreprise dans chacune des situations appariées avec sa partie `Condition`. Ainsi, alors que le *Q-learning* tabulaire considère des triplets $(s, a, q) \in S \times A \times \Re$ pour modéliser la fonction $Q : S \times A \to \Re$, les systèmes de classeurs comme XCS considèrent des classeurs `Condition Action p`. La partie `Condition` peut contenir des symboles # et être appariée avec plusieurs situations, et la partie `Action` représente la commande sur les effecteurs. La prédiction `p` correspond, dans la terminologie utilisée dans le *Q-learning*, aux qualités associées aux couples situation-action. Pour maximiser la récompense attendue, quand il perçoit une situation, XCS sélectionne les classeurs dont la

[12]Grâce aux symboles #, certaines valeurs d'attributs peuvent être omises (voir section 2.1.1).
[13]Un symbole # est apparié avec n'importe quelle valeur particulière de l'attribut.

partie `Condition` est appariée avec la situation. Il choisit ensuite l'action du classeur sélectionné qui a la plus grande prédiction `p`.

Le problème de la généralisation dans un système de classeurs comme XCS consiste à trouver les parties `Condition` et `Action` de sorte que les symboles `#` soient bien placés. Pour ce faire, XCS utilise des algorithmes génétiques[14] pour faire évoluer une population de classeurs, conformément au style « Michigan »[15]. Chaque classeur est un individu qui est évalué tout au long de l'interaction de l'agent avec l'environnement. Pour cette évaluation, comme illustré par la figure 2.9, on lui associe :

- une prédiction `p` qui représente le cumul des récompenses immédiates et futures à partir des couples (s, a) couverts par le classeur ;
- une erreur `e` qui représente l'erreur estimée sur la prédiction p ;
- une valeur sélective `f` qui est fonction de l'erreur de prévision. Si l'erreur est grande, alors la valeur sélective est faible et inversement.

Les opérations de mise à jour de ces différents estimations et les modalités d'application de l'algorithme génétique sont détaillées par Butz et Wilson (2002).

Kovacs (1997) a montré que XCS découvre des classeurs fiables et aussi généraux que possible. Un classeur est dit fiable si, dans chaque situation appariée avec la condition, choisir l'action proposée mène toujours effectivement au même cumul de récompenses, c'est-à-dire si `p` prédit bien la qualité associée à tous les couples situation-action correspondant au classeur. Un classeur fiable a donc une erreur `e` faible et une valeur sélective `f` élevée. Les classeurs trop généraux et qui correspondent à des couples situation-action avec une qualité différente sont éliminés par l'algorithme génétique. En effet, un classeur apparié avec plusieurs couples (s, a) à différents niveaux de récompense attendue n'est pas fiable parce qu'il est trop général. Ainsi, XCS généralise en fonction de la prédiction de la récompense attendue et il utilise le critère de fiabilité pour sélectionner les classeurs adéquats.

De plus, comme l'ont montré Butz et Pelikan (2001), l'algorithme génétique de XCS favorise le maintien dans la population des classeurs les plus généraux. Ainsi, les classeurs fiables mais qui pourraient être plus généraux sont éliminés et XCS découvre des classeurs fiables à un niveau maximal de généralisation, c'est-à-dire des classeurs qui ne seraient plus fiables s'ils contenaient plus de symboles `#`. En outre, XCS tend vers des ensembles de classeurs complets, où chacun des couples (s, a) possibles est apparié avec au moins un classeur.

2.4 L'apprentissage latent dans les systèmes de classeurs

2.4.1 Apprentissage latent et généralisation

Dans la section 1.3.2, nous avons introduit l'architecture *Dyna* qui comprend un module d'apprentissage par renforcement et un module d'apprentissage latent. Si les systèmes de classeurs

[14] voir section 2.1

[15] voir section 2.1.3.0

décrits plus haut permettent d'introduire une généralisation pour l'apprentissage par renforcement direct, ils n'abordent pas la question de l'apprentissage latent. Nous nous proposons d'utiliser la généralisation telle qu'elle est définie dans les systèmes de classeurs de manière à rendre plus compact et intelligible le résultat de l'apprentissage d'un modèle des interactions entre l'agent et son environnement.

À chaque pas de temps, en plus d'une récompense, l'agent reçoit la situation résultant de sa dernière action. La conséquence d'une action n'est donc pas seulement la récompense associée, mais aussi la nouvelle situation perçue. Ces relations entre situations successives doivent pouvoir être modélisées d'une manière ou d'une autre. Dans ce cadre, il est possible d'utiliser un système de classeurs pour apprendre un modèle de son environnement, de manière à le pourvoir d'une capacité à anticiper les nouvelles situations. Ce modèle peut être utilisé dans une architecture de type *Dyna* pour accélérer l'apprentissage d'une politique.

Pour apprendre un tel modèle de la dynamique des interactions entre l'agent et son environnement, Holland (1990) a proposé une approche implicite. En étiquetant les messages internes, il est possible de spécifier si une action postée sur la liste des messages est une action effective, une anticipation ou un message interne quelconque. Apprendre à poster des messages identifiés comme des anticipations et reflétant de manière fiable les conséquences des actions, c'est apprendre un modèle de son environnement. Riolo (1990) a mis en application cette idée dans CFSC2. Toutefois, ce système est complexe et recourt à trois types de classeurs et trois forces différentes.

Plus simple et permettant de reproduire les mêmes expériences, ACS (Stolzmann, 1998) implémente la théorie de Hoffmann (1993) concernant le contrôle du comportement animal par l'anticipation. Ce système ajoute aux classeurs une partie `Effet` qui permet de rendre compte des changements perçus dans l'environnement. Sans cette partie supplémentaire, les systèmes de classeurs comme XCS ne peuvent qu'offrir une prédiction concernant une seule valeur (en l'occurrence, la valeur de la récompense attendue). Or, pour anticiper une situation, il faut être capable de prédire plusieurs valeurs, une par attribut des situations. La nouvelle partie `Effet` modélise les changements survenus dans l'environnement lorsque l'action spécifiée par le classeur est entreprise dans une situation appariée avec sa partie `Condition`. Les classeurs `Condition Action Effet` correspondent aux triplets (s_t, a_t, s_{t+1}) utilisés dans *DynaQ+* mais offrent une capacité de généralisation.

Dans la section suivante, nous décrivons le formalisme et le fonctionnement d'ACS. Nous ne détaillons ici que les mécanismes utilisés pour l'apprentissage latent. L'apprentissage d'une politique et la sélection de l'action optimale seront abordés dans le chapitre 5.

2.4.2 ACS

2.4.3 Des heuristiques plutôt que des algorithmes génétiques

Étant inspiré de théories psychologiques, ACS a la particularité, par rapport à d'autres systèmes de classeurs, de ne pas utiliser d'algorithmes génétiques pour la construction de classeurs fiables, mais d'utiliser des heuristiques[16]. Spécialiser, c'est introduire une distinction supplémentaire en remplaçant une valeur `#` dans une partie `Condition` par une valeur particulière. ACS réalise des spécialisations explicites, pour implémenter la notion de différenciation introduite dans la section 1.1.2. Ces spécialisations explicites sont guidées par l'expérience au lieu de reposer des mutations et des croisements aléatoires.

2.4.4 Le formalisme d'ACS

Comme nous l'avons dit précédemment, pour réaliser l'apprentissage latent du modèle de l'environnement, ACS adjoint une partie `Effet` aux classeurs, pour permettre d'anticiper les conséquences des actions et construire un modèle de l'environnement. Ainsi, pour modéliser l'ensemble des triplets (s_t, a_t, s_{t+1}), ACS utilise des classeurs comprenant trois parties :

- une partie `Condition` qui joue le même rôle que les parties `Condition` dans d'autres systèmes de classeurs comme XCS : elle spécifie le domaine d'application du classeur en utilisant des symboles `#` pour permettre de généraliser ;
- une partie `Action` qui spécifie l'action proposée par le classeur ;
- une partie `Effet` qui décrit les changements intervenant dans l'environnement lorsque l'action spécifiée par le classeur est entreprise dans n'importe laquelle des situations appariée avec la partie `Condition` du classeur.

Deux forces sont associées à chaque classeur :

- une force `fa` représentant la qualité de l'anticipation. Sa mise à jour est décrite dans la section suivante ;
- une force `fr` associant une récompense attendue à chaque classeur. Elle est mise à jour comme le sont les qualités dans le *Q-learning* (voir section 1.2.3), ou les prédictions dans XCS(voir section 2.3.1). Elles sont utilisées pour l'apprentissage d'une politique, et n'influent pas sur l'apprentissage latent décrit ici.

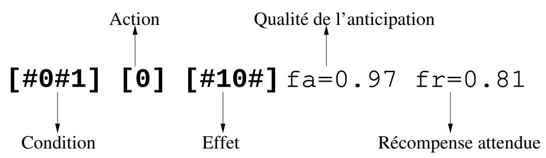

FIG. *2.11: Exemple de classeur dans ACS.*

Les classeurs ne sont donc plus des classeurs `Condition Action p` mais des classeurs `Condition Action Effet fe fr`. Ainsi, un classeur devient capable de prédire les nouvelles valeurs de plusieurs attributs. La figure 2.11 donne un exemple d'un tel classeur. Si les effets spécifiés par un classeur sont souvent effectivement observés, alors sa qualité d'anticipation sera grande. Le mécanisme d'apprentissage latent d'ACS sélectionne ces classeurs et permet donc de construire

[16]ACS n'utilise pas d'algorithmes génétiques dans sa première version mais les extensions de Butz en introduisent pour la généralisation. Ces extensions sont décrites par Butz et al. (2000a).

un modèle fiable de la dynamique des interactions entre l'agent et son environnement.

Les parties `Effet` sont composées d'autant d'attributs que les situations et peuvent contenir des symboles `#` en plus des valeurs particulières que peuvent prendre les attributs dans les différentes situations possibles. Un symbole `#` dans une partie `Effet` indique que l'attribut de la situation lui correspondant demeure inchangé lorsque l'action proposée est choisie. Si la valeur d'un attribut de la partie `Condition` n'est pas `#`, alors le classeur anticipe la valeur en question pour cet attribut, dans toutes les situations appariées avec la partie `Condition`.

Étant donné une situation s_t et une action a_t, un classeur peut donc anticiper la nouvelle situation s_{t+1} si sa partie `Condition` est appariée à s_t, et si sa partie `Action` est appariée à a_t. La situation anticipée est obtenue en appliquant la fonction *passthrough* à chaque attribut. Pour le $i^{ème}$ attribut, l'opérateur *passthrough* est défini ainsi :

$$passthrough(s_t^i, e^i) = \begin{cases} s_t^i & si\ e^i = \texttt{\#} \\ e^i & sinon \end{cases}$$

où s_t^i est le $i^{ème}$ attribut de la situation s_t et où e^i est le $i^{ème}$ attribut de la partie `Effet` du classeur.

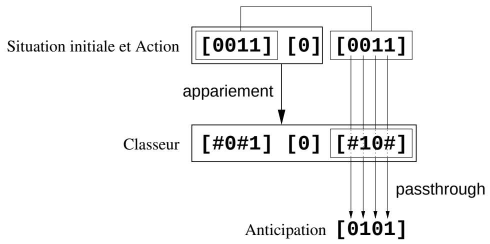

FIG. *2.12: Mécanisme d'anticipation d'un classeur dans ACS.*

Considérons par exemple le classeur `[#0#1] [0] [#10#]` de la figure 2.12. Grâce aux symboles `#` dans sa partie `Condition`, il anticipe les conséquences de l'action `[0]` dans 4 situations possibles : `[0001]`, `[0011]`, `[1001]` et `[1011]`. Selon la partie `Effet`, en appliquant l'opérateur *passthrough* :

- le premier attribut demeure inchangé quelle que soit la valeur initiale (`0` ou `1` à cause du symbole `#` dans la partie `Condition`) ;
- la valeur du deuxième attribut passera de `0` à `1` ;
- le troisième attribut prendra la valeur `1` quelle que soit la valeur initiale ;
- la valeur du dernier attribut reste `1`.

Ainsi, dans la situation `[0001]`, si l'action `[0]` est choisie, le classeur anticipe la situation `[0101]`.

Pour anticiper la situation suivante lorsqu'une action a_t est entreprise dans une situation s_t, ACS sélectionne le classeur avec la plus grande qualité d'anticipation `fa` parmi tous les classeurs dont la partie `Condition` est appariée à s_t et dont la partie `Action` est appariée à a_t. En appliquant la fonction *passthrough* à s_t et à sa partie `Effet`, ce classeur propose une anticipation. Celle-ci est la nouvelle situation prédite par le système. Si l'anticipation proposée

Transition	[0110] [0]	[1010]
Cas 1	[0###] [0] [10##]	[1010] est la situation anticipée. La force `fa` augmente.
Cas 2	[##10] [0] [##10]	[0110] est la situation anticipée. Les deux attributs changeants sont spécialisables donc le classeur est corrigeable. Il devient [0110] [0] [1010].
Cas 3	[01##] [0] [0110]	[0110] est la situation anticipée. La force `fa` diminue.

TAB. *2.6: L'ALP d'ACS*

par un classeur est égale à la situation suivante s_{t+1} réellement observée, alors le classeur anticipe bien. Dans le cas contraire, un mécanisme d'apprentissage latent permet de modifier les qualités d'anticipation `fa`, et de corriger l'erreur pour améliorer le modèle de l'environnement.

L'ALP d'ACS

Dans ACS, un mécanisme d'apprentissage indépendant de la récompense permet de découvrir un ensemble de classeurs permettant d'anticiper correctement dans chaque couple situation-action. Ce mécanisme est nommé ALP (Stolzmann, 1998; Butz, 2002a) (Anticipatory Learning Process), c'est une implémentation des théories de Hoffman concernant l'apprentissage latent dans le contrôle du comportement animal par l'anticipation.

L'ALP sélectionne, parmi tous les classeurs possibles, ceux qui anticipent bien. À chaque classeur est donc associée une force `fa` qui représente la capacité du classeur à anticiper correctement. Les classeurs dotés d'une faible force `fa` sont supprimés.

ACS doit aussi être en mesure de trouver des classeurs dont les parties `Condition` sont aussi générales que possible, tout en permettant de modéliser de façon fiable la dynamique des interactions de l'agent avec l'environnement[17]. Lorsqu'un classeur n'est pas fiable, l'ALP introduit des distinctions supplémentaires dans sa partie `Condition` pour créer un nouveau classeur.

Pour mettre à jour les forces `fa` et créer de nouveaux classeurs, ACS considère tour à tour les triplets successifs (s_{t-1}, a_{t-1}, s_t). À chaque pas de temps, ACS sélectionne le classeur doté de la plus grande force `fa` parmi tous les classeurs dont la partie `Condition` est appariée avec s_{t-1}, donc la partie `Action` est appariée avec a_{t-1}. En utilisant la fonction *passthrough*, ce classeur propose une anticipation en fonction de s_{t-1} et de sa partie `Effet` (voir section précédente). L'ALP supprime les classeurs qui anticipent mal.

Le mécanisme d'apprentissage est illustré par le tableau 2.6. Plus généralement, si l'anticipation du classeur sélectionné est identique à la nouvelle situation réellement observée s_t, deux

[17]Comme dans XCS, un classeur est dit aussi général que possible s'il ne peut contenir aucun symbole # supplémentaire sans perdre en fiabilité.

cas de figure se présentent :

- si $s_{t-1} \neq s_t$ alors des changements ont été observés et bien prédits : la qualité du classeur augmente. C'est le cas 1 de la figure 2.6 ;
- par contre, si $s_{t-1} = s_t$ alors aucun changement n'a été perçu dans la situation et la qualité du classeur diminue, alors même qu'il avait bien anticipé. En effet, ACS ne cherche pas à modéliser les transitions qui ne provoquent aucun changement dans la situation perçue.

Si l'anticipation d'un classeur ne correspond pas à la nouvelle situation alors, là aussi, deux cas de figure se présentent. Considérons les attributs de la nouvelle situation dont la valeur est différente dans l'anticipation proposée par le classeur. Ces attributs sont les attributs dont la valeur a changé :

- si tous les attributs changeants sont des symboles `#` dans la partie `Condition` et `Effet`, alors le classeur peut être corrigé. On effectue une spécialisation en modifiant le classeur : les attributs changeants de la partie `Condition` prennent les valeurs spécifiques qui leur correspondent dans s_{t-1}. Les attributs correspondant dans la partie `Effet` prennent leur nouvelle valeur dans s_t. C'est le cas 2 de la figure 2.6 ;
- s'il n'est pas possible de spécialiser le classeur de la sorte, la qualité du classeur diminue. C'est le cas 3 de la figure 2.6.

Lorsque la qualité d'anticipation d'un classeur est très basse, le classeur est supprimé de la liste des classeurs. Ainsi, seuls les classeurs anticipant un changement de manière fiable restent dans la liste. Ici, un classeur est fiable s'il anticipe correctement la situation suivant l'action qu'il spécifie dans toutes les situations appariées à sa partie `Condition`. Ainsi, dans ACS, la fiabilité d'un classeur dépend de sa capacité à prévoir une situation, plutôt qu'une récompense attendue, comme c'est le cas dans XCS. ACS généralise donc en fonction des changements de situation alors que XCS généralise en fonction de la récompense.

Autres mécanismes d'ACS

D'autres mécanismes ont été ajoutés à ACS par Stolzmann (1999) et par Butz (2002a), de manière à remédier aux problèmes de sur-spécialisation et de sur-généralisation de la version originale d'ACS.

Quand ACS spécialise des attributs de la partie `Condition`, il en spécialise beaucoup en même temps. Il en résulte un problème de sur-spécialisation que Butz corrige en incluant un mécanisme de généralisation fondé sur des algorithmes génétiques.

En ce qui concerne la sur-généralisation, l'ALP d'ACS ne spécialise un attribut dans une partie `Condition` que lorsque la valeur de cet attribut change dans la transition observée (voir le cas 2 de la figure 2.6). Or, il se peut qu'il soit plus intéressant de spécialiser un attribut qui, pourtant, n'a pas changé lors de la dernière transition. En outre, une telle spécialisation peut être indispensable à une modélisation correcte. C'est pourquoi Butz a ajouté à ACS un mécanisme de spécialisation des attributs qui ne changent pas (Butz *et al.*, 2000a), de manière à pouvoir spécialiser une partie `Condition` sans spécialiser la partie `Effet`.

Ces deux problèmes sont discutés plus en détail lors de la comparaison entre ACS et YACS, dans la section 3.4.

Spécialisation et effets

Dans ACS, on utilise le terme de « spécialisation » quand les parties `Effet` sont modifiées. Pourtant, seules les parties `Condition` ont une influence sur le domaine de validité du classeur. Certains classeurs plus généraux sont alors applicables dans un plus grand nombre de situations que d'autres. La partie `Effet` n'a donc aucune influence sur le domaine de validité et sa modification ne devrait pas être qualifiée de spécialisation, et encore moins conditionner une spécialisation de la partie `Condition`.

En fait, ACS utilise le même symbole spécial `#` pour les parties `Condition` et `Effet` alors que ce symbole a une sémantique tout à fait différente selon la partie du classeur où elle apparaît. Dans une partie `Condition`, `#` représente un joker[18], alors que, dans une partie `Effet`, `#` représente une absence de changement[19]. Dans YACS, notre premier système décrit au chapitre suivant, nous utilisons le même formalisme qu'ACS, mais nous préférons insister sur cette différence sémantique en utilisant des symboles différents dans les parties `Condition` et `Effet`, de manière à éviter toute confusion.

2.5 Discussion

2.5.1 Utilisation de systèmes à base de règles pour la généralisation

Dans ce chapitre, nous avons décrit le formalisme des systèmes de classeurs, et nous avons montré comment il permet de généraliser en exploitant des régularités dans la dynamique des interactions entre l'agent et son environnement. Cette généralisation permet d'éviter d'utiliser une liste de tous les cas possibles pour représenter une fonction de qualité, une fonction de transition ou une politique. Le modèle résultant gagne donc en compacité.

Il existe d'autres représentations qui proposent une généralisation, comme les réseaux de neurones formels ou les arbres de décision.

Un réseau de neurones formels comme celui de la figure 2.13 est caractérisé par une structure et un ensemble de poids. Le nombre de ces paramètres est donc fixe. Un tel réseau permet de calculer une sortie pour un nombre arbitraire d'entrées. De plus, il peut proposer une solution dans des cas encore jamais rencontrés par le système. Un réseau de neurones formels offre donc une capacité de généralisation. Si ces solutions peuvent être efficaces, leur paramétrage peut être délicat et le modèle appris manque d'intelligibilité.

L'efficacité n'est en effet pas le seul critère rentrant en ligne de compte pour le choix d'un modèle. On peut également vouloir structurer les données issues de la boucle sensori-motrice de manière à ce qu'elles soient plus facilement intelligibles par des experts humains. Dès lors, il

[18] symboles «*don't care*»
[19] symboles «*don't change*»

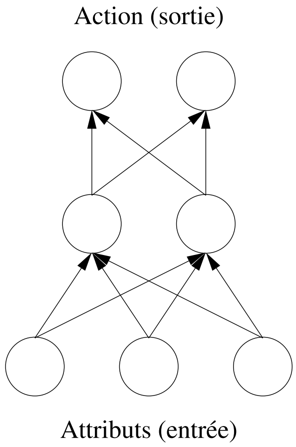

FIG. *2.13: Exemple simple de réseau de neurones formels.* Les réseaux de neurones formels sont caractérisés par une structure et un ensemble de poids attachés à chaque connexion (les flèches) entre deux neurones (les cercles). Dans cet exemple, la structure du réseau comporte trois couches, et chaque neurone est connecté à tous les neurones de la couche suivante. En fonction de valeurs d'entrée, le système utilise les poids pour calculer les activations de chaque neurone et les valeurs de sortie. L'apprentissage modifie les poids de manière à ce que la sortie calculée soit conforme à ce que l'on attend.

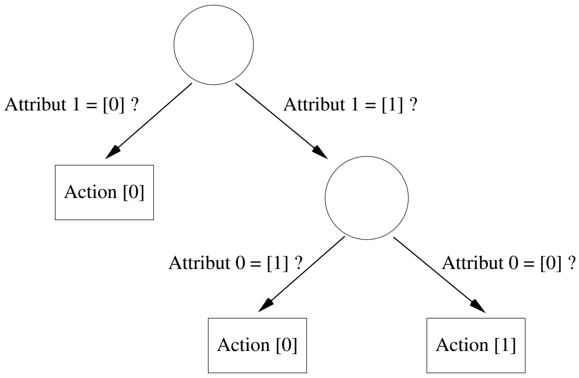

FIG. *2.14: Exemple simple d'arbre de décision.* Un tel arbre se lit du haut vers le bas. Il est possible de descendre vers un nœud de niveau inférieur dès que les entrées considérées vérifient la condition associée à la flèche qui y va. Les feuilles de l'arbre indiquent la décision. Pour cet arbre, la situation `[01]` provoque l'action `[0]`.

devient intéressant de se tourner vers d'autres types de formalismes. Parmi les alternatives aux représentations connexionnistes, les plus populaires sont les arbres de décision et les systèmes à base de règles.

L'arbre de décision de la figure 2.14 est fonctionnellement équivalent à l'ensemble de règles suivant :

- `[0#] [0]`
- `[10] [0]`
- `[11] [1]`

Dans le cas où le premier attribut a pour valeur `0`, il n'est pas nécessaire de contrôler la valeur du deuxième attribut ; dans tous les cas, l'action proposée est `[0]`. Cet arbre de décision permet donc de généraliser.

Mais la structure arborescente est contraignante et elle manque de souplesse. Or dans ce cas d'un apprentissage incrémental, il est souvent nécessaire de reconsidérer des éléments appris plus tôt. Or ceci peut être difficile lorsqu'une structure arborescente contraint le modèle construit. Pour reconsidérer une distinction qui apparaît au sommet d'un arbre, par exemple, il faut reconsidérer toute la structure de l'arbre. Souvent, la construction d'un arbre de décision, comme dans les travaux de Mc Callum (1995), ajoute progressivement des distinctions supplémentaires, sans retour en arrière possible.

Les systèmes à base de règles, tout en restant intelligibles, ne sont pas contraints par une structure arborescente. Ils offrent en outre une flexibilité appréciable pour l'apprentissage incrémental.

2.5.2 Systèmes de classeurs *vs.* Dyna pour apprentissage latent

Dans la section 1.3.2, nous avons présenté l'architecture *Dyna*, qui construit un modèle de l'environnement pour accélérer l'apprentissage par renforcement. *DynaQ+* apprend un tel modèle en constituant une liste de triplets (s_t, a_t, s_{t+1}). ACS réalise le même type d'apprentissage latent, mais en introduisant de la généralisation dans le modèle.

Si les situations perçues ne sont pas composées d'attributs, comme c'est le cas dans les expériences concernant *DynaQ+*, il n'y a aucun moyen de généraliser et un système de classeurs à anticipation comme ACS ne pourrait pas profiter de ses mécanismes spécifiques. Il construirait alors un ensemble de classeurs sans aucune généralisation, en se bornant à énumérer toutes les transitions rencontrées, de même que *DynaQ+* constitue une liste exhaustive de tous les triplets (s_t, a_t, s_{t+1}).

Si les situations sont composées d'attributs, un système sans généralisation, comme *DynaQ+*, identifie les différents états par leurs attributs, mais il ne peut pas utiliser cette particularité et il constitue une liste exhaustive de triplets. Par contre, un système de classeurs peut mettre à profit certaines régularités pour introduire de la généralisation et réduire le nombre de règles. L'identification de régularités permet aussi de proposer une anticipation dans des situations encore jamais rencontrées. Par exemple, quand un système de classeur comporte une règle indiquant qu'un mouvement dans une direction ne provoque aucun changement dès lors qu'il y a un obstacle dans cette direction, ce classeur est valide dans un grand nombre de situations, y compris dans des situations que le système n'aurait pas encore rencontrées.

En outre, la généralisation dans les systèmes à base de règles permet d'améliorer la lisibilité du modèle construit. Par exemple, si on cherche à comprendre comment l'agent anticipe les conséquences d'un mouvement vers un obstacle, il suffit de consulter un seul classeur alors que dans le cas de *DynaQ+*, il est nécessaire de consulter tous les triplets faisant intervenir une situation dont la description fait état d'un mur dans le direction considérée.

Donc, les systèmes de classeurs à anticipation abordent le problème de la construction d'un modèle de l'environnement compact et intelligible. Dans une architecture *Dyna*, ce type de modèle peut remplacer avantageusement un modèle constitué d'une liste exhaustive de triplets. Par exemple, Butz et al. (2002b) propose une architecture *Dyna* faisant intervenir des systèmes de classeurs différents. Le premier, ACS, permet d'apprendre un modèle de l'environnement. Il est utilisé pour accélérer la convergence d'un deuxième système de type XCS, qui apprend une politique.

Dans ce chapitre, nous avons décrit le formalisme des systèmes de classeurs, et nous avons montré comment ces systèmes exploitent des régularités dans la dynamique des interactions entre l'agent et son environnement, pour généraliser les conséquences des actions dans plusieurs situations. Nous avons introduit ACS, qui utilise ce type de formalisme de manière à réaliser un apprentissage latent qui peut prendre place dans une architecture *Dyna*. Son originalité, par rapport à d'autres systèmes de classeurs qui utilisent des algorithmes génétiques, est l'utilisation d'heuristiques pour construire un modèle de l'environnement. Dans le chapitre suivant, nous proposons un premier système, YACS, qui utilise le même formalisme qu'ACS mais des heuristiques différentes, qui permettent d'améliorer la vitesse de l'apprentissage latent.

Chapitre 3

YACS : apprentissage latent et généralisation avec des heuristiques

YACS (Gérard et Sigaud, 2001) utilise le même formalisme qu'ACS, et les deux systèmes créent des classeurs explicites de la forme `Condition Action Effet` pour modéliser l'ensemble des transitions (s_t, a_t, s_{t+1}). L'adjonction d'une partie `Effet` composée de plusieurs attributs permet aux classeurs de prédire une valeur pour chacun d'eux. Cette partie supplémentaire décrit les changements perçus sur tous les attributs lorsqu'une action est effectuée dans certaines situations.

La différence entre ACS et YACS réside dans les mécanismes de découverte des classeurs appropriés à la modélisation de la dynamique des interactions entre l'agent et son environnement. Nous en proposons de nouveaux, plus efficaces.

Une discussion sur les différences entre YACS et ACS est présentée à la fin de ce chapitre, dans la section 3.4. Nous décrivons le formalisme de YACS dans la section 3.1 et dans la section 3.2, nous décrivons ses mécanismes. Dans l'étude expérimentale présentée dans la section 3.3, nous reproduisons des expériences déjà menées dans la cadre d'ACS, de manière à démontrer un net gain en vitesse d'apprentissage par rapport à ACS.

3.1 Le formalisme et l'architecture de YACS

La figure 3.1 donne un exemple de classeur `Condition Action Effet` avec toutes les informations qui lui sont associées dans YACS. Ces informations seront introduites tout au long de la description des mécanismes de YACS, présentés dans la section 3.2.

La partie `Effet` d'un classeur représente les conséquences de l'action dans les situations appariées avec la partie `Condition`. Elle représente les changements perçus dans l'environnement. Dans ACS et dans YACS, comme dans les autres systèmes de classeurs, une partie `Condition` est un vecteur d'attributs qui peuvent prendre la valeur `#` en plus des valeurs permises dans les situations. Une partie `Effet` est aussi composée de plusieurs attributs, qui peuvent également prendre une valeur spéciale. A la différence des parties `Condition`, cette valeur spéciale n'est

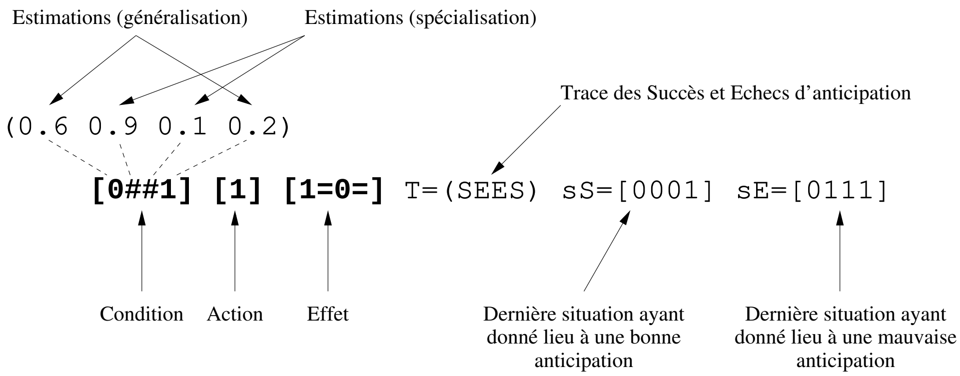

FIG. *3.1: Exemple de classeur dans YACS.* Les estimations sont utilisées dans les mécanismes de spécialisation et de généralisations décrits dans les sections 3.2.5 et 3.2.6. Les situations s_S et s_S sont utilisées pour mettre à jour les estimations pour la spécialisation (voir section 3.2.5). La trace T est introduite dans la section 3.2.1

Classeur		
`[#0#1]`	`[0]`	`[=10=]`
`[0001]`	$\rightarrow$	`[0101]`
`[0011]`	$\rightarrow$	`[0101]`
`[1001]`	$\rightarrow$	`[1101]`
`[1011]`	$\rightarrow$	`[1101]`
Situations initiales		Situations anticipées

TAB. *3.1: Un classeur modélisant transitions.*

pas `#` mais `=`. Un tel symbole indique que l'attribut de la situation correspondant au symbole `=` demeure inchangé lorsque l'action proposée est choisie. Une valeur particulière dans la partie `Effet` signifie que la valeur de l'attribut correspondant change et prend la valeur indiquée, quelle que soit la partie `Condition`. Comme ACS, YACS utilise la fonction *passthrough* pour produire des anticipations. Cet opérateur est défini de manière similaire à celui d'ACS :

$$passthrough(s_t^i, e^i) = \begin{cases} s_t^i & si\ e^i = \text{« = »} \\ e^i & sinon \end{cases}$$

où s_t^i est le $i^{ème}$ attribut de la situation s_t et où e^i est le $i^{ème}$ attribut de la partie **`Effet`** du classeur.

Considérons par exemple le classeur `[#0#1] [0] [=10=]` du tableau 3.1. Grâce aux symboles `#` dans sa partie `Condition`, il anticipe les conséquences de l'action `[0]` dans 4 situations possibles (`[0001]`, `[0011]`, `[1001]` et `[1011]`). Selon la partie `Effet` :

- le premier attribut demeure inchangé quelle que soit sa valeur initiale (`0` ou `1` à cause du symbole `#` dans la partie `Condition`) ;
- la valeur du deuxième attribut changera de `0` à `1` ;

– le troisième attribut prendra la valeur `0` quelle que soit sa valeur initiale.
– la valeur du dernier attribut reste `1`.

Ce formalisme permet aux classeurs de représenter des régularités dans les interactions avec l'environnement, comme par exemple *« dans un monde de cases, quand l'agent perçoit un mur au nord, quoi qu'il perçoive dans les autres directions, se déplacer vers le nord le conduit à rester dans la même case (il heurte le mur), donc aucun changement ne sera perçu dans sa situation ».*

Dans YACS, la généralisation est possible grâce à l'utilisation simultanée des symboles `#` et `=`. Elle permet de représenter des régularités dans les transitions entre les situations successives et elle permet au système :

– de sélectionner les attributs nécessaires pour distinguer les couples situation-action suivis d'effets différents ;
– de spécifier un ensemble de situations en utilisant une seule condition, ce qui permet de réduire la taille du modèle de l'environnement.

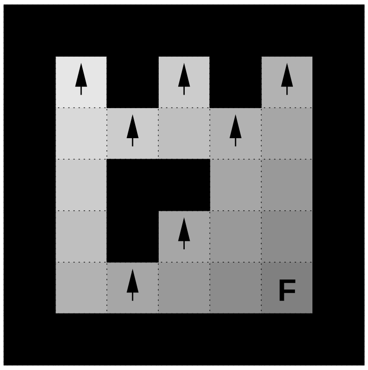

FIG. *3.2: Généralisation et niveaux de récompense attendue.* Ici, les actions représentées par les flèches conduisent toutes à heurter le mur, et aucun changement de situation n'est perçu.

Les mêmes changements perçus peuvent être observés à partir de différentes situations, indépendemment de la récompense associée. Dans l'exemple de la figure 3.2, YACS peut représenter toutes les transitions indiquées par des flèches, avec un seul classeur spécifiant que, s'il y a un mur au nord, alors un mouvement dans cette direction ne change rien à la situation perçue. Un tel classeur est fiable puisque, dans YACS, la fiabilité ne dépend que de la capacité à prévoir les changements de situation. Pourtant, ces actions mènent chacune à des situations avec des récompenses attendues différentes, représentées par des niveaux de gris différents. Il n'est donc pas possible d'associer aux classeurs une valeur concernant la récompense attendue[20]. Il convient donc d'enregistrer les informations concernant la récompense dans un module séparé de la liste

[20]Pourtant, dans la version d'ACS initialement proposée par Stolzmann, une récompense attendue `fr` était associée à chaque classeur pour permettre un apprentissage par renforcement (voir section 2.4.4). Si Stolzmann a tout de même obtenu des résultats en effectuant un apprentissage par renforcement de cette manière, c'est parce qu'ACS crée des classeurs trop spécialisés, si bien que les interférences entre généralisation par rapport à la situation, et par rapport à la récompense, ne se produisent pas.

des classeurs.

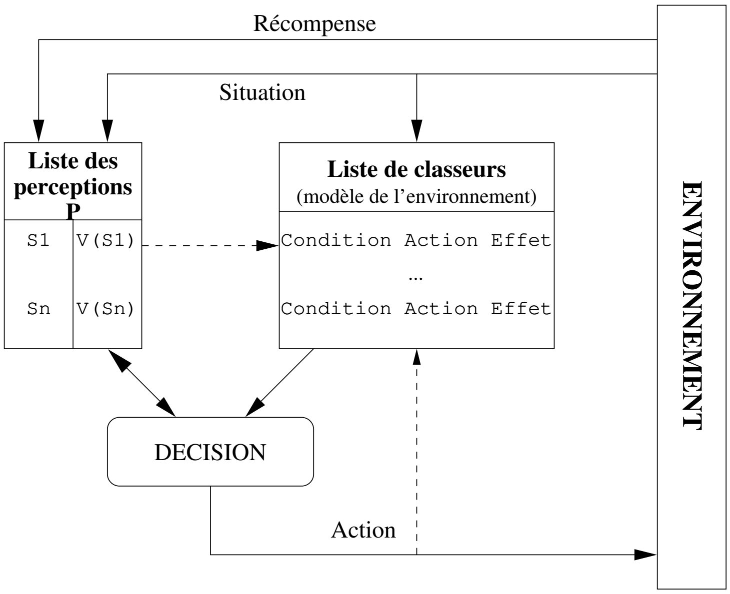

FIG. *3.3: Architecture de YACS.* L'apprentissage du modèle de l'environnement, constitué de classeurs, n'utilise que les informations concernant les situations et actions successives. Il n'utilise pas la récompense. Les récompenses attendues $V(s_i)$ sont associées à chaque situation dans une liste de perceptions `P`. Un module de décision permet d'apprendre une politique. Les mécanismes de généralisation et de spécialisation de YACS utilisent la liste des perceptions, mais jamais les valeurs associées.

Pour apprendre une politique, conformément à l'architecture *Dyna* illustrée par la figure 1.11, YACS conserve donc dans un module séparé les informations sur la valeur de récompense attendue à partir de chaque situation. Pour ce faire, YACS utilise une liste contenant chaque situation s_i perçue pendant l'expérience de l'agent, et la récompense attendue $V(s_i)$ qui lui est associée (voir figure 3.3). Ces valeurs $V(s_i)$ correspondent aux valeurs $V(s)$ de l'équation 1.2. Pour choisir une action dans une situation donnée, YACS utilise son modèle de l'environnement pour anticiper les situations possibles après chaque action possible. L'action choisie est celle qui permet d'atteindre la situation dans laquelle récompense attendue sera la plus grande.

La liste de perceptions est notée `P`. Elle contient un seul exemplaire de chacune des situations perçues[21]. Ainsi, au début de chaque pas de temps, la situation perçue s_t est ajoutée à cette liste si elle n'y est pas déjà présente.

La liste `P` des situations perçues n'utilise pas de généralisation et sa taille est proportionelle la taille des environnements. Si l'environnement est vaste, elle peut devenir très longue. Toutefois,

[21]Cette liste est moins grande que celle de toutes les situations théoriquement possibles, en fonction des valeurs que peuvent prendre chacun des attributs. Dans un grand problème multi-agents comme celui décrit dans Sigaud et Gérard (2001), par exemple, le nombre de situations effectivement perçues est de 290, alors que le nombre de situations théoriquement possibles est de 8192

elle permet de ne pas utiliser une table de tous les couples état-action possibles, et leurs qualités associées. Butz et al. (2002b) proposent d'utiliser XCS pour réduire la taille du modèle des récompenses attendues.

3.2 La stratégie de YACS pour la découverte de nouveaux classeurs

Dans les sections suivantes, nous présentons brièvement les principaux mécanismes de YACS relatifs à l'apprentissage latent, tels qu'ils ont été présentés par ailleurs (Gérard et Sigaud, 2001) et (Gérard, 2001). Contrairement à ce qui est fait dans ACS, les parties `Effet` et les parties `Condition` sont modifiées par des mécanismes indépendants. En particulier, la spécialisation des parties `Condition` n'est plus liée à aucune modification des parties `Effet`. Ainsi, la découverte des parties `Condition` et `Effet` est plus décorrélée dans YACS qu'elle ne l'est dans ACS.

3.2.1 Évaluation des classeurs

Pour évaluer les classeurs, YACS met à jour une trace T de marqueurs booléens *succès* et *échec* associée à chaque classeur (voir figure 3.1). Elle mémorise les succès et les erreurs d'anticipation de chaque classeur. Cette trace T fonctionne comme une file de longueur finie θ_e. Dès qu'elle est pleine, les marqueurs *succès* et *échec* les plus anciens sont écartés pour placer les plus récents.

YACS dispose à tout instant de la dernière perception courante s_t résultant de la dernière action choisie a_{t-1} dans la dernière situation s_{t-1}. Grâce à cette information, YACS détermine l'effet réel (noté `ER`) qui représente exactement les changements intervenus dans d'environnement entre les deux derniers pas de temps. Cet effet réel correspond donc à la partie `Effet` d'un classeur qui aurait correctement anticipé au pas de temps précédent.

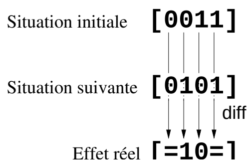

FIG. *3.4: Calcul de l'effet réel.*

Comme une partie `Effet` ordinaire, `ER` est composée d'attributs. La valeur de l'attribut de rang i dans `ER` est déterminée en fonction des valeurs des $i^{èmes}$ attributs s_{t-1}^i et s_t^i correspondant dans s_{t-1} et s_t. Cette valeur est donnée par la fonction *diff* définie comme suit entre deux attributs, et illustrée par la figure 3.4 :

$$diff(s_t^i, s_{t-1}^i) = \begin{cases} = & si\ s_t^i = s_{t-1}^i \\ s_t^i & sinon \end{cases}$$

Transition			Effet réel
s_{t-1}	a_{t-1}	s_t	$diff(s_t, s_{t-1})$
`[0110]`	`[0]`	`[1010]`	`[10==]`
Classeur			Évaluation
`[1###]`	`[0]`	`[==10]`	Non évalué
`[0###]`	`[0]`	`[10==]`	Ajout d'un marqueur S
`[##1#]`	`[0]`	`[==10]`	Ajout d'un marqueur E

TAB. *3.2: Évaluation des classeurs*

Appliquée à chaque paire d'attributs de même rang dans s_{t-1} et s_t, cette fonction permet de calculer chacun des attributs de `ER` conformément aux éléments de formalisme illustrés par le tableau 3.1.

À chaque pas de temps, YACS vérifie l'exactitude de la partie `Effet` de chacun des classeurs qui aurait pu être sollicité pour fournir une anticipation. Ces classeurs sont tels que leur partie `Condition` est appariée avec s_{t-1} et leur partie `Action` est égale à a_{t-1}. La partie `Effet` d'un tel classeur devrait être `ER`. Ainsi, comme illustré par le tableau 3.2,

- si sa partie `Effet` est égale à `ER`, alors le classeur a bien anticipé et un marqueur de succès S est ajouté à la trace T de marqueurs booléens du classeur considéré ;
- si sa partie `Effet` est différente de `ER`, alors le classeur a mal anticipé et un marqueur d'échec E est ajouté à la trace T.

3.2.2 Couverture des effets

Avec s_t et s_{t-1}, la fonction *diff* permet de calculer un effet réel `ER`. Si aucun des classeurs qui ont une partie `Condition` et une partie `Action` appariées à s_{t-1} et s_{t-1} n'ont leur partie `Effet` égale à `ER`, il convient d'en créer un qui représente la dernière transition. Dans ce cas, un nouveau classeur est créé tel que sa partie `Effet` est égale à `ER`, sa partie `Action` est a_{t-1}, et sa partie `Condition` est choisie aléatoirement parmi les parties `Condition` des classeurs qui ont mal anticipé et qui ont reçu un marqueur E.

Dans l'exemple du tableau 3.3, le système de classeurs ne contient que quatre classeurs dont l'action est appariée à a_{t-1}. Parmi ces quatre classeurs, seuls deux ont une partie `Condition` adéquate, mais aucun ne propose un effet conforme à l'effet réel. Un nouveau classeur est donc créé : `[0#1#] [0] [10==]`.

3.2.3 Couverture des conditions

Étant donné une des actions possibles, lorsque YACS reçoit de l'environnement une nouvelle situation s_t, il peut arriver que, parmi tous les classeurs dont la partie `Action` est appariée à cette action, aucun n'ait une partie `Condition` appariée à la situation précédente s_{t-1}. Dans ce cas,

Transition			Effet réel
s_{t-1}	a_{t-1}	s_t	$diff(s_t, s_{t-1})$
[0110]	[0]	[1010]	[10==]
Classeurs			**Évaluation**
[0#1#]	[0]	[==10]	Échec
[0#1#]	[0]	[1==0]	Échec
[0#0#]	[0]	[0=10]	Non évalué
[1###]	[0]	[=110]	Non évalué
[0#1#]	[0]	[10==]	Création

TAB. *3.3: Couverture des effets*

Transition			Effet réel
s_{t-1}	a_{t-1}	s_t	$diff(s_t, s_{t-1})$
[0110]	[0]	[1010]	[10==]
Classeurs			**Évaluation**
[0#0#]	[0]	[0=10]	Non apparié
[1###]	[0]	[=110]	Non apparié
[##1#]	[0]	[10==]	Création

TAB. *3.4: Couverture des conditions*

le système de classeurs est incapable de prédire les conséquences de l'action considérée dans la situation s_{t-1}. Dès lors, un nouveau classeur est créé pour combler ce manque. La partie `Action` de ce nouveau classeur est égale à l'action considérée. Sa partie `Effet` est calculée en fonction de s_{t-1} et de s_t, grâce à la fonction *diff*, de la même manière que l'on calcule l'effet réel `ER`. Sa partie `Condition` doit être appariée avec s_{t-1}. Pour éviter toute redondance, il est aussi souhaitable que cette condition ne soit ni strictement plus générale ni plus spécialisée que celle d'aucun autre classeur doté de la même partie `Action`. YACS construit une nouvelle partie `Condition` aussi générale que possible, sous réserve que les contraintes précédentes soient satisfaites.

Dans l'exemple du tableau 3.4, le système de classeurs ne contient que deux classeurs dont l'action est appariée à a_{t-1}. Parmi ces deux classeurs, aucun n'a une partie `Condition` appariée avec s_{t-1}. Un nouveau classeur est donc créé pour couvrir cette situation : `[##1#] [0] [10==]`. La partie `Condition` de ce nouveau classeur est appariée à s_{t-1}, et elle contient autant de symboles `#` que possible, sans être strictement plus générale qu'une autre partie `Condition`. Le classeur `[0###] [0] [10==]` a le même nombre de symboles `#`, mais sa partie `Condition` aurait été strictement plus générale que celle du premier classeur de la liste. Il n'est donc pas retenu et le classeur `[##1#] [0] [10==]` lui est préféré.

3.2.4 Sélection des classeurs fiables

De manière à établir un ensemble de classeurs qui anticipent avec précision, YACS dispose d'un mécanisme de suppression des classeurs qui anticipent mal. La trace T de taille θ_e des marqueurs S et E rend compte des capacités d'anticipation d'un classeur. À chaque pas de temps, YACS examine les classeurs dont la partie `Condition` est appariée avec s_{t-1} et la partie `Action` est égale à a_{t-1}. Ces classeurs sont les seuls susceptibles d'avoir vu leur trace T modifiée au cours de ce pas de temps.

- si la trace d'un de ces classeurs est pleine et si elle contient seulement des marqueurs E, alors YACS suppose que le classeur anticipe toujours mal, qu'il n'est pas fiable du tout, et le retire de la liste des classeurs ;
- si la trace est pleine et si elle contient à la fois des marqueurs de succès et d'échec, nous disons que le classeur oscille, c'est-à-dire qu'il anticipe parfois bien et parfois mal. Sa partie `Condition` est alors trop générale, elle doit être spécialisée davantage. Le mécanisme de spécialisation est décrit à la section suivante.

3.2.5 Spécialisation des conditions

À l'initialisation, YACS ne fait aucune distinction entre les situations (les parties `Condition` ne contiennent que des symboles `#`) et procède incrémentalement à des spécialisations des parties `Condition`. Ces spécialisations sont guidées par l'expérience, au lieu d'être décidées au hasard, puis évaluées *a posteriori* comme ce serait le cas avec des algorithmes génétiques.

Une partie `Condition` doit être aussi générale que possible et représenter des régularités dans les transitions entre les situations successives. Mais elle doit être suffisamment spécifique pour que le classeur anticipe toujours bien. Le mécanisme de spécialisation opère sur les parties `Condition` de manière incrémentale, jusqu'à atteindre le bon niveau de spécialisation.

L'opérateur de spécialisation

Le mécanisme de spécialisation de YACS utilise une variante de l'opérateur *mutspec* présenté par Dorigo (1994) et illustré par la figure 3.5. Cette variante choisit un attribut général de la partie `Condition` d'un classeur et produit une nouvelle partie `Condition` pour chaque valeur spécialisée possible de l'attribut choisi. Ces parties sont donc identiques à la partie `Condition` originale, à l'exception d'un attribut `#` qui prend une valeur spécifique.

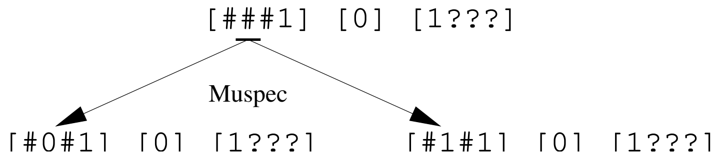

FIG. *3.5: L'opérateur de spécialisation.*

Un nouveau classeur est créé pour chacune de ces nouvelles parties `Condition` spécialisées. Les parties `Action` de ces nouveaux classeurs sont inchangées par rapport au classeur original. Des modifications peuvent intervenir dans les parties `Effet` dans le cas où la spécialisation de l'attribut général dans la partie `Condition` mène à une égalité entre le nouvel attribut de la partie `Condition` et l'attribut correspondant dans la partie `Effet`. Dans ce cas, le classeur spécifie une absence de changement concernant cet attribut, et l'attribut considéré devient `=` dans la partie `Effet` du nouveau classeur.

Par exemple, spécialiser le classeur `[0#0#] [0] [=01=]` selon le deuxième attribut devrait produire le classeur `[000#] [0] [=01=]`. Or, ce classeur spécifie que le second attribut ne change pas. Ceci doit donc être indiqué dans la partie `Effet`, pour que le mécanisme d'évaluation ne commette pas d'erreur. Le classeur effectivement créé est donc `[000#] [0] [==1=]`.

Une fois que ces nouveaux classeurs sont créés, le classeur original est écarté. Si la partie `Condition` du classeur original était appariée avec plusieurs situations, alors chaque nouvelle partie `Condition` définit un sous-ensemble de ces situations. Bien choisir l'attribut à spécialiser doit permettre de découper un ensemble de situations en sous-ensembles à l'intérieur desquels il devient possible d'anticiper correctement.

Les estimations pour la spécialisation

Choisir au hasard l'attribut à spécialiser, comme dans Alecsys (Dorigo, 1994), engendre des spécialisations non souhaitables. Par exemple, dans un monde de cases, le classeur `[1#######] [0] [========]` peut signifier que s'il y a un mur au nord, alors aller dans cette direction ne provoquera aucun changement. Dans ce cas, c'est le premier attribut qui est discriminant. Or ce classeur est obtenu par la spécialisation d'un classeur plus général : `[########] [0][========]`. Si le mécanisme de spécialisation spécialise d'abord selon un autre attribut que le premier, il faudra plusieurs classeurs pour modéliser un ensemble de transitions qui peuvent pourtant être représentées par un seul classeur. On est alors confronté à un problème de sur-spécialisation : certaines spécialisations auraient pu être évitées et le nombre de classeurs est sous-optimal. Il est donc souhaitable de se doter d'un mécanisme qui puisse non seulement choisir délibérément de spécialiser les seuls classeurs qui oscillent, mais qui puisse en outre les spécialiser selon l'attribut adéquat.

YACS améliore la sélection des attributs à spécialiser en utilisant des estimations e_s^i qui mesurent l'amélioration attendue par la spécialisation de chaque attribut général. Dans chaque classeur, une telle estimation est associée à chaque attribut dont la valeur est `#`. Cette estimation indique à quel point la spécialisation de l'attribut considéré permettrait de scinder l'ensemble des situations appariées avec la partie `Condition` en plusieurs sous-ensembles de cardinalité égale.

Considérons un classeur qui peut être sollicité dans l'anticipation des conséquences d'une action particulière dans différentes situations. Si la valeur de l'attribut de rang i des situations dans lesquelles le classeur anticipe bien est souvent différente de la valeur de l'attribut de même rang dans les situations à partir desquelles le classeur anticipe mal, alors cet attribut est vrai-

Transition			Effet réel
s_{t-1}	a_{t-1}	s_t	$diff(s_t, s_{t-1})$
[0110]	[0]	[1010]	[10==]

Classeurs			s_E	s_S	Mises à jour
[0#1#]	[0]	[10==]	[0010]	[0111]	*Succès de l'anticipation.* – La valeur du deuxième attribut est différente dans s_{t-1} et dans s_E : l'estimation associée est incrémentée. – La valeur du quatrième attribut est identique dans s_{t-1} et dans s_E : l'estimation associée est décrémentée. – La situation s_S associée au classeur devient s_{t-1}, c'est-à-dire [0110]
[0#1#]	[0]	[1==0]	[0111]	[0010]	*Échec de l'anticipation.* – La valeur du deuxième attribut est identique dans s_{t-1} et dans s_S : l'estimation associée est décrémentée. – La valeur du quatrième attribut est différente dans s_{t-1} et dans s_S : l'estimation associée est incrémentée. – La situation s_E associée au classeur devient s_{t-1}, c'est-à-dire [0110]

TAB. *3.5: Mise à jour des estimations pour la spécialisation*

semblablement indiqué pour distinguer les situations dans lesquelles le classeur anticipe bien de celles dans lesquelles il anticipe mal. Ainsi, la partie `Condition` du classeur doit être spécialisée selon cet attribut de rang i et l'estimation e_s^i qui lui est associée doit avoir une valeur élevée.

De manière à calculer ces estimations, comme indiqué dans la figure 3.1, à chaque classeur est associée la situation s_E précédant la dernière erreur d'anticipation, et la situation s_S précédant la dernière anticipation correcte. Pour chaque classeur tel que sa partie `Condition` est appariée avec s_{t-1} et sa partie `Action` avec a_{t-1} :

- si le classeur anticipe bien, alors pour chaque attribut :
  - si l'attribut de s_E est égal à l'attribut correspondant de s_{t-1}, alors l'estimation correspondante e_s^i est décrémentée ;
  - si l'attribut de s_E est différent de l'attribut correspondant de s_{t-1}, alors l'estimation correspondante e_s^i est incrémentée ;
- si le classeur anticipe mal, pour chaque attribut :
  - si l'attribut de s_S est égal à l'attribut correspondant de s_{t-1}, alors l'estimation corres-

pondante e_s^i est décrémentée ;
– si l'attribut de s_S est différent de l'attribut correspondant de s_{t-1}, alors l'estimation correspondante e_s^i est incrémentée.

Un exemple de mise à jour des estimations est donné par le tableau 3.5. Les estimations e_s^i sont incrémentées et décrémentées selon une règle de Widrow-Hoff paramétrée par un taux d'apprentissage $\beta_{spéc}$. Ainsi, les incrémentations s'effectuent selon la formule :

$$e_s^i \leftarrow (1 - \beta_{spéc})e_s^i + \beta_{spéc}$$

les décrémentations s'effectuent selon la formule :

$$e_s^i \leftarrow (1 - \beta_{spéc})e_s^i$$

Les valeurs initiales sont 0.5. Un attribut déjà spécialisé prend la valeur par défaut 0.5[22].

Le mécanisme de spécialisation de YACS

Un classeur oscille lorsque sa trace T contient à la fois des marqueurs S et E, c'est-à-dire lorsqu'il anticipe parfois bien et parfois mal. Si tel est le cas, la partie `Condition` de ce classeur doit être encore spécialisée, de manière à réduire encore le domaine de validité des conditions, jusqu'à ce qu'elle permettent de distinguer les situations bien anticipées des situations mal anticipées. À chaque pas de temps, compte tenu de s_{t-1} et de a_{t-1}, seuls les classeurs qui ont été évalués, et qui ont donc gagné un marqueur S ou E, sont susceptibles d'être oscillants alors qu'ils ne l'étaient pas auparavant. Ainsi, à chaque pas de temps, YACS cherche à spécialiser tous les classeurs tels que leur partie `Condition` est appariée avec s_{t-1}, et la partie `Action` avec a_{t-1}.

Le tableau 3.6 donne un exemple de spécialisation. Plutôt que de spécialiser chaque classeur indépendamment dès qu'il oscille, c'est-à-dire dès qu'il a au moins un marqueur S et au moins un marqueur E, YACS attend de collecter plus d'informations afin de rendre les estimations plus fiables. Pour ce faire, YACS attend que la trace T d'un classeur soit pleine pour le déclarer oscillant. Cette condition est vérifiée lorsque le classeur a été évalué au moins θ_e fois. La taille de la trace des classeurs θ_e est donc un seuil déterminant si un classeur a été suffisamment évalué pour être spécialisé.

De plus, YACS groupe les classeurs considérés par sous-ensembles avec les mêmes parties `Condition` et `Action`, et il attend que tous les membres d'un sous-ensemble oscillent pour les spécialiser. Cette prudence permet de collecter plus d'informations et d'affiner les estimations pour les rendre plus fiables.

Ainsi, à l'intérieur de chaque sous-ensemble de classeurs oscillants, l'attribut à spécialiser est choisi en fonction des estimations de chacun des classeurs correspondants. Les estimations relatives à chacun des attributs sont sommées et l'attribut doté de la plus grande somme est choisi. L'opérateur de spécialisation est alors appliqué à tous les classeurs du groupe, selon

[22] Ainsi, il n'est nécessaire de stocker ces estimations que pour les attributs généraux, les attributs spécialisés prenant tous la valeur 0.5.

Transition			Effet réel
s_{t-1}	a_{t-1}	s_t	$diff(s_t, s_{t-1})$
[0110]	[0]	[1010]	[10==]

Groupes de classeurs	Trace T	Estimations	Mises à jour
[##1#] [0] [10==]	$(SEES)$	$(0.1, 0.6, 0.5, 0.8)$	*Tous les classeurs du groupe*
[##1#] [0] [1==0]	(SE)	$(0.2, 0.6, 0.5, 0.7)$	*n'ont pas été suffisamment évalués.*
			– Aucune spécialisation.
[0###] [0] [10==]	$(SEES)$	$(0.2, 0.7, 0.5, 0.6)$	*Groupe de classeurs suffisamment*
[0###] [0] [1==0]	$(ESSE)$	$(0.1, 0.8, 0.5, 0.7)$	*évalués.*
			– Tous les classeurs du groupe oscillent.
			– Le deuxième attribut est crédité de
		Somme :	la plus grande somme d'estimations
		$(0.3, 1.5, 1.0, 1.3)$	– Les classeurs originaux sont supprimés.
			– Les classeurs suivant sont créés :
			[00##] [0] [10==]
			[01##] [0] [10==]
			[00##] [0] [1==0]
			[01##] [0] [1==0]

Tab. *3.6: Le mécanisme de spécialisation de YACS* le même attribut. Les classeurs originaux sont écartés. Certains des classeurs produits de la sorte anticiperont toujours mal mais seront éliminés par le mécanisme de sélection. Ceux qui oscilleront toujours seront spécialisés à nouveau, jusqu'à ce que suffisamment de spécialisations soient intervenues pour que les parties `Condition` permettent aux classeurs d'être fiables. Dans un environnement déterministe, si un classeur dont la partie `Condition` est complètement spécialisée oscille, c'est que cet environnement est non-markovien. Ce cas sort du champ de notre étude, mais sera toutefois discuté dans le chapitre 6.

De plus, l'opérateur de spécialisation peut produire des situations qui ne peuvent pas être rencontrées. Ainsi, les classeurs créés par l'opérateur de spécialisation, mais qui ne sont appariés avec aucune des situations déjà rencontrées, ne sont pas ajoutés à la liste. Cette propriété est vérifiée grâce à l'ensemble `P` des situations déjà perçues.

3.2.6 Généralisation des conditions

Même si le mécanisme de spécialisation est prudent, il peut produire des classeurs dont la partie `Condition` est à un niveau de généralisation sous-optimal. C'est surtout le cas au début de l'apprentissage, lorsque YACS spécialise alors qu'il n'a pas perçu beaucoup de situations parmi celles qui sont possibles. Ainsi, un mécanisme de généralisation est nécessaire pour revenir sur des spécialisations inadéquates.

Comme le mécanisme de spécialisation, le mécanisme de généralisation met en œuvre des heuristiques de manière à utiliser autant que possible l'information de la boucle sensori-motrice, contrairement à ACS qui emploie des algorithmes génétiques.

La généralisation repose, elle aussi, sur des estimations. Une estimation e_g^i mesure l'amélioration attendue par la généralisation dans la partie `Condition` de l'attribut de rang i considéré. Ainsi, une valeur e_g^i est associée à chaque attribut spécialisé de la partie `Condition`. Une telle estimation évalue dans quelle mesure la partie `Effet` du classeur demeurerait fiable si l'attribut était remplacé par un symbole `#`.

Ces estimations sont utilisées pour créer de nouveaux sous-ensembles de classeurs, à partir de sous-ensembles de classeurs initiaux dont les parties `Action` et `Effet` sont identiques. Les nouveaux classeurs sont plus généraux, leur nombre est plus petit et ils n'entrent pas en conflit avec d'autres classeurs du système.

Les estimations utilisées par YACS pour la généralisation

Afin de calculer les estimations e_g^i, YACS contrôle à chaque pas de temps les classeurs dont la partie `Action` est égale à a_{t-1}, et dont la partie `Condition` n'est pas appariée à s_{t-1}. Ces classeurs proposent la bonne action, mais leur condition est trop spécialisée pour qu'ils aient pu anticiper.

Étant donné de tels classeurs, pour chaque attribut spécialisé de la partie `Condition`, YACS vérifie si la partie `Condition` du classeur deviendrait compatible avec s_{t-1} si l'attribut considéré était un symbole `#`. Dans ce cas, l'estimation e_g^i considérée est mise à jour :

Transition			Effet réel
s_{t-1}	a_{t-1}	s_t	$diff(s_t, s_{t-1})$
[0110]	[0]	[1010]	[10==]

Classeurs			Mises à jour
[0#1#]	[0]	[10==]	*Condition appariée à* s_{t-1} . – Pas de mise à jour
[1#1#]	[0]	[10==]	*Condition non appariée à* s_{t-1} – La condition aurait été appariée si le premier attribut avait été général. – Avec une condition plus générale, le classeur aurait bien anticipé. – L'estimation du premier attribut est incrémentée
[1#1#]	[0]	[10==]	*Condition non appariée à* s_{t-1} – La condition aurait été appariée si le premier attribut avait été général. – Avec une condition plus générale, le classeur aurait mal anticipé. – L'estimation du premier attribut est décrémentée.

Tab. *3.7: Mise à jour des estimations pour la généralisation*

- si la partie `Effet` du classeur correspond à l'effet calculé en fonction de s_t et de s_{t-1}, alors un classeur avec une partie `Condition` plus générale aurait bien anticipé et l'estimation est incrémentée.
- si la partie `Effet` du classeur ne correspond pas à l'effet calculé en fonction de s_t et de s_{t-1}, alors un classeur avec une partie `Condition` plus générale aurait mal anticipé et l'estimation est décrémentée.

Les estimations sont incrémentées et décrémentées selon une règle de Widrow-Hoff de taux d'apprentissage $\beta_{gén}$. Les incrémentations s'effectuent selon la formule :

$$e_g^i \leftarrow (1 - \beta_{gén})e_g^i + \beta_{gén}$$

les décrémentations s'effectuent selon la formule :

$$e_g^i \leftarrow (1 - \beta_{gén})e_g^i$$

Les valeurs initiales sont 0.5. La valeur par défaut d'un attribut général est 0.5.

Le mécanisme de généralisation

Avec ce mécanisme et les estimations e_g^i, YACS est capable de déterminer si un attribut d'une partie `Condition` devrait être généralisé ou pas, en fonction des interactions du système avec l'environnement.

D'une situation s_{t-1} à une autre s_t, l'action sélectionnée a_{t-1} provoque un effet réel `ER` que l'on détermine grâce à la fonction *diff* définie à la section 3.2.2. YACS cherche à généraliser les classeurs avec cette partie `Effet` et une partie `Action` correspondant à l'action entreprise a_{t-1}. Ces classeurs sont les seuls susceptibles d'avoir bien anticipé le pas de temps précédent, et d'avoir ainsi vu augmenter leur nombre de marqueurs S[23], de manière à être considérés comme plus fiables par le système.

Le processus de généralisation opère donc sur des groupes de classeurs dotés des mêmes parties `Effet` et `Action`. Un tel groupe n'est examiné que lorsque chacun de ses classeurs a été suffisamment évalué pour être considéré comme fiable. Un classeur fiable doit n'avoir que des marqueurs S dans sa trace de marqueurs, et il doit avoir été évalué au moins θ_e fois.

La figure 3.6 illustre le mécanisme de généralisation. Dans cette figure, on ne considère qu'un seul de ces groupes, noté groupe A. À partir de chacun de ces groupes A, YACS forme un groupe B de classeurs plus généraux que ceux du groupe A. Le processus de généralisation d'un groupe A continue si chacun des classeurs du groupe A a été suffisamment évalué, et que sa trace T ne contient que des marqueurs attestant de succès dans l'anticipation. Les classeurs du groupe B sont construits de la manière suivante :

- si toutes les estimations e_g^i d'un des classeurs du groupe A sont inférieurs à 0.5, alors il n'est pas un bon candidat pour la généralisation et il est ajouté au groupe B sans modification ;

[23] Succès

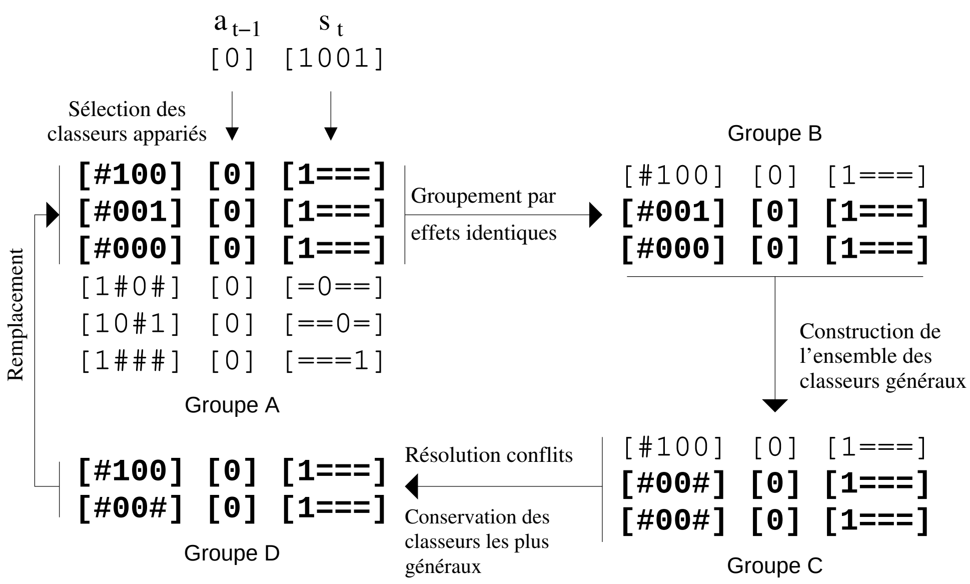

FIG. 3.6: *YACS : processus de généralisation.*

– dans le cas contraire, un nouveau classeur est créé et ajouté dans le groupe B. Ce nouveau classeur est égal à celui du groupe A, à ceci près que l'attribut de sa partie `Condition` doté de la plus grande estimation devient `#`.

Dans l'exemple de la figure 3.6, les estimations n'apparaissent pas, mais leurs valeurs ont conduit à généraliser le quatrième attribut des deuxième et troisième classeurs. En effet, cet attribut pourrait être général dans la mesure où les deux classeurs originaux sont fiables et prédisent les mêmes effets. Le premier classeur reste inchangé et les deux classeurs restants sont généralisés selon leur quatrième attribut.

À ce point, les classeurs du groupe B sont examinés pour détecter d'éventuels conflits avec d'autres classeurs de la liste de tous les classeurs. Deux classeurs sont en conflit si leur partie `Effet` est différente et que leur partie `Action` est la même, alors qu'il existe une situation prise en compte par les parties `Condition` des deux classeurs. Dans ce cas particulier, le système pourrait utiliser indifféremment un des deux classeurs pour anticiper des conséquences contradictoires. Il en résulterait une indécision et une perte de fiabilité du système d'anticipation. Parce que nous voulons éviter de perdre de la connaissance au cours de l'apprentissage, nous voulons éviter ce type de conflits. En conséquence, si un classeur du groupe B est en conflit avec un autre, alors il est remplacé par le classeur correspondant du groupe A. Le groupe résultant est noté C.

De manière à éliminer les doublons et à ne conserver que les classeurs les plus généraux du groupe C, YACS examine itérativement toutes les paires possibles de classeurs du groupe C. Lorsque la partie `Condition` d'un classeur est identique ou plus générale que celle d'un autre classeur, alors le premier classeur est conservé, et le deuxième est écarté du groupe C.

De cette manière, YACS a créé un nouveau groupe C dont chaque classeur est plus général ou égal à un classeur du groupe original, et dont aucun n'est en conflit avec d'autres classeurs. Dès lors, les classeurs du groupe A sont remplacés dans la liste générale par les classeurs du groupe C.

Ce mécanisme permet de remplacer des groupes de classeurs par d'autres classeurs plus généraux et moins nombreux. Il est le pendant du mécanisme de spécialisation. L'association de ces deux mécanismes permet de régler le niveau de généralité de classeurs de façon incrémentale.

3.3 Étude expérimentale

3.3.1 Les automates à état finis de type Wilson Woods

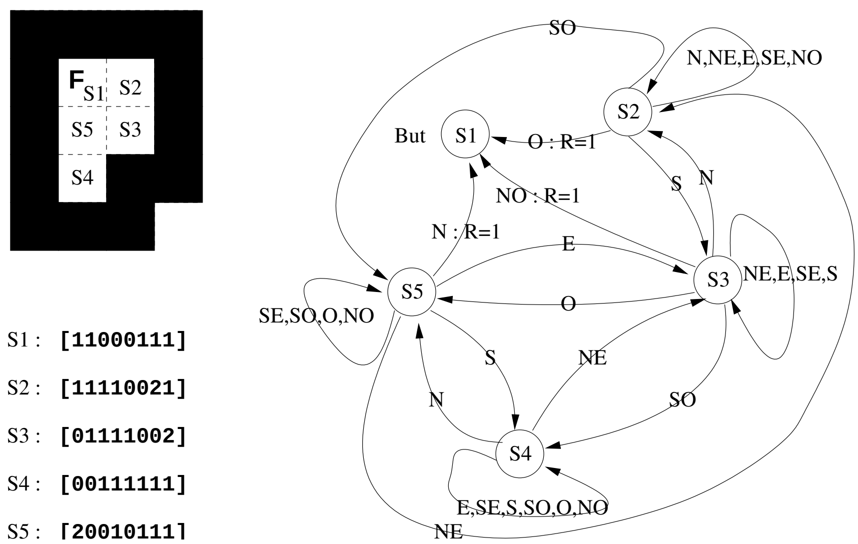

FIG. *3.7: Environnement de type « Wilson Woods » et automate à états finis correspondant.* Les cases du labyrinthe correspondent aux noeuds de l'automate. Les transitions dans l'automate correspondent, dans le labyrinthe, à des actions permettant de passer de case en case, dans chacune des directions cardinales.

Plusieurs types d'environnements sont utilisées pour étudier les systèmes de classeurs. Les principaux sont les *Wilson Woods* (Wilson, 1985; Wilson, 1994) et les autres sont les automates à états finis (Riolo, 1988; Smith, 1994). Comme nous allons le montrer, les *Wilson Woods* sont des cas particuliers d'automates à états finis, dont la représentation est plus intelligible. La figure 3.7 montre un *Wilson Wood* et l'automate à états finis qu'il représente.

Les automates à états finis sont composés de nœuds et d'arcs orientés reliant ces nœuds. Les labels associés aux arcs représentent l'action à entreprendre pour que la transition ait lieu. Une récompense peut être associée à chaque transition. Dans le cas d'environnements stochastiques, on peut aussi leur associer des probabilités. Ce type d'automates permet de représenter un problème de décision markovien quelconque. L'agent doit apprendre à maximiser la récompense reçue. Si l'agent reçoit une récompense dès qu'il atteint un nœud but, sa tâche est d'apprendre à atteindre le but dans les meilleurs délais.

Les *Wilson Woods* sont des mondes de cases simulés. Ils se présentent sous la forme d'un tableau de cases, chacune pouvant être vide, obstruée par un obstacle ■, ou contenir de la nourriture F. L'agent est situé dans une case et perçoit les huit cases adjacentes. Chaque attribut spécifie la présence ou l'absence d'un obstacle dans chacune de ces cases. La valeur d'un attribut est `0` lorsque la case correspondante est vide, `1` lorsqu'elle est obstruée par un obstacle et `2` lorsqu'elle contient de la nourriture[24]. Les attributs sont agrégés pour former une situation en commençant par celui correspondant à la case au nord, puis en les prenant dans le sens des aiguilles d'une montre. A chaque pas de temps, l'agent perçoit sa situation et doit décider dans quelle case adjacente se déplacer. Il a donc huit actions à sa disposition, une pour chaque direction cardinale. Lorsqu'il atteint la case but marquée F, il reçoit une récompense. Sa tâche est d'apprendre à rejoindre le but le plus rapidement possible.

Pour étudier YACS, nous utiliserons des *Wilson Woods* parce que nous pensons que la définition d'un problème de Markov est plus simple et intelligible (Lanzi, 2000). Même la représentation par automates de *Wilson Woods* très simples peut être complexe et difficile à appréhender. En outre, dans ce type d'environnement, il est plus facile d'identifier les attributs associés à chaque état, et de comprendre les régularités exploitées par le système.

L'utilisation de *Wilson Woods* est un choix qui concerne seulement la représentation d'un problème. Les algorithmes que nous développons ne sont pas dévolus à la seule navigation dans un monde de cases. Toutefois, si n'importe quel *Wilson Wood* peut être représenté par un automate à états finis, l'inverse n'est pas vrai. Pour que les résultats obtenus dans des *Wilson Woods* soient généralisables à d'autres problèmes de décision markoviens[25], il convient de ne faire, dans les algorithmes conçus, aucune hypothèse concernant la particularité des *Wilson Woods*. Par exemple, dans de tels environnements, les transitions sont symétriques[26], ce qui n'est généralement pas le cas. Dans YACS, nous n'avons jamais fait d'hypothèses de ce type.

3.3.2 Les environnements Maze4 et Maze6

Dans cette étude expérimentale, nous présentons des résultats de YACS interagissant avec des *Wilson Woods* utilisés pour tester ACS (Butz *et al.*, 2000a). La topologie de ces environnements – Maze4 et Maze6 – est donnée par les figures 3.8 et 3.9.

Les expériences sont divisées en essais. Au début de chaque essai, l'agent est placé aléatoirement dans une case vide. À chaque pas de temps, l'agent perçoit sa situation puis choisit une action qui le fait progresser dans le labyrinthe. L'essai se termine lorsque l'agent atteint la case contenant de la nourriture. À cet instant, il reçoit une récompense scalaire de 1 et l'agent est

---

[24] Dans les *Wilson Woods* originaux, la valeur `0` est aussi utilisée dans le cas d'un attribut qui correspond à une case contenant de la nourriture.

[25] Initialement conçu et validé avec des *Wilson Woods*, ACS a été ensuite utilisé tel quel pour d'autres problèmes, comme le contrôle d'un robot Khepera (Stolzmann, 1999) ou la simulation de transport de cubes (Butz *et al.*, 2000b).

[26] À chaque action permettant de passer d'un premier état à un second, en correspond une autre qui permet de passer du second au premier.

`[1#######] [0] [========]`	`[#1######] [1] [========]`
`[##1#####] [2] [========]`	`[###1####] [3] [========]`
`[####1###] [4] [========]`	`[#####1##] [5] [========]`
`[######1#] [6] [========]`	`[#######1] [7] [========]`

TAB. *3.8: Les huit classeurs permettant de représenter toutes les transitions n'occasionnant aucun changement dans l'environnement.* Il y a un classeur pour chaque action possible. Les parties `Condition` de ces classeurs ne contiennent que des attributs généraux, à l'exception des attributs correspondant à la direction de l'action, qui spécifient qu'un obstacle est présent dans cette direction. Les parties `Effet` de ces classeurs sont uniquement composées de symboles =, pour indiquer que rien ne change dans la situation perçue

replacé aléatoirement dans le labyrinthe pour commencer un nouvel essai.

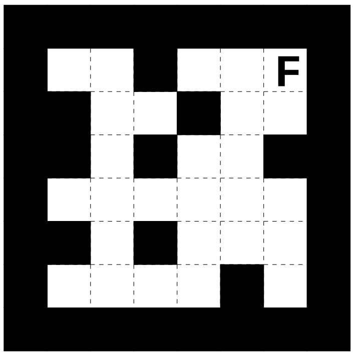

FIG. *3.8: L'environnement Maze4.* L'agent perçoit les huit cases autour de lui et peut entreprendre un mouvement vers n'importe laquelle de ces huit cases. Il y a 208 transitions dans Maze4, dont 93 concernent des mouvements vers un mur et peuvent donner lieu à une généralisation. Des attributs supplémentaires sont ajoutés pour donner d'autres opportunités de généralisation.

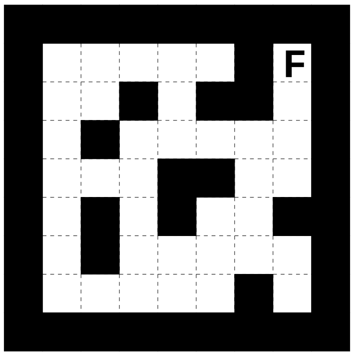

FIG. *3.9: L'environnement Maze6.* L'agent perçoit les huit cases autour de lui et peut entreprendre un mouvement vers n'importe laquelle de ces huit cases. Il y a 288 transitions dans Maze4, dont 135 concernent des mouvements vers un mur et peuvent donner lieu à une généralisation. Des attributs supplémentaires sont ajoutés pour donner d'autres opportunités de généralisation.

Dans ces environnements, il est possible à YACS d'utiliser la généralisation dès qu'une action ne mène à aucun changement. C'est notamment le cas lorsqu'une action mène l'agent vers un mur. Dans ce cas, l'agent reste sur la même case et sa situation ne change pas. Il y a respectivement 93 et 135 transitions de ce type dans Maze4 et Maze6. En utilisant ses capacités de généralisation, toutes les transitions résultant de telles actions peuvent être modélisées par les seuls 8 classeurs du tableau 3.8. Il n'y a pas d'autres régularités exploitables par YACS dans Maze4 et Maze6.

Puisque le nombre total de transitions dans Maze4 et Maze6 est respectivement 208 et 288, les nombres optimaux de classeurs que YACS devrait atteindre compte tenu de son formalisme sont respectivement 123 $(208 - 93 + 8)$ et 161 $(288 - 135 + 8)$.

De plus, comme cela a été fait pour ACS (Butz *et al.*, 2000b), de manière à donner à YACS plus d'occasions d'utiliser la généralisation, nous avons ajouté des attributs supplémentaires et sans signification aux situations perçues. Une valeur aléatoire `0` ou `1` est attribuée à chacun d'eux au début de chaque essai. Leur valeur ne change pas tout au long de l'essai. Ces attributs n'apportent aucune information utile à la discrimination des situations et peuvent être ignorés. Pour un système dépourvu de capacités de généralisation, comme *DynaQ+*, ces attributs mèneraient à considérer de nombreuses situations nouvelles. Avec deux attributs supplémentaires, le nombre de situations est multiplié par quatre. Un système incapable de généraliser devrait avoir une liste de 832 (208×4) transitions pour Maze4 et 1152 (288×4) transitions pour Maze6. Par contre, dans la mesure où ces nouveaux attributs ne sont d'aucune utilité pour distinguer les différentes situations, le nombre de classeurs pour YACS devrait rester le même, quel que soit le nombre d'attributs supplémentaires ajoutés.

3.3.3 Résultats expérimentaux

Comme pour ACS (Butz *et al.*, 2000b), pour rendre compte de l'évolution de la fiabilité et de la complétude du modèle au cours du temps, nous utilisons une mesure du pourcentage de connaissance fourni par le modèle. Pour chaque transition possible dans l'environnement, nous vérifions si le système de classeurs la modélise correctement, c'est-à-dire s'il anticipe une seule situation, celle qui est effectivement observée. Le pourcentage de connaissance est le taux de transitions bien modélisées par le système. Cette mesure nécessite de confronter le modèle appris par l'agent à un modèle parfait connu par ailleurs. Elle ne peut donc pas être effectuée par le système lui-même puisqu'elle suppose une connaissance parfaite de l'environnement dès le début de l'apprentissage. Puisque Maze4 et Maze6 sont des environnements simulés, on en a une connaissance parfaite et il devient possible de mesurer l'évolution de ce pourcentage de connaissance.

Comme pour ACS, l'agent choisit ses actions aléatoirement. Tous les résultats présentés sont des moyennes sur 100 expériences. Pour YACS, la taille θ_e de la trace de marqueurs est égale à 5 et les taux d'apprentissage $\beta_{spéc}$ et $\beta_{gén}$ sont égaux à 0.1. Ces taux d'apprentissage sont utilisés dans les règles de Widrow-Hoff pour permettre la mise à jour des différentes estimations utilisées pour la spécialisation et la généralisation.

Les figures 3.10, 3.12 et 3.14 montrent l'évolution du pourcentage de connaissance au cours des pas de temps successifs, lorsque YACS évolue dans Maze4 avec respectivement 0, 1 et 2 attributs supplémentaires. Les figures 3.11, 3.13 et 3.15 montrent l'évolution du nombre de classeurs dans les mêmes conditions. Les figures 3.16, 3.18, 3.20, 3.17, 3.19 et 3.21 montrent les résultats correspondants pour Maze6.

Le tableau 3.9 résume, pour chacune des expériences, la moyenne et l'écart type du nombre de classeurs obtenus, ainsi que les statistiques attachées au nombre de pas de temps nécessaires

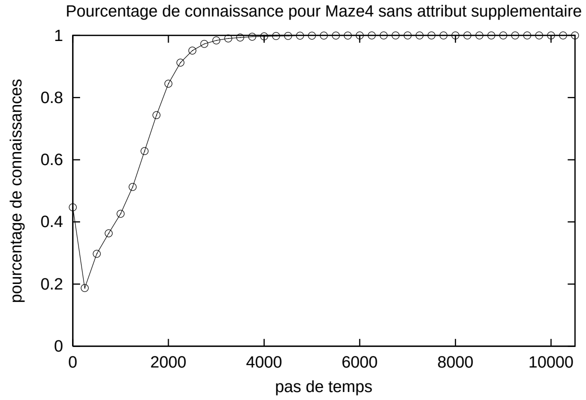

FIG. *3.10: YACS : évolution du pourcentage de connaissance pour Maze4 sans attribut supplémentaire.*

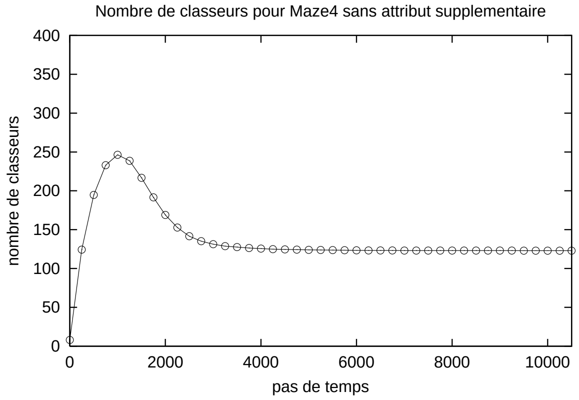

FIG. *3.11: YACS : évolution du nombre de classeurs pour Maze4 sans attribut supplémentaire.*

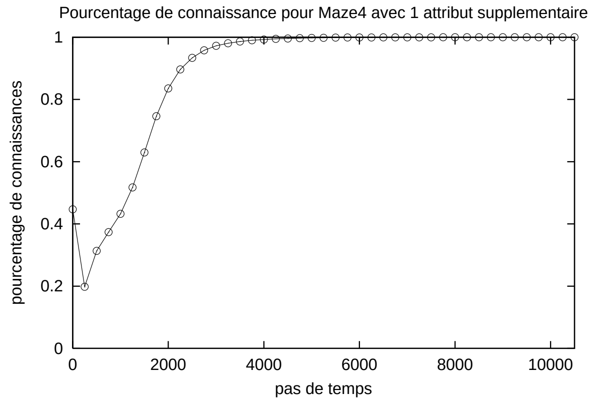

FIG. *3.12: YACS : évolution du pourcentage de connaissance pour Maze4 avec 1 attribut supplémentaire.*

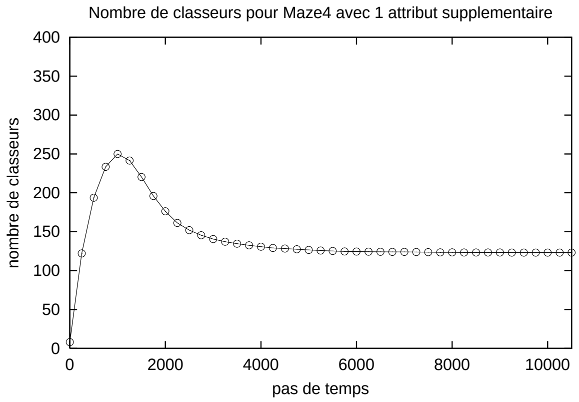

FIG. *3.13: YACS : évolution du nombre de classeurs pour Maze4 avec 1 attribut supplémentaire.*

à YACS pour atteindre un pourcentage de connaissance de 0.99. Les expériences ont toutes été menées sur 20 000 pas de temps. Le nombre de classeurs obtenus est donc celui qui a été atteint après ce délai.

Durant les premiers pas de temps, le pourcentage de connaissance croît très vite. En effet, YACS modélise rapidement les transitions qui ne provoquent aucun changement dans les situations perçues, puisque trouver les parties `Condition` appropriées ne nécessite qu'une seule spécialisation, pour peu que cette spécialisation soit guidée par des estimations fiables. Les autres transitions sont plus complexes à modéliser et, à mesure que le modèle devient plus complet, l'exploration aléatoire fournit de moins en moins fréquemment d'informations utiles à l'amélioration du modèle. Les courbes d'évolution du pourcentage de connaissance sont donc asymptotiques.

Les dégradations de performances lorsque l'on ajoute des attributs supplémentaires semblent plus importantes pour Maze4 que pour Maze6. Cela s'explique par le fait que, comme Maze6 est

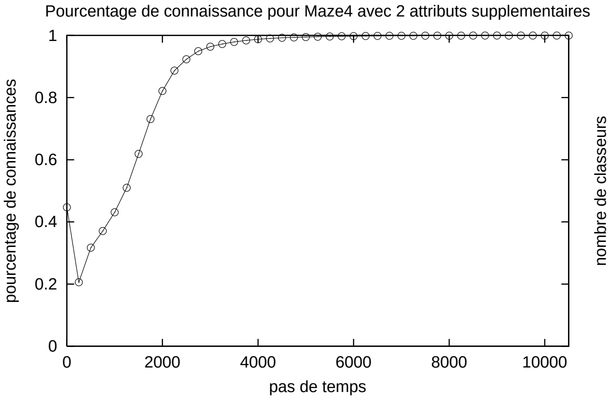

FIG. 3.14: *YACS : évolution du pourcentage de connaissance pour Maze4 avec 2 attributs supplémentaires.*

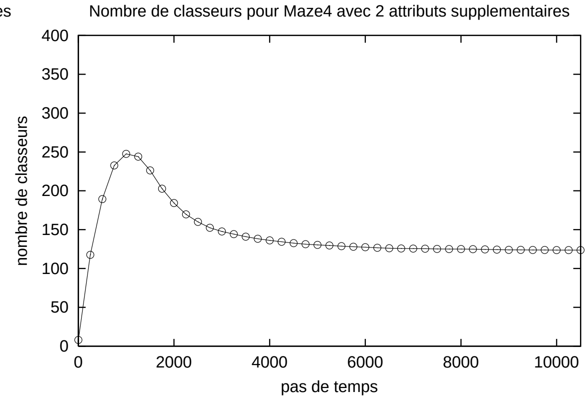

FIG. 3.15: *YACS : évolution du nombre de classeurs pour Maze4 avec 2 attributs supplémentaires.*

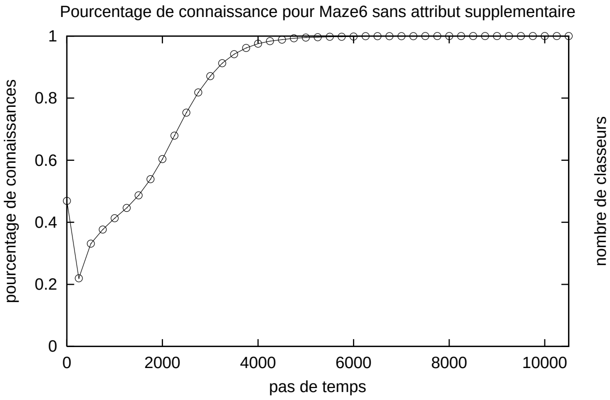

FIG. 3.16: *YACS : évolution du pourcentage de connaissance pour Maze6 sans attribut supplémentaire.*

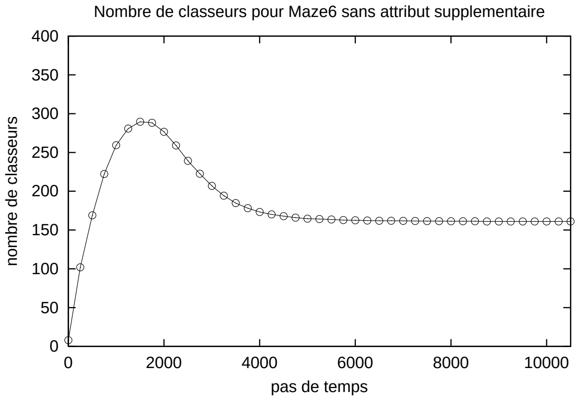

FIG. 3.17: *YACS : évolution du nombre de classeurs pour Maze6 sans attribut supplémentaire.*

plus grand que Maze4, l'exploration aléatoire conduit YACS à y achever un essai moins rapidement. Dès lors, l'agent reste plus longtemps dans des situations où les attributs supplémentaires ne changent pas et l'estimation attachée à la spécialisation est plus fiable. En conséquence, en présence d'attributs supplémentaires, YACS montre une moindre tendance à la sur-spécialisation dans des environnements plus grands comme Maze6.

Butz et al. (2000b) présentent les résultats d'ACS concernant l'expérience de Maze4 avec deux attributs supplémentaires. Ces résultats indiquent qu'ACS atteint un niveau de connaissances de 0.99 après approximativement 12 500 pas de temps (contre 4 100 pour YACS). ACS semble converger vers un nombre de classeurs proche de 400 (contre 123 pour YACS), après que le nombre de classeurs ait culminé à 1 400 (contre 250 pour YACS). Ainsi, YACS semble converger beaucoup plus vite vers des solutions beaucoup plus compactes qu'ACS. Dans la section suivante, nous comparons ACS et YACS et discutons les différences entre leurs mécanismes respectifs, de

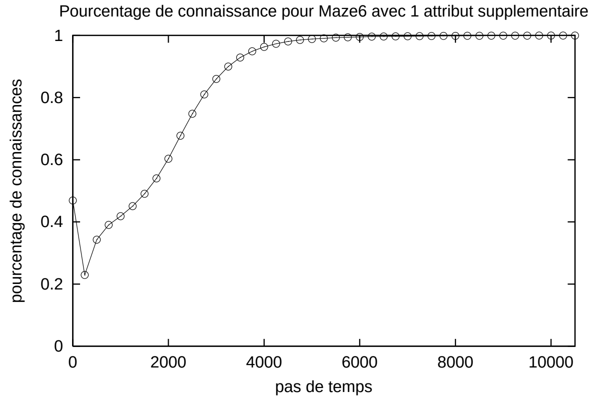

FIG. *3.18: YACS : évolution du pourcentage de connaissance pour Maze6 avec 1 attribut supplémentaire.*

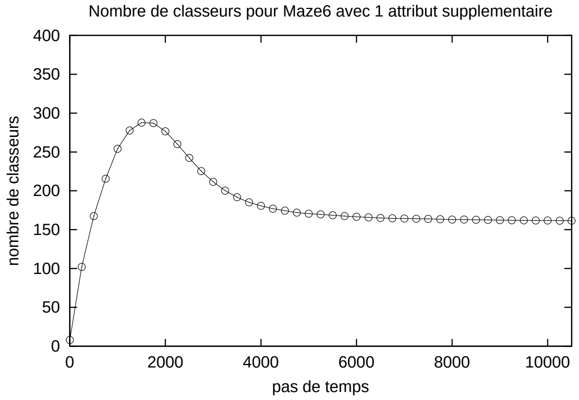

FIG. *3.19: YACS : évolution du nombre de classeurs pour Maze6 avec 1 attribut supplémentaire.*

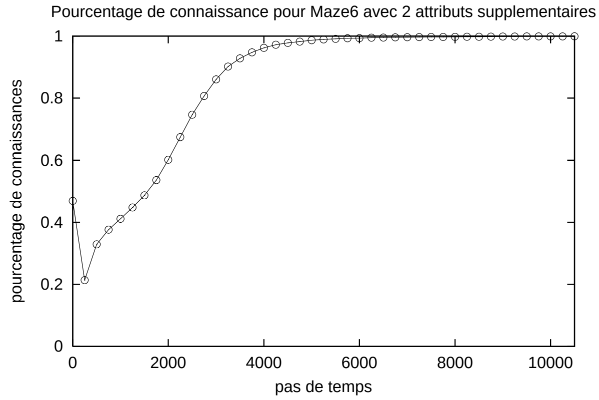

FIG. *3.20: YACS : évolution du pourcentage de connaissance pour Maze6 avec 2 attributs supplémentaires.*

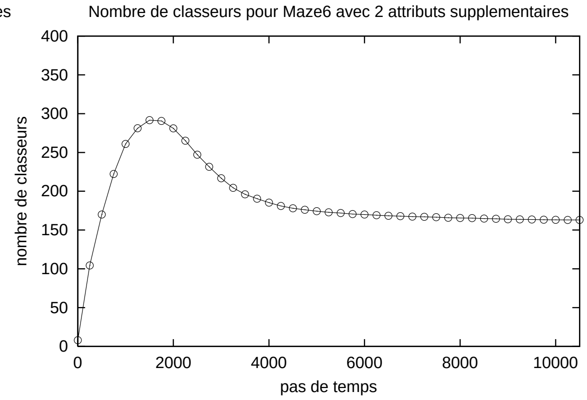

FIG. *3.21: YACS : évolution du nombre de classeurs pour Maze6 avec 2 attributs supplémentaires.*

manière à expliquer cette différence de performance.

3.4 Comparaison avec ACS

3.4.1 Problèmes abordés

La première différence entre YACS et ACS est l'étendue des problèmes abordés par l'un et l'autre système.

Problèmes non-markoviens

Contrairement à YACS, ACS (Stolzmann, 1998; Stolzmann, 1999) est doté d'extensions capables de traiter les problèmes non-markoviens (voir section 1.2.2). Dans de tels problèmes, les

	Nombre de classeurs		Temps de convergence	
Expérience	Optimal	Moyenne	Moyenne	Écart type
Maze4, aucun attribut supplémentaire	123	123	3116	417
Maze4, 1 attribut supplémentaire	123	123	3622	619
Maze4, 2 attributs supplémentaires	123	123	4091	913
Maze6, aucun attribut supplémentaire	161	161	4592	527
Maze6, 1 attribut supplémentaire	161	161	5155	1102
Maze6, 2 attributs supplémentaires	161	161.2	5509	1602

TAB. *3.9: Synthèse des résultats expérimentaux concernant YACS*

seules situations perçues par l'agent ne lui fournissent pas suffisamment d'informations pour pouvoir décider de l'action optimale. Dans ce cas, les informations sensorielles ne permettent pas de discriminer tous les états de l'environnement, et l'agent perçoit la même situation dans des états différents. Ces situations sont dites « ambiguës ». Dans une situation ambiguë, une action peut mener à des conséquences différentes.

Ainsi, si la partie `Condition` d'un classeur est appariée avec une situation ambiguë, même si elle est complètement spécialisée[27], le classeur anticipe parfois bien et parfois mal. En conséquence, il oscille mais il ne peut être spécialisé plus avant. Il n'est pas non plus supprimé puisqu'il anticipe parfois correctement. En outre, à cause du mécanisme de couverture des effets, au moins un autre classeur a des parties `Condition` et `Action` identiques, mais une partie `Effet` différente. Chaque classeur modélise alors une des conséquences possibles. Pour une situation et une action donnée, sans information concernant le passé, YACS serait donc incapable de proposer une anticipation fiable. De plus, puisqu'aucun classeur dont la partie `Condition` est appariée avec la situation ambiguë ne peut être fiable, YACS les spécialise tous, provoquant un problème de sur-spécialisation.

Pour résoudre le problème lié aux ambiguïtés, ACS apprend des séquences d'actions (voir section 6.2.5).

Problèmes stochastiques

De plus, alors que YACS est dévolu à des environnements déterministes, Butz et al. (2001) proposent une extension d'ACS qui est capable d'aborder des problèmes stochastiques. En enregistrant pour chaque classeur plusieurs effets possibles, chacun valué par une probabilité, ACS est en mesure d'interagir avec des environnements stochastiques. Dans ce type de problèmes, le résultat des actions entreprises n'est pas déterministe mais obéit à une distribution de probabilités. Cette distribution de probabilités peut intégrer deux sortes de bruit :

– du bruit sur les actions implique que l'action choisie n'est pas forcément exactement celle qui est effectivement exécutée ;

[27]Elle ne contient aucun symbole `#`.

s^i_t	e^i	$diff(s^i_t, s^i_{t-1})$	$passthrough(s^i_{t-1}, e^i)$	YACS	ACS
$= s^i_{t-1}$	=#	#	s^i_{t-1}	succès	succès
$\neq s^i_{t-1}$	=#	s^i_t	s^i_{t-1}	échec	échec
$= s^i_{t-1}$	$= s^i_t$	#	s^i_t	*échec*	*succès*
$\neq s^i_{t-1}$	$= s^i_t$	s^i_t	s^i_t	succès	succès
$= s^i_{t-1}$	$\star$	#	$\star$	échec	échec
$\neq s^i_{t-1}$	$\star$	s^i_t	$\star$	échec	échec

TAB. *3.10: Table de vérité*

– du bruit sur les perceptions simule des capteurs défectueux ou imprécis, et amène certains des attributs de la situation perçue à être différents de ce qu'ils devraient être.

Dans de telles conditions, une action entreprise dans une situation particulière pourrait avoir des effets divers. De même que dans le cas des ambiguïtés, les classeurs construits par YACS ne pourraient jamais être fiables. Même un classeur dont la partie `Condition` serait complètement spécialisée anticiperait parfois bien et parfois mal : tous les classeurs oscilleraient. En outre, le mécanisme de couverture des effets conduirait à produire de plus en plus de classeurs. Il y aurait donc toujours plusieurs classeurs dont les parties `Condition` et `Action` seraient respectivement appariées avec la situation et l'action courante.

3.4.2 Évaluation des classeurs

Toutefois, les extensions à ACS évoquées ci-dessus ne sont pas utilisées Butz et al. (2000b) et ne permettent pas d'expliquer la différence constatée lors de l'étude expérimentale.

Si les formalismes de YACS et d'ACS sont semblables et même si l'interprétation d'un classeur est identique, le nombre de classeurs produits par YACS est inférieur au nombre de classeurs produits par ACS dans les mêmes conditions. Cela pourrait résulter des critères utilisés par les deux systèmes pour décider si un classeur est adéquat ou non. Dans ce cas, il serait normal que les deux systèmes produisent des classeurs en nombres différents.

Stolzmann (1998) donne une description détaillée de l'évaluation telle qu'elle est effectuée dans ACS. Alors que, dans YACS, un effet réel `ER` est calculé en fonction de s_{t-1} et de s_t grâce à la fonction *diff* pour ensuite être comparé avec les parties `Effet` des classeurs concernés, le mécanisme est inverse dans ACS. En effet, dans ACS, pour chaque classeur concerné par s_{t-1} et a_{t-1}, une situation anticipée est produite grâce à la fonction *passthrough* en fonction de sa partie `Effet` et de s_{t-1}, pour être ensuite comparée à s_t (voir section 2.4.4).

Ces deux mécanismes produisent effectivement des différences, mais elles sont trop marginales pour expliquer la différence dans ce nombre de classeurs produits. Les opérateurs *diff* et *passthrough* sont tous deux définis sur chacun des attributs des situations. Considérons s^i_{t-1} le $i^{ème}$ attribut de s_{t-1} et s^i_t l'attribut correspondant de s_t. Le tableau 3.10 compare, dans tous les cas de figure possibles, le critère $e^i = diff(s^i_t, s^i_{t-1})$ utilisé par YACS, et le critère

Transition			Effet réel
s_{t-1}	a_{t-1}	s_t	$diff(s_t, s_{t-1})$
[0110]	[0]	[1010]	[10==]

Système	Classeur			Évaluation
ACS	[0###]	[0]	[10=0]	Anticipation correcte : $passthrough(s_{t-1}i, \texttt{Effet}) = s_t$
YACS	[0###]	[0]	[10=0]	Anticipation erronée : $\texttt{Effet} \neq diff(s_t, s_{t-1})$

TAB. *3.11: Différence dans l'évaluation d'un classeur par ACS et par YACS.*

$passthrough(s^i_{t-1}, e^i) = s^i_t$ utilisé par ACS. Ici, e^i désigne le $i^{ème}$ attribut de la partie `Effet` du classeur considéré. Ce tableau montre que les deux critères sont équivalents sauf dans un cas, lorsque $s^i_{t-1} = s^i_t = e^i$. L'exemple du tableau 3.11 donne un exemple de classeur qui sera mal évalué par YACS mais bien évalué par ACS.

Dans ACS, la spécialisation d'un attribut de la partie `Effet` entraîne systématiquement la spécialisation de l'attribut correspondant dans la partie `Condition`. En conséquence, sans le mécanisme de généralisation d'ACS, ce cas ne se produit jamais. Pourtant, ACS produit encore plus de classeurs sans son mécanisme de généralisation. La différence d'efficacité en termes de nombre de classeurs produits vient donc des heuristiques utilisées.

3.4.3 Les mécanismes d'ACS et de YACS

YACS est défini de manière à décorréler la découverte de parties `Condition` discriminantes et le positionnement des parties `Effet`. Dans YACS, la partie `Effet` spécifie seule les changements perçus dans l'environnement. Une valeur spécifique indique toujours un changement, quelle que soit la partie `Condition`. De même, un attribut `=` indique toujours que la valeur de l'attribut ne changera pas. YACS garantit qu'une valeur spécifique dans une partie `Condition` n'est jamais égale à celle de l'attribut correspondant dans la partie `Effet`. Dans ce cas, la valeur adéquate de ce dernier serait `=` pour indiquer une absence de changement (voir section 3.2.2).

Cette propriété affecte le processus d'apprentissage latent. Une première partie de l'apprentissage latent consiste à introduire des parties `Effet` correspondant aux changements effectivement observés dans l'environnement. YACS ne cherche pas seulement des classeurs permettant d'anticiper correctement la situation suivante, mais cherche aussi à ce que les parties `Effet` décrivent seules les changements observés alors que, pour spécifier un changement, ACS utilise à la fois la partie `Effet` et la partie `Condition`. Les mécanismes de spécialisation et de généralisation de YACS permettent de trouver des parties `Condition` en adéquation avec les parties `Effet`, sans modifier ces dernières. Une partie `Condition` doit permettre de discriminer les situations dans lesquelles la partie `Effet` est toujours correcte des situations dans lesquelles elle est toujours fausse.

Dans ACS, les parties `Effet` sont découvertes grâce au mécanisme de spécialisation des attributs changeants (l'ALP décrit à la section 2.4.4). Dès qu'un classeur anticipe mal, cette partie de l'apprentissage latent cherche à créer un nouveau classeur en spécialisant à la fois la partie `Condition` et la partie `Effet`. Or, comment parler de spécialisation dans une partie `Effet` ? Ces symboles `=` ont, en effet, une sémantique différente des symboles `#`, qui représentent des jokers. Les symboles `=` ne font que représenter des absences de changements. C'est pourquoi, dans YACS, le processus incrémental de spécialisation/généralisation n'est appliqué qu'aux parties `Condition`. Les parties `Effet` sont positionnées d'un obstacle, quel que soit le nombre de symboles `=`.

Pour ACS, en revanche, on considère qu'une spécialisation intervient aussi sur la partie `Effet`. Ne prenant pas en compte le fait que le rôle de la spécialisation est simplement de réduire le champ d'application du classeur, considérant donc qu'elle ne doit concerner que la partie `Condition` dont c'est le rôle strict, Stolzmann (1998) a défini la spécialisation d'ACS de manière à ce qu'elle affecte également les parties `Effet`. Cela engendre essentiellement deux problèmes.

En premier lieu, spécialiser uniquement les attributs qui changent d'un pas de temps à l'autre peut empêcher d'identifier les attributs vraiment pertinents pour la spécialisation, même s'ils ne changent pas une fois que l'action a été exécutée. Pour résoudre ce problème, ACS inclut un nouveau mécanisme dédié qui permet de spécialiser les attributs invariants, quand l'ALP ne permet pas une discrimination adéquate. Dans YACS, l'attention portée à la séparation des parties `Condition` et `Effet` a permis de proposer un mécanisme de spécialisation unique et plus simple.

En second lieu, la spécialisation telle qu'elle est faite dans ACS conduit à spécialiser selon des attributs qui changent, certes, mais qui ne sont d'aucune utilité pour distinguer les situations dans lesquelles le classeur anticipe bien des situations dans lesquelles il anticipe mal. Cette façon de faire introduit donc une forte tendance à la sur-spécialisation des parties `Condition`. En conséquence, puisque la méthode d'évaluation des classeurs dans ACS et YACS, bien que différente, ne permet pas d'expliquer pourquoi YACS produit moins de classeurs, ce sont les mécanismes de spécialisation qui sont en cause. En effet, la méthode utilisée par YACS pour spécialiser, en décorrélant les parties `Effet` et `Condition`, permet de ne spécialiser que les attributs dont on a de bonnes raisons de penser qu'ils sont discriminants, au lieu de spécialiser systématiquement tous ceux qui changent.

Nous montrons ailleurs (Gérard *et al.*, 2002) que YACS, même privé de son mécanisme de généralisation, produit moins de classeurs qu'ACS avec le sien. C'est le cas parce que YACS fait assez peu d'erreurs quant au choix des attributs à spécialiser, principalement parce que YACS son mécanisme de spécialisation est prudent et attend de collecter suffisamment d'information avant de décider une spécialisation.

De plus, le mécanisme de généralisation d'ACS repose sur des algorithmes génétiques (Butz *et al.*, 2000a) qui ne sont pas explicitement dirigés par l'expérience. Ils créent donc beaucoup de classeurs qui anticipent mal et introduisent une grande redondance dans l'ensemble des classeurs. En effet, s'il y a plusieurs façons d'identifier un groupe de transitions à l'exclusion d'un autre groupe, tant que les classeurs restent fiables, ACS conserve cette redondance. Chacun des classeurs peut être maximalement général[28], sans que la taille du système de classeurs soit optimale. Cette redondance peut être souhaitable, mais le choix de l'éliminer, qui a été fait dans le cadre de YACS, permet de produire un plus petit nombre de classeurs tout en étant sensiblement plus efficace.

3.5 Discussion

3.5.1 Décorrélation des conditions et des effets

Dans ce chapitre, nous avons proposé YACS, un nouveau système de classeurs à anticipation. Ce système utilise un formalisme semblable à celui d'ACS, et il utilise des classeurs `Condition Action Effet`. L'explicitation des rôles tout à fait différents des parties `Condition` et `Effet` conduit à repenser les mécanismes des spécialisations et généralisations qu'ACS met en œuvre simultanément sur les parties `Effet` et `Condition`.

Nous proposons avec YACS de nouveaux mécanismes de généralisation et de spécialisation explicites n'intervenant que sur la partie `Condition`. Les parties `Effet` sont modifiées par ailleurs et, contrairement à ce qui se passe dans ACS, aucun mécanisme n'intervient sur les classeurs entiers. C'est la synergie entre ces mécanismes qui permet au système d'apprendre un modèle de l'environnement. Sans modifier le formalisme d'ACS, mais par le seul biais des mécanismes utilisés, YACS décorrèle les parties `Condition` et `Effet` des classeurs.

3.5.2 Utilisation d'estimations

Avec YACS, nous avons donc proposé de nouvelles heuristiques pour les systèmes de classeurs à anticipation. Ces heuristiques sont fondées sur des estimations mises à jour incrémentalement et permettant de guider la création de nouveaux classeurs.

Sans utiliser les mêmes heuristiques que YACS, il reste néanmoins possible d'utiliser les mêmes estimations de manière indépendante, pour introduire des biais dans des algorithmes génétiques, par exemple. En effet, la plupart des systèmes de classeurs utilisent des algorithmes génétiques et ont recours à des mutations et des croisements aléatoires dont les effets ne sont évalués qu'*a posteriori*, et jamais estimés *a priori*. Pourtant, Dorigo (1994) avait montré de nets gains de performances en introduisant un biais par l'opérateur *mutspec* et l'identification des classeurs oscillants. Nous croyons que des biais utilisant des estimations similaires à celles de YACS offrent des gains en performance supérieurs parce qu'ils permettraient de choisir quels attributs spécialiser et pas seulement quels classeurs spécialiser. De tels biais sont de nouvelles heuristiques.

[28] dans la mesure où l'ajout d'un `#` dans sa partie `Condition` le mènerait à osciller

3.5.3 Heuristiques ou algorithmes génétiques ?

Dans ACS, la problématique de l'anticipation avait conduit Stolzmann à définir des heuristiques inspirées par les théories psychologiques de Hoffmann. Ces heuristiques sont utilisées pour la spécialisation, et la généralisation est dévolue à un algorithme génétique. Dans YACS, tous les mécanismes présidant à la création des classeurs sont des heuristiques.

Les résultats expérimentaux présentés dans la section 3.3.3 montrent un net avantage de performances pour YACS, par rapport à ACS. Pour autant, les heuristiques utilisées par YACS en limitent la portée. En particulier, il se révélerait incapable de prendre en compte des environnements stochastiques. Dès lors se pose la question suivante : vaut-il mieux utiliser des heuristiques ou des algorithmes génétiques ? Le travail présenté ici ne permet pas de répondre définitivement à cette question, mais il la soulève en proposant une alternative aux algorithmes génétiques, et en affichant des performances accrues par rapport à ACS.

Cette question ne concerne pas que les systèmes de classeurs dévolus à l'anticipation et l'apprentissage latent d'un modèle de l'environnement, mais touche l'ensemble des systèmes de classeurs. Déjà, dans XCS, des heuristiques sont utilisées. Ses mécanismes de couverture et de subsomption utilisés sont des biais introduits dans des algorithmes génétiques. La couverture permet, dès qu'une situation est perçue et qu'aucune partie `Condition` des classeurs ne la prend en compte, de créer un nouveau classeur qui couvre la situation en question. Ce mécanisme de couverture est propre à XCS, et il biaise les algorithmes génétiques purs, dans lesquels les conditions ne sont découvertes que par les opérateurs de mutation et de croisement. Ainsi, la couverture est une heuristique. La subsomption en est une autre. Dès que, parmi deux classeurs fiables, l'un est plus général que l'autre, ce mécanisme induit que seul le premier est conservé. Ces deux heuristiques améliorent notablement les performances de XCS.

En outre, les algorithmes génétiques, même non biaisés par des opérateurs de couverture ou de subsomption, sont en fait des heuristiques pour résoudre les problèmes généraux d'optimisation. Ces heuristiques utilisent des valeurs sélectives conjointement à des opérateurs de croisement et de mutation pour accélérer la convergence par rapport à une recherche aléatoire ou systématique. Dans le cadre des systèmes de classeurs, des opérateurs de spécialisation et de généralisation explicites rendent possible un raffinement des heuristiques associées. Il est alors possible de maîtriser des modifications pourtant aisément identifiables *a priori* comme sous-optimales en ce qui concerne l'amélioration du modèle. En particulier, le remplacement d'un classeur oscillant par un classeur plus général n'a aucune chance d'améliorer la fiabilité du modèle. De telles généralisations ne devraient être effectuées que de manière mesurée, pour élargir l'espace de recherche et éviter des *optima* locaux.

Si les algorithmes génétiques sont très utiles pour atteindre un optimum global et explorent efficacement un très large spectre de l'espace des solutions, on peut en revanche craindre que ce ne soit au détriment de leur rapidité à trouver une solution. L'utilisation d'estimations et d'heuristiques pour guider la recherche de classeurs fiables doit permettre d'accélérer ce processus. Mais pour ne pas souffrir des mêmes problèmes que YACS, on peut définir des heuristiques plus souples pour permettre de prendre en compte des environnements stochastiques ou éviter des *optima* locaux.

Quelques soient les heuristiques ou algorithmes utilisés, nous montrons dans le chapitre suivant que les formalismes d'ACS et de YACS ne permettent pas, de toutes façons, d'exprimer un certain nombre de régularités. Les capacités de ces systèmes à généraliser sont limitées par leur formalisme. Dans le chapitre suivant, nous proposons un nouveau formalisme pour l'apprentissage latent dans les systèmes de classeurs, et nous présentons un nouveau système pour l'utiliser : MACS.

Chapitre 4

MACS : représentation de nouvelles régularités pour l'apprentissage latent

Jusqu'ici, nous avons expliqué comment YACS réalise un apprentissage latent pour construire un modèle des transitions dans l'environnement. Il établit une liste de classeurs, chacun d'eux représentant une ou plusieurs transitions. La généralisation permet de représenter des régularités dans la dynamique des interactions avec l'environnement.

Dans ce chapitre, nous identifions des régularités que ni ACS ni YACS ne peuvent représenter. Nous proposons MACS, qui utilise un nouveau formalisme et des heuristiques très semblables à celles développées pour YACS pour tenir compte de ces nouvelles régularités.

4.1 Les régularités non exploitables par ACS et YACS

Considérons un agent dans un monde de cases. Chaque case peut être libre ou contenir un obstacle. L'agent perçoit les huit cases adjacentes (`0` pour une case libre et `1` pour un obstacle). Les huit attributs perçus sont pris dans le sens des aiguilles d'une montre, en commençant par la case en face de l'agent. L'agent peut avancer ou tourner de 90 degrés[29] dans le sens des aiguilles d'une montre ou dans le sens contraire.

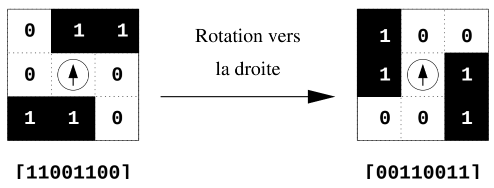

FIG. *4.1: Exemple de transition.*

[29]Ce type d'environnement est différent des mondes de cases définis précédemment puisque l'agent est orienté dans le labyrinthe. Ainsi, à une case du labyrinthe correspondent quatre états, mais les actions sont au nombre de trois, au lieu de huit précédemment.

Comme illustré par la figure 4.1, tourner dans le sens des aiguilles d'une montre induit un décalage des attributs de deux positions vers la gauche. Par exemple, l'agent peut avoir à modéliser des transitions comme [11001100] [↷] [00110011]. Cette transition concerne une action de rotation [↷] qui mène l'agent à percevoir la situation [00110011] juste après la situation [11001100]. Dans ce cas précis, chaque attribut change et le formalisme de YACS est incapable de représenter une telle régularité. Pourtant, le changement dans les situations perçues est effectivement une régularité de la dynamique des interactions : quelle que soit la situation, quand l'agent tourne dans le sens des aiguilles d'une montre, la valeur du premier attribut devient celle du troisième, la valeur du deuxième devient celle du quatrième, etc.

La particularité d'une telle régularité est que la nouvelle valeur d'un attribut dépend de la valeur précédente d'un autre attribut. Or la généralisation utilisée dans ACS ou dans YACS, qui utilise des symboles =, ne permet pas de représenter de telles régularités. Les seules que l'on peut représenter sont telles que la nouvelle valeur d'un attribut dépend de la valeur précédente du même attribut.

Pour exploiter les régularités telles que la nouvelle valeur d'un attribut dépend de l'ancienne valeur d'un autre attribut, nous proposons d'anticiper les attributs indépendamment les uns des autres. Dans la section suivante, nous définissons un formalisme pour dissocier les attributs.

4.2 Le formalisme de MACS

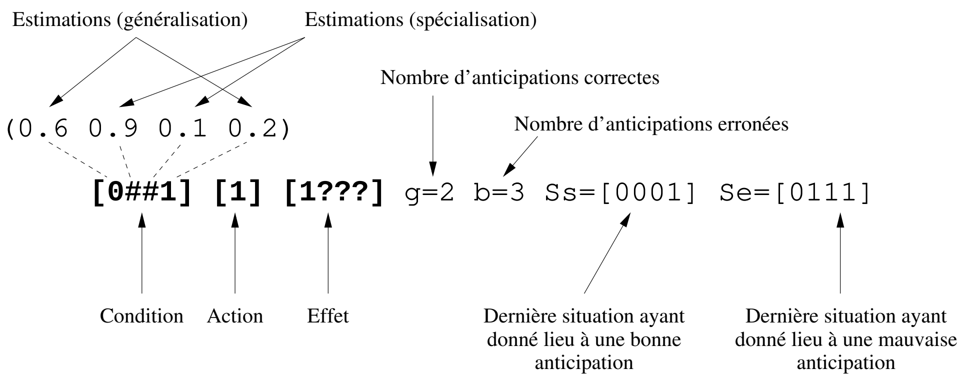

Fig. 4.2: *Exemple de classeur dans MACS.* Les informations associées aux classeurs sont assez semblables à celle utilisées par YACS (voir figure 3.1). La trace T des succès et échecs d'anticipations est remplacée par les deux valeurs g et b.

La figure 4.2 montre un classeur de MACS. Les informations qui y sont rattachées seront détaillées dans la section 4.3. Comme dans YACS, chaque classeur est composé de trois parties. Les parties Condition et Action sont identiques à celles qui sont utilisées dans YACS, et leur sémantique est la même. La partie Condition spécifie des restrictions au domaine d'applicabilité du classeur en utilisant des symboles #, et la partie Action spécifie l'action considérée.

Afin de dissocier les attributs, nous proposons de considérer les parties Effet comme des

[11001100]		$\leftarrow$ Situation
[1#######]	[$\curvearrowright$]	[??????1?]
[#1######]	[$\curvearrowright$]	[???????1]
[##0#####]	[$\curvearrowright$]	[0???????]
[###0####]	[$\curvearrowright$]	[?0??????]
[####1###]	[$\curvearrowright$]	[??1?????]
[#####1##]	[$\curvearrowright$]	[???1????]
[######0#]	[$\curvearrowright$]	[????0???]
[#######0]	[$\curvearrowright$]	[?????0??]
[#######0]	[$\curvearrowright$]	[?????1??]
Anticipations $\rightarrow$		[00110011]
		[00110111]

TAB. *4.1: Mécanisme d'intégration des anticipations partielles dans MACS*

situations, sans symboles =, mais avec de nouveaux symboles ?. Un symbole ? dans une partie `Effet` indique que le classeur ne cherche pas à anticiper l'attribut en question. Par exemple, le classeur `[####1###] [`$\curvearrowright$`] [??1?????]` signifie que, juste après une rotation vers la droite, l'agent perçoit toujours un mur à sa droite quand il a perçu un mur derrière lui, quelle que soit la valeur des autres attributs (grâce au symbole #). Ce classeur ne fournit pas d'information au sujet des nouvelles valeurs d'autres attributs (grâce au symbole ?).

Ce nouveau symbole ?, et son utilisation simultanée avec les symboles #, permettent à un classeur de n'anticiper que quelques attributs et le système global devient capable de représenter de nouvelles régularités. Par rapport au symbole =, le symbole ? dans la partie `Effet` a une sémantique plus proche de celle du symbole #. Pourtant, nous avons choisi de les distinguer parce que leur fonction est très différente au sein du classeur. Si un symbole # permet d'étendre le domaine de validité d'un classeur grâce à la généralisation, un symbole ? a au contraire vocation à restreindre le domaine d'application d'un classeur, en ne lui permettant pas de se prononcer sur certains attributs.

Puisqu'un classeur n'anticipe pas une situation complète mais propose une anticipation partielle, l'unité anticipatrice n'est plus le classeur mais le système entier. Étant donné une situation et une action, un classeur seul ne peut pas prévoir la situation à venir. Le système a besoin d'un mécanisme qui intègre les anticipations partielles de chaque classeur, afin d'anticiper une situation complète, sans aucun symbole ?. Ce mécanisme est illustré par le tableau 4.1.

Dans le mécanisme d'intégration, le système examine les parties `Effet` de chaque classeur dont la partie `Action` est appariée avec l'action souhaitée, et dont la partie `Action` correspond à l'action envisagée. La valeur de chaque attribut différent d'un symbole ? est une anticipation partielle pour cet attribut. Le procédé d'intégration établit toutes les situations possibles à venir, en fonction des anticipations partielles concernant chaque attribut. Si tous les classeurs sont fiables, alors une seule valeur est proposée pour chaque attribut, et ce mécanisme anticipe une seule situation. Dans le cas contraire, comme pour le dernier attribut dans le tableau 4.1, les classeurs proposent plusieurs valeurs possibles pour un attribut. Chacune de ces valeurs possibles multiplie le nombre de situations anticipées.

4.3 La stratégie de découverte des classeurs par MACS

Comme il a été précisé dans la section 4.2, une partie `Effet` peut contenir plusieurs symboles `?` et plusieurs valeurs spécifiques. Mais nous avons adopté un point de vue radical en n'autorisant, dans une partie `Effet`, qu'une et une seule valeur spécifique, c'est-à-dire différente de `?`. Ainsi, chaque classeur ne peut prévoir la valeur que d'un seul attribut.

Dans cette section, nous détaillons chacun des mécanismes d'apprentissage latent de MACS, un nouveau système de classeurs conçu pour utiliser le formalisme qui vient d'être proposé. La plupart d'entre eux sont très semblables à ceux de YACS (Gérard et Sigaud, 2001; Gérard, 2001; Gérard *et al.*, 2003). Il n'y a aucun mécanisme correspondant à la couverture des effets dans MACS.

4.3.1 Mise à jour de la liste des situations perçues

Comme YACS (voir section 3.1), MACS utilise une liste `P` des situations déjà perçues pour apprendre une politique. À chaque situation s_i est associée une valeur $V(s_i)$ qui représente la récompense attendue à partir de s_i. Au début de chaque pas de temps, la situation perçue s_t est ajoutée à cette liste si elle n'y est pas déjà présente.

4.3.2 Évaluation et sélection des classeurs fiables

Pour que l'apprentissage latent élabore un modèle compact et fiable des interactions entre l'agent et l'environnement, il est nécessaire d'évaluer la fiabilité des classeurs, et de les supprimer au besoin. À cette fin, deux valeurs entières `g` et `b` sont associées à chaque classeur :

– `g` [30] pour le nombre d'anticipations correctes depuis la création du classeur ;
– `b` [31] pour le nombre d'erreurs d'anticipation depuis sa création.

À chaque pas de temps, MACS contrôle chaque classeur dont la partie `Condition` est appariée avec la situation précédente s_{t-1}, et dont la partie `Action` est appariée avec a_{t-1} :

– si sa partie `Effet` est appariée avec s_t (un symbole `?` est apparié avec n'importe quelle valeur), alors `g` est augmenté de 1 ;
– si sa partie `Effet` n'est pas appariée avec s_t, alors `b` est augmenté de 1.

Un classeur qui anticipe toujours mal après quelques évaluations est supposé inadéquat et il est supprimé. Ce nombre d'évaluations est un paramètre du système, noté θ_e. Un classeur est

[30]`g` pour « *good* »
[31]`b` pour « *bad* »

supprimé lorsque `g`= 0 et `b`= θ_e. Ce seuil est aussi utilisé pour définir le nombre d'évaluations nécessaires pour décider qu'un classeur est fiable.

4.3.3 Spécialisation des conditions

Comme dans YACS, on dit d'un classeur qu'il oscille quand il anticipe parfois bien, et parfois mal. C'est le cas lorsque sa partie `Condition` est appariée à un trop grand nombre de situations. Il doit donc être spécialisé.

Estimations utilisées par MACS pour la spécialisation

Comme dans YACS, le mécanisme de spécialisation est guidé par des estimations. Une estimation e_s^i est associée à chaque attribut général (symbole `#`) de la partie `Condition` de chaque classeur. Considérons un classeur qui peut être sollicité dans l'anticipation des conséquences d'une action particulière dans différentes situations. Si, dans les situations à partir desquelles le classeur anticipe bien, la valeur de l'attribut de rang i est souvent différente de sa valeur quand le classeur anticipe mal, alors cet attribut est vraisemblablement discriminant. Il est indiqué pour distinguer les situations dans lesquelles le classeur anticipe bien, de celles dans lesquelles il anticipe mal. Ainsi, la partie `Condition` du classeur doit être spécialisée selon cet attribut de rang i, et l'estimation e_s^i qui lui est associée doit avoir une valeur élevée.

Pour calculer ces estimations, chaque classeur garde une trace de la situation s_E précédant la dernière erreur d'anticipation, et de la situation s_S précédant la dernière anticipation correcte. Pour chaque classeur dont la partie `Condition` est appariée avec s_{t-1}, et dont la partie `Action` l'est avec a_{t-1} :

- si le classeur anticipe bien, alors pour chaque attribut :
  - si l'attribut de s_E est égal à l'attribut correspondant de s_{t-1}, alors l'estimation correspondante e_s^i est décrémentée ;
  - si l'attribut de s_E est différent de l'attribut correspondant de s_{t-1}, alors l'estimation correspondante e_s^i est incrémentée ;
- si le classeur anticipe mal, pour chaque attribut :
  - si l'attribut de s_S est égal à l'attribut correspondant de s_{t-1}, alors l'estimation correspondante e_s^i est décrémentée ;
  - si l'attribut de s_S est différent de l'attribut correspondant de s_{t-1}, alors l'estimation correspondante e_s^i est incrémentée.

Les estimations e_s^i sont incrémentées et décrémentées selon une règle de Widrow-Hoff de taux d'apprentissage $\beta_{spéc}$. Ainsi, les incrémentations s'effectuent selon la formule :

$$e_s^i \leftarrow (1 - \beta_{spéc})e_s^i + \beta_{spéc}$$

les décrémentations s'effectuent selon la formule :

$$e_s^i \leftarrow (1 - \beta_{spéc})e_s^i$$

Les valeurs initiales sont 0.5. Un attribut déjà spécialisé prend la valeur par défaut 0.5.

Mécanisme de spécialisation de MACS

Un classeur oscille quand il a été suffisamment évalué, et quand il a parfois bien anticipé, parfois mal. La première condition utilise le seuil θ_e pour définir le nombre d'évaluations nécessaires pour déclarer un classeur comme oscillant. Le classeur est suffisamment évalué lorsque $\mathtt{g}+\mathtt{b} \geq \theta_e$. Un classeur a parfois bien anticipé, et d'autrefois mal, lorsque ni son nombre d'évaluations positives `g` ni son nombre d'évaluations négatives `b` ne sont égaux à 0, c'est-à-dire lorsque $\mathtt{g} \times \mathtt{b} > 0$.

Dès qu'un classeur oscillant est identifié, l'opérateur *mutspec* (voir section 3.2.5) lui est appliqué. Il est ainsi remplacé par plusieurs classeurs plus spécialisés. Certains des classeurs produits par l'opérateur *mutspec* ne seront pas fiables, mais ils seront éliminés plus tard. Un nouveau classeur dont la partie `Condition` n'est appariée avec aucune des situations déjà perçues n'est pas ajouté. Cette propriété est vérifiée en utilisant la liste `P` des situations déjà perçues pendant la vie de l'agent (voir section 4.3.1).

Contrairement à YACS, MACS spécialise les classeurs un par un, sans attendre de confronter les estimations de plusieurs classeurs. Si la prudence de YACS garantit un meilleur guidage de la spécialisation, elle peut en revanche nuire à la vitesse d'apprentissage. C'est notamment le cas lorsque différents classeurs qui doivent être spécialisés ensemble, ne peuvent être évalués que dans des situations très éloignées les unes des autres dans l'environnement[32]. Pour spécialiser, le système doit alors attendre que chacun de ces classeurs ait été suffisamment évalué. Ce problème devient critique dans le cas où la topologie de l'environnement n'est pas ouverte[33]. Pour ce type de topologie, dont un exemple est représenté sur par la figure 4.6, si l'agent explore son environnement de façon aléatoire, il risque de devoir effectuer un grand nombre d'actions avant de recueillir l'information nécessaire à l'évaluation des différents classeurs susceptibles d'être spécialisés.

4.3.4 Généralisation des conditions

Le mécanisme de spécialisation de MACS peut donc produire des classeurs avec une partie `Condition` à un niveau sous-optimal de généralisation, particulièrement dans les cas d'exploration locale, c'est-à-dire quand l'agent n'a encore visité qu'une partie de l'environnement. MACS a donc besoin d'un mécanisme de généralisation qui soit capable de reconsidérer des spécialisations inadéquates. Comme la spécialisation, la généralisation utilise des estimations et des heuristiques afin d'exploiter au mieux l'expérience, pour guider le mécanisme.

[32]Les situations sont éloignées dans le sens où beaucoup d'actions successives sont nécessaires pour aller d'une situation à l'autre.

[33]Les situations définissant le problème peuvent être regroupées en sous-graphes tels que les situations de chaque sous-graphe sont fortement connectées les unes avec les autres par des actions. Lorsque l'environnement est peu ouvert, ces sous-graphes sont faiblement connectés entre eux et il est difficile d'aller d'une région à une autre.

Estimations utilisées par MACS pour la généralisation

Pour chaque classeur, une estimation e_g^i est associée à chaque attribut dont la valeur est spécifique dans la partie `Condition`.

Afin de calculer les estimations e_g^i, MACS contrôle, à chaque pas de temps, les classeurs dont la partie `Action` est égale à a_{t-1}, et dont la partie `Condition` n'est pas appariée à s_{t-1}. Ces classeurs proposent la bonne action, mais leur condition est trop spécialisée pour qu'ils soient sollicités dans une anticipation à partie de s_{t-1}.

Étant donné de tels classeurs, pour chaque attribut spécialisé de la partie `Condition`, MACS vérifie si la partie `Condition` du classeur deviendrait compatible avec s_{t-1} si l'attribut considéré était un symbole `#`. Dans ce cas, l'estimation e_g^i considérée est mise à jour :

- si la partie `Effet` du classeur correspond à l'effet calculé en fonction de s_t et de s_{t-1}, alors un classeur avec une partie `Condition` plus générale aurait bien anticipé et l'estimation est incrémentée.
- si la partie `Effet` du classeur ne correspond pas à l'effet calculé en fonction de s_t et de s_{t-1}, alors un classeur avec une partie `Condition` plus générale aurait mal anticipé et l'estimation est décrémentée.

Les estimations sont incrémentées et décrémentées selon une règle de Widrow-Hoff de taux d'apprentissage $\beta_{gén}$. Les incrémentations s'effectuent selon la formule :

$$e_g^i \leftarrow (1 - \beta_{gén})e_g^i + \beta_{gén}$$

les décrémentations s'effectuent selon la formule :

$$e_g^i \leftarrow (1 - \beta_{gén})e_g^i$$

Les valeurs initiales sont 0.5. La valeur par défaut d'un attribut général est 0.5.

Mécanisme de généralisation de MACS

Avec ce mécanisme et les estimations e_g^i, MACS est capable de déterminer si un attribut d'une partie `Condition` peut être généralisé ou non, en fonction des interactions du système avec l'environnement.

MACS ne généralise que les classeurs fiables. À chaque pas de temps, il cherche à généraliser les classeurs dont la partie `Effet` est appariée à s_t. Ces classeurs sont les seuls susceptibles d'avoir bien anticipé au pas de temps précédent, et d'avoir ainsi vu augmenter leur nombre `g` d'anticipations correctes, de manière à être considérés comme plus fiables par le système.

Le processus de généralisation opère donc sur des groupes de classeurs avec les mêmes parties `Effet` et `Action`. Un tel groupe n'est examiné que lorsque chacun de ses classeurs a été suffisamment évalué pour être considéré comme fiable. Pour être fiable, un classeur ne doit jamais avoir mal anticipé, et il doit avoir été évalué au moins θ_e fois. Autrement dit, il faut que `g`+`b`$\geq \theta_e$ et que `b`$= 0$.

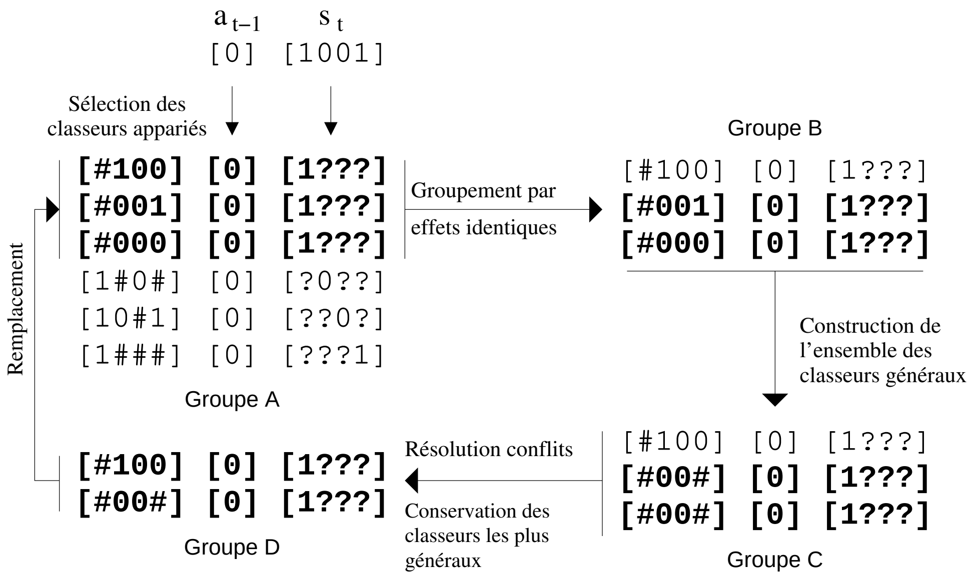

FIG. *4.3: MACS : processus de généralisation.*

La figure 4.3 illustre le mécanisme de généralisation. Dans cette figure, on ne considère qu'un seul de ces groupes, noté groupe A. À partir de chacun de ces groupes A, MACS forme un groupe B de classeurs plus généraux que ceux du groupe A. Le processus de généralisation d'un groupe A continue si chacun des classeurs du groupe A a été suffisamment évalué, et si sa trace T ne contient que des marqueurs attestant de succès dans l'anticipation. Les classeurs du groupe B sont construits de la manière suivante :

- si toutes les estimations e_g^i d'un des classeurs du groupe A sont inférieurs à 0.5, alors il n'est pas un bon candidat pour la généralisation et il est ajouté au groupe B sans modification ;
- dans le cas contraire, un nouveau classeur est créé et ajouté dans le groupe B. Ce nouveau classeur est égal à celui du groupe A, mais l'attribut de sa partie `Condition` doté de la plus grande estimation devient `#`.

Dans l'exemple de la figure 4.3, les estimations n'apparaissent pas, mais leurs valeurs ont conduit à généraliser le quatrième attribut des deuxième et troisième classeurs. En effet, cet attribut pourrait être général dans la mesure où les deux classeurs originaux sont fiables et prédisent les mêmes effets. Le premier classeur reste inchangé et les deux classeurs restants sont généralisés selon leur quatrième attribut.

À ce point, les classeurs du groupe B sont examinés pour détecter d'éventuels conflits avec d'autres classeurs de la liste de tous les classeurs. Deux classeurs sont en conflit si leur partie `Effet` est différente et si leur partie `Action` est la même, alors qu'il existe une situation prise en compte par les parties `Condition` des deux classeurs. Dans ce cas particulier, le système pourrait utiliser indifféremment un des deux classeurs pour anticiper des conséquences contradictoires. Il en résulterait une indécision et une perte de fiabilité du système d'anticipation. Parce que nous voulons éviter de perdre de la connaissance au cours de l'apprentissage, nous voulons éviter ce type de conflits. En conséquence, si un classeur du groupe B est en conflit avec un autre, alors il est remplacé par le classeur correspondant du groupe A. Le groupe résultant est noté C.

De manière à éliminer les doublons et à ne conserver que les classeurs les plus généraux du groupe C, MACS examine itérativement toutes les paires possibles de classeurs du groupe C. Lorsque la partie `Condition` d'un classeur est identique ou plus générale que celle d'un autre classeur, alors le premier classeur est conservé, et le deuxième est écarté du groupe C.

De cette manière, MACS a créé un nouveau groupe C dont chaque classeur est plus général ou égal à un classeur du groupe original, et dont aucun n'est en conflit avec d'autres classeurs. Dès lors, les classeurs du groupe A sont remplacés dans la liste générale par les classeurs du groupe C.

Ce mécanisme permet de remplacer des groupes de classeurs par d'autres classeurs plus généraux et moins nombreux. Il est le pendant du mécanisme de spécialisation. L'association de ces deux mécanismes permet de régler incrémentalement le niveau de généralité des classeurs.

4.3.5 Couverture des conditions

Pour être complet, le modèle doit permettre de rendre compte de chaque transition. Pourtant, il peut être incomplet pour les raisons suivantes :

– le système est initialisé avec une liste de classeurs vide ;
– en cas d'exploration locale, un classeur indispensable peut être supprimé par le mécanisme de sélection, au lieu d'être spécialisé ;
– le mécanisme de spécialisation crée un nouveau classeur uniquement si ce dernier est apparié avec au moins une des situations déjà perçues. Aussi longtemps que l'agent n'a pas perçu chacune des situations possibles, certains classeurs ne seront donc pas créés alors qu'ils auraient été utiles par la suite.

Pour prendre en compte les transitions non couvertes par le modèle, MACS considère, à chaque pas de temps, la transition (s_{t-1}, a_{t-1}, s_t).

Pour chaque attribut, parmi les classeurs dont la partie `Action` est appariée avec a_{t-1}, MACS vérifie s'il existe au moins un classeur dont la partie `Condition` est appariée avec s_{t-1}, et dont la partie `Effet` permet d'anticiper correctement la valeur de l'attribut considéré. Si ce n'est pas le cas, il n'y a aucun classeur pour anticiper l'attribut en question et un nouveau classeur est créé.

La partie `Action` de ce nouveau classeur est égale à a_{t-1}. Sa partie `Condition` est aussi générale que possible, en étant appariée avec s_{t-1}, sans pour autant être appariée avec la partie `Condition` d'aucun autre classeur ayant les mêmes parties `Effet` et `Action` (voir section 3.2.3). La partie `Effet` est telle que son seul attribut différent de `?` est celui qui n'était pas anticipé. Cet attribut prend sa valeur dans s_t.

4.4 Étude expérimentale

Jusqu'ici, nous avons décrit MACS, un nouveau système de classeurs qui utilise un type de généralisation différent de ceux de XCS, ACS ou YACS. Nous avons déjà comparé (voir section 3.3) les capacités d'ACS et de YACS à apprendre un modèle de la dynamique entre l'agent et son environnement. Nous avons montré à cette occasion que, bien qu'utilisant le même formalisme, YACS apprenait plus vite qu'ACS, tout en généralisant mieux.

Dans cette partie, nous comparons les capacités de YACS et de MACS à extraire des modèles compacts des environnements Maze228, Maze252, Maze288 et Maze324, décrits dans la section suivante. Les résultats sont donnés dans la section 4.4.2 et discutés dans la section 4.4.3. Ces expériences ne concernent que l'apprentissage latent. Nous présentons des résultats concernant l'apprentissage par renforcement dans le chapitre 5. Les résultats sont donnés dans la section 4.4.2 et discutés dans la section 4.4.3.

4.4.1 Les environnements Maze228, Maze252, Maze288 et Maze324

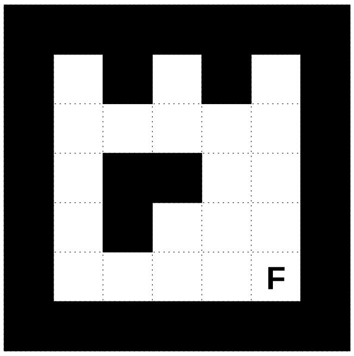

FIG. *4.4: L'environnement Maze228*

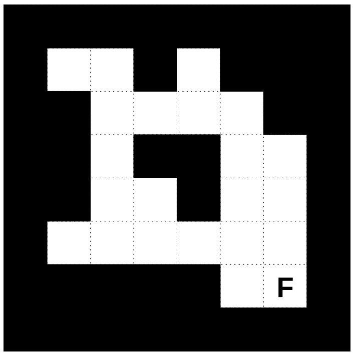

FIG. *4.5: L'environnement Maze252*

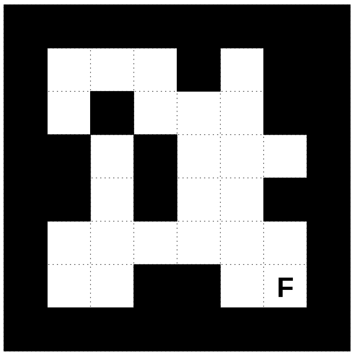

FIG. *4.6: L'environnement Maze288*

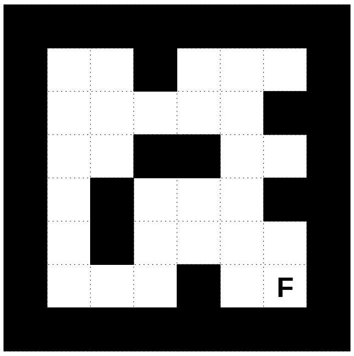

FIG. *4.7: L'environnement Maze324*

Maze228, Maze252, Maze288 et Maze324 sont des mondes de cases. Chaque case peut être vide, contenir un obstacle ■ ou de la nourriture F. L'agent est situé sur une case, et à la différence des environnement Maze4 et Maze6 de la section 3.3.2, il est orienté dans une des quatre directions cardinales. Il perçoit sa situation comme l'agrégation de neuf attributs, huit correspondant aux cases directement autour de lui et un concernant la case sur laquelle il se trouve. Chaque attribut peut prendre trois valeurs : 0 (case vide), 1 (obstacle) ou 2 (nourriture). Les attributs sont ordonnés conformément à la figure 4.8. L'agent a trois actions à sa disposition : tourner d'un quart de tour sur sa droite ou sur sa gauche, ou avancer d'une case. Dans ce cas, si la case devant lui contient un obstacle, l'agent reste dans la même case.

0	1	2
3	4	5
6	7	8

$\longrightarrow$ 012345678

FIG. *4.8: Ordre des attributs dans Maze228, Maze252, Maze288 et Maze324.* Cet ordre suppose que l'agent soit situé dans la case 4, et qu'il fasse face au nord. Il est différent de celui utilisé dans l'exemple de la figure 4.1. Le premier attribut correspond à la case en face, à gauche de l'agent, et le second correspond à la case en bas, à droite de l'agent.

La topologie des environnements étudiés est illustrée par les figures 4.4, 4.5, 4.6 et 4.7. Quantitativement, Maze228 contient 19 cases vides à partir desquelles il est possible d'agir (toutes les cases vides). L'agent peut être situé dans chacune de ces cases selon quatre directions et, dans chacun de ces cas, il a l'opportunité de choisir entre trois actions. En conséquence, il y a $19 \times 4 \times 3 = 228$ transitions à modéliser pour Maze228. Comme Maze252 présente 21 cases non terminales, cet environnement offre 252 transitions. De même, Maze288 et Maze324 présentent respectivement 25 et 28 cases vides, donc 288 et 324 transitions possibles.

D'un point de vue qualitatif, la particularité topologique de Maze288 par rapport à Maze228, Maze252 et Maze324, est qu'il est moins ouvert. Sa partie gauche est constituée de deux zones, chacune accessible d'une seule manière. Par contre, dans les autres environnements, il est plus facile de circuler d'une zone à l'autre. Cette particularité de Maze288 conduit un agent guidé de manière aléatoire à parcourir moins régulièrement l'ensemble des situations possibles.

Les expériences sont divisées en essais. L'agent commence un essai dans une case vide choisie au hasard, avec une orientation aléatoire. L'essai est terminé lorsque l'agent atteint la case avec de la nourriture. Celle-ci ne disparaît pas, l'agent reçoit une récompense de 1, perçoit la situation résultante et un nouvel essai commence.

Dans les expériences présentées ici, à chaque pas de temps, YACS et MACS choisissent une action aléatoirement entre les trois actions disponibles.

4.4.2 Résultats expérimentaux

Pour rendre compte de l'évolution de la fiabilité et de la complétude du modèle au cours du temps, comme dans la section 3.3, nous utilisons une mesure du pourcentage de connaissance élaborée par le modèle. Pour chaque transition possible dans l'environnement, nous vérifions si le système de classeurs la modélise correctement, c'est-à-dire s'il anticipe une seule situation :

celle qui est effectivement observée. Le pourcentage de connaissance est le taux de transitions bien modélisées par le système. Cette mesure ne peut pas être effectuée par le système lui-même puisqu'elle suppose une connaissance parfaite de l'environnement. Elle n'est possible que dans la mesure où les environnements étudiés sont des environnements simulés.

Tous les résultats présentés sont des moyennes sur 100 expériences. Pour YACS, comme dans la section 3.3, la taille θ_e de la trace des classeurs est égale à 5 et les taux d'apprentissage $\beta_{spéc}$ et $\beta_{gén}$ utilisés dans les règles de Widrow-Hoff, pour permettre la mise à jour des différentes estimations utilisées pour la spécialisation et la généralisation, sont égaux à 0.1. Pour MACS, nous avons également utilisé des taux d'apprentissage de 0.1. Le seuil θ_e est aussi égal à 5 (voir la section 4.3).

Les figures 4.9, 4.11, 4.13 et 4.15 présentent l'évolution moyenne, sur 100 expériences, du pourcentage de connaissance quand YACS et MACS interagissent l'un et l'autre avec Maze228, Maze252, Maze288 et Maze324. Les deux systèmes convergent toujours vers un modèle complet et fiable des transitions. Les figures 4.10, 4.12, 4.14 et 4.16 présentent l'évolution moyenne du nombre de classeurs quand YACS et MACS interagissent l'un et l'autre avec Maze228, Maze252, Maze288 et Maze324.

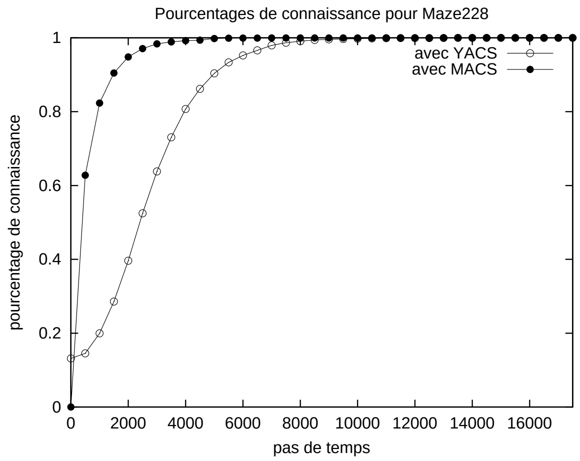

FIG. *4.9: MACS* vs. *YACS en exploration aléatoire : évolution du pourcentage de connaissance pour Maze228.*

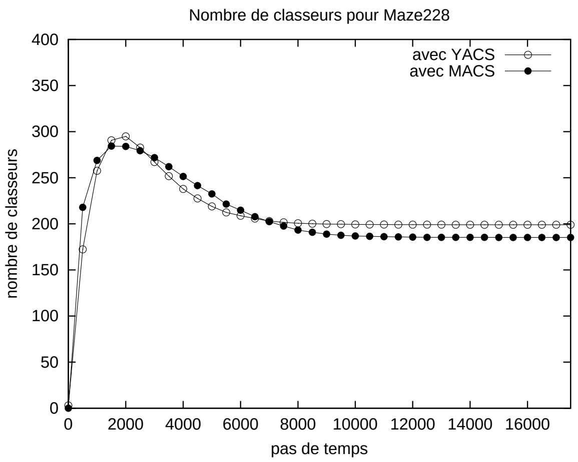

FIG. *4.10: MACS* vs. *YACS en exploration aléatoire : évolution du nombre de classeurs pour Maze228.*

Le tableau 4.2 résume les résultats expérimentaux obtenus au cours des différentes expériences. Ils présentent la moyenne et l'écart type associée, sur les 100 expériences, du temps mis par YACS et MACS pour atteindre une connaissance de 0.99 sur l'environnement. Ils montrent aussi la moyenne du nombre de classeurs nécessaires pour modéliser les interactions entre le système et son environnement.

Pour chaque environnement, des tests de Wilcoxon unilatéraux ont servi à tester statistiquement l'hypothèse de l'égalité des temps de convergence avec MACS et avec YACS, contre l'hypothèse selon laquelle les temps de convergence de MACS seraient inférieurs. Les p-valeurs

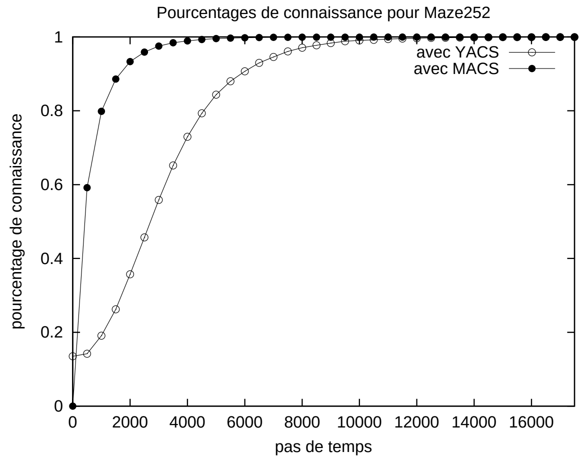

FIG. *4.11: MACS* vs. *YACS en exploration aléatoire : évolution du pourcentage de connaissance pour Maze252.*

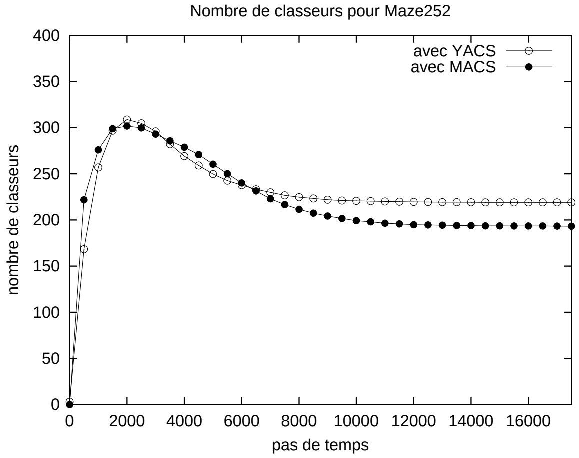

FIG. *4.12: MACS* vs. *YACS en exploration aléatoire : évolution du nombre de classeurs pour Maze252.*

associées à chaque couple d'échantillons sont toutes inférieures à 10^{-5}. Les tests de Wilcoxon acceptent donc, à un seuil de 10^{-5}, l'hypothèse selon laquelle, pour chaque environnement testé, MACS apprend plus rapidement que YACS.

Les figures 4.17 et 4.18 reprennent ces résultats et présentent les relations entre la taille de l'environnement, d'une part, et le nombre de classeurs et le temps de convergence vers un modèle complet et fiable, d'autre part.

4.4.3 Discussion des résultats expérimentaux

Nombre de classeurs

Avec le formalisme de YACS, les classeurs modélisent les transitions comme un tout alors que, dans le formalisme de MACS, chaque classeur anticipe la valeur d'un seul attribut. Dans les environnements avec peu de régularités impliquant des attributs différents, le nombre de classeurs utilisés par MACS devrait être plus important, mais dans Maze228, Maze252, Maze288 et Maze324, MACS converge vers un plus petit nombre de classeurs que YACS, grâce aux nouvelles régularités exprimables dans son formalisme.

Dans les environnements utilisés, il y a deux types de régularités :

– les premières régularités dépendent du type d'environnement utilisé, plutôt que de la topologie particulière de chaque environnement. Par exemple, pour anticiper la nouvelle situation après une rotation, toute l'information nécessaire est entièrement disponible dans la situation initiale, indépendamment de la topologie de l'environnement : chaque attribut prend la valeur d'un autre attribut. Il en va de même pour l'anticipation des attributs correspondant aux trois cases derrière l'agent, et aux trois cases à son niveau, quand il avance (voir figure 4.19) : les attributs au niveau de l'agent prennent les valeurs de ceux

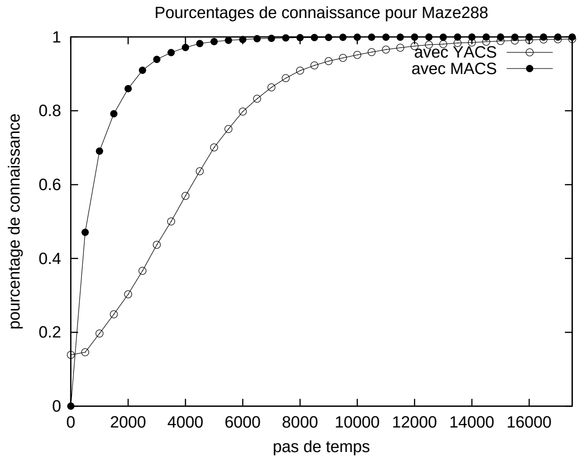

FIG. *4.13: MACS* vs. *YACS en exploration aléatoire : évolution du pourcentage de connaissance pour Maze288.*

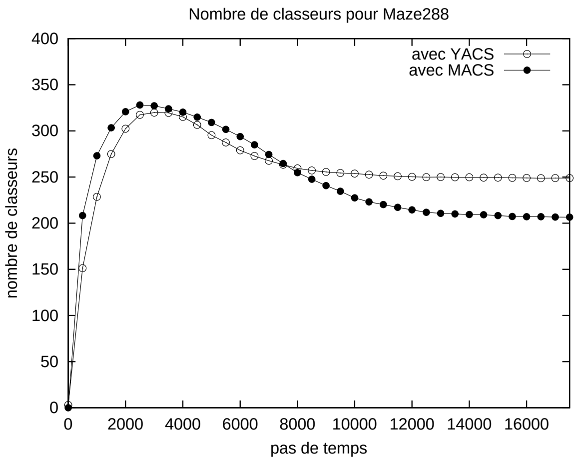

FIG. *4.14: MACS* vs. *YACS en exploration aléatoire : évolution du nombre de classeurs pour Maze288.*

qui sont devant lui, et les attributs derrière l'agent prennent les valeurs de qui sont ceux à son niveau ;

– les secondes régularités sont spécifiques à la topologie de chaque environnement. En effet, pour anticiper la valeur des attributs correspondant aux trois cases devant l'agent, il ne suffit pas de recopier l'ancienne valeur d'autres attributs, et la valeur anticipée est différente pour chaque environnement.

Dans MACS, chaque transition faisant intervenir une régularité du premier type est modélisée simplement, avec le même nombre de classeurs, indépendamment de la taille de l'environnement. Les trois classeurs suivants suffisent, par exemple, à anticiper la valeur du premier attribut après une rotation vers la gauche[34] :

```
[#|#|#|#|#|#|0|#|#] [1] [0|?|?|?|?|?|?|?|?]
[#|#|#|#|#|#|1|#|#] [1] [1|?|?|?|?|?|?|?|?]
[#|#|#|#|#|#|2|#|#] [1] [2|?|?|?|?|?|?|?|?]
```

Puisque YACS ne peut pas représenter les régularités impliquant plusieurs attributs, il a besoin de plus de classeurs pour modéliser ces transitions, à mesure que la taille de l'environnement augmente.

Toutefois, le formalisme de MACS n'utilise pas de symbole =. En conséquence, les régularités comme « *se déplacer vers un mur ne provoque aucun changement dans la situation* », sont représentées avec plus de classeurs dans MACS que dans YACS : un pour chaque valeur de chaque

[34]Nous rappelons que dans les expériences concernant MACS, décrites à la section 4.4, le système perçoit neuf attributs, et non pas huit, et que leur ordre est différent de celui de l'exemple de la section 4.1. Ainsi, le premier attribut représente la case en face et à gauche de l'agent, et le septième représente la case de derrière, à gauche de l'agent.

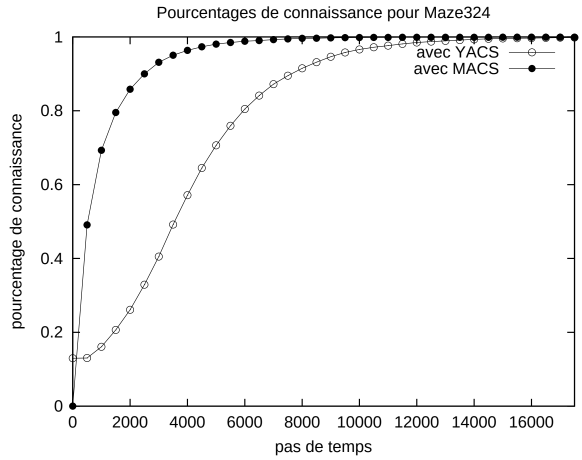

FIG. *4.15: MACS* vs. *YACS en exploration aléatoire : évolution du pourcentage de connaissance pour Maze324.*

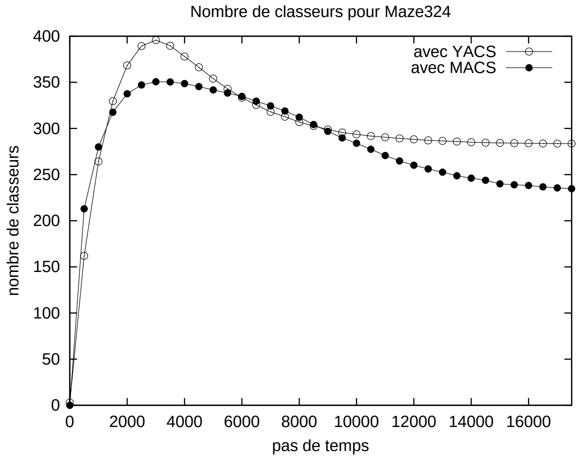

FIG. *4.16: MACS* vs. *YACS en exploration aléatoire : évolution du nombre de classeurs pour Maze324.*

attribut. En revanche, le nombre de classeurs nécessaires pour anticiper ce type de régularités est constant, quelle que soit la taille de l'environnement. Par exemple, les deux classeurs suivants permettent d'anticiper la valeur du dernier attribut, pour n'importe quel environnement de même type que Maze238, Maze252, Maze288 et Maze324 :

```
[#|1|#|#|#|#|#|#|0] [0] [?|?|?|?|?|?|?|?|0]
[#|1|#|#|#|#|#|#|1] [0] [?|?|?|?|?|?|?|?|1]
```

Dans les parties `Condition`, le deuxième attribut indique un mur en face de l'agent. Dans ce cas, la valeur du dernier attribut est inchangée lorsque l'agent avance. Il y a deux classeurs de ce type par attribut, et ce nombre est constant.

Les seuls attributs que MACS anticipe avec un grand nombre de classeurs, correspondent aux trois cases en face de l'agent, quand il avance et qu'aucun mur ne se trouve juste en face de lui. L'information à modéliser dépend alors de la topologie particulière de chaque environnement, plutôt que de leur dynamique générale. Par exemple, pour anticiper la valeur de second attribut, dans Maze228, MACS utilise les 36 classeurs suivants :

```
[#|#|#|#|#|0|#|#|2] [0] [?|0|?|?|?|?|?|?|?]      [0|0|1|#|#|#|#|1|#] [0] [?|0|?|?|?|?|?|?|?]
[#|#|0|0|#|1|0|0|0] [0] [?|0|?|?|?|?|?|?|?]      [1|#|#|#|#|0|#|2|0] [0] [?|0|?|?|?|?|?|?|?]
[#|0|#|1|#|1|0|#|#] [0] [?|0|?|?|?|?|?|?|?]      [1|#|#|1|#|0|0|#|1] [0] [?|0|?|?|?|?|?|?|?]
[#|0|0|#|#|#|1|0|1] [0] [?|0|?|?|?|?|?|?|?]      [1|#|1|1|#|0|1|0|0] [0] [?|0|?|?|?|?|?|?|?]
[#|0|1|#|#|#|1|1|1] [0] [?|0|?|?|?|?|?|?|?]      [1|0|#|#|#|1|1|#|1] [0] [?|0|?|?|?|?|?|?|?]
[0|#|#|#|#|2|#|#|#] [0] [?|0|?|?|?|?|?|?|?]      [1|0|#|1|#|#|1|1|1] [0] [?|0|?|?|?|?|?|?|?]
[0|#|#|0|#|1|0|2|1] [0] [?|0|?|?|?|?|?|?|?]      [1|0|1|#|#|1|1|#|#] [0] [?|0|?|?|?|?|?|?|?]
[0|#|#|1|#|0|1|#|0] [0] [?|0|?|?|?|?|?|?|?]      [#|#|#|#|#|#|0|1|1] [0] [?|1|?|?|?|?|?|?|?]
[0|#|#|2|#|0|#|1|1] [0] [?|0|?|?|?|?|?|?|?]      [#|#|#|0|#|#|1|#|0] [0] [?|1|?|?|?|?|?|?|?]
[0|0|#|#|#|1|0|0|#] [0] [?|0|?|?|?|?|?|?|?]      [#|#|0|0|#|0|#|#|1] [0] [?|1|?|?|?|?|?|?|?]
```

Expérience	Nombre de classeurs (moyenne)	Temps de convergence	
		Moyenne	Écart type
Maze228 - YACS	199	7533	1079
Maze228 - MACS	185.2	2990	786
Maze252 - YACS	219	9509	1715
Maze252 - MACS	193.3	3679	987
Maze288 - YACS	249	14978	4395
Maze288 - MACS	206.4	4944	1403
Maze324 - YACS	283	12515	1890
Maze324 - MACS	231.1	5508	1315

TAB. *4.2: Synthèse des résultats expérimentaux concernant la comparaison entre YACS et MACS*

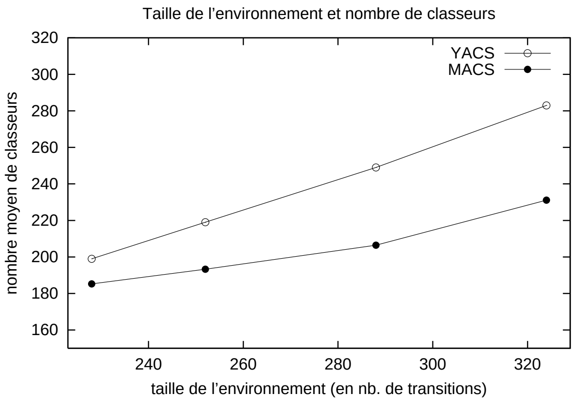

FIG. *4.17: Nombre de classeurs obtenus en fonction de la taille de l'environnement.*

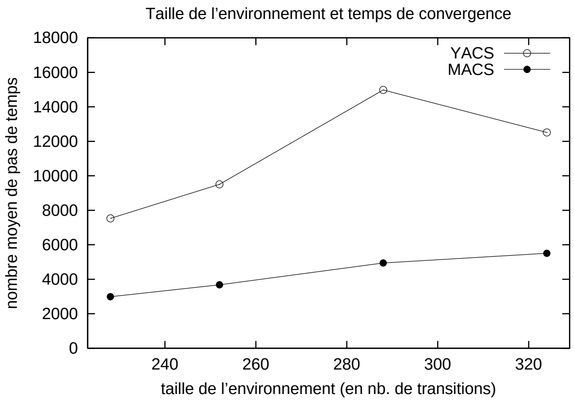

FIG. *4.18: Temps de convergence en fonction de la taille de l'environnement.*

```
[#|#|0|1|#|#|0|0|#]  [0]  [?|1|?|?|?|?|?|?|?]
[#|#|0|1|#|1|1|0|0]  [0]  [?|1|?|?|?|?|?|?|?]
[#|#|1|1|#|#|1|0|1]  [0]  [?|1|?|?|?|?|?|?|?]
[#|0|#|#|#|#|2|0|#]  [0]  [?|1|?|?|?|?|?|?|?]
[#|1|#|#|#|#|#|1|#]  [0]  [?|1|?|?|?|?|?|?|?]
[#|1|#|0|#|#|#|#|#]  [0]  [?|1|?|?|?|?|?|?|?]
[#|1|#|1|#|#|#|0|#]  [0]  [?|1|?|?|?|?|?|?|?]
[#|1|#|2|#|#|#|#|#]  [0]  [?|1|?|?|?|?|?|?|?]
[#|2|#|#|#|0|#|#|0]  [0]  [?|1|?|?|?|?|?|?|?]
```

```
[0|#|0|#|#|#|0|1|#]  [0]  [?|1|?|?|?|?|?|?|?]
[0|0|0|1|#|#|#|#|1]  [0]  [?|1|?|?|?|?|?|?|?]
[0|2|#|#|#|#|#|0|1]  [0]  [?|1|?|?|?|?|?|?|?]
[1|#|#|#|#|#|#|1|0]  [0]  [?|1|?|?|?|?|?|?|?]
[1|#|1|0|#|#|0|#|#]  [0]  [?|1|?|?|?|?|?|?|?]
[0|#|#|0|#|1|1|0|#]  [0]  [?|2|?|?|?|?|?|?|?]
[1|0|0|1|#|0|#|0|0]  [0]  [?|2|?|?|?|?|?|?|?]
```

Afin de diminuer le nombre de classeurs nécessaires pour anticiper dans ces cas précis, une solution pourrait consister à pourvoir MACS d'un mécanisme qui lui permette de créer des parties `Effet` avec plusieurs symboles spécialisés, comme nous l'avions initialement proposé dans la section 4.2. Bien que cette solution conduise à une représentation plus compacte, la solution

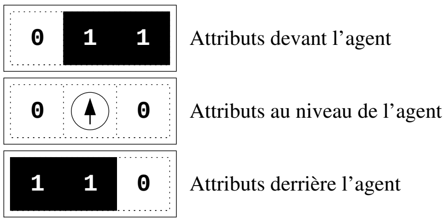

FIG. *4.19: Attributs correspondant aux cases devant, derrière et au niveau de l'agent, en fonction de son orientation.*

retenue dans MACS offre une plus grande flexibilité, et les opérateurs de spécialisation et de généralisation restent simples.

De plus, en restreignant les anticipations partielles à un seul attribut, il est possible de définir des groupes de classeurs comme celui ci-dessus, anticipant chacun un seul attribut. Pour comprendre les régularités exploitées par le système, on peut alors s'intéresser à chaque groupe indépendamment des autres, et l'intelligibilité s'en trouve améliorée, même si le nombre de classeurs total est assez grand. C'est à l'intérieur de chacun de ces groupes que MACS utilise la généralisation pour réduire le nombre de classeurs. Dans l'exemple ci-dessus, par exemple, MACS utilise 17 classeurs pour caractériser les situations dans lesquelles la valeur du deuxième attribut devient `0`, alors que dans Maze228, il y a 25 situations à partir desquelles avancer conduit le deuxième attribut à valoir `0`.

Dans les environnements testés, il n'y a que trois attributs pour lesquels un grand nombre d'attributs est nécessaire, et le nombre total de classeurs reste inférieur avec MACS. Toutefois, dans des environnements tels que les attributs de ce type sont plus nombreux, MACS construit un nombre total de classeurs supérieur, même si à l'intérieur de chaque groupe, la généralisation est utilisée. Par exemple, dans l'environnement Maze2232 de la figure 5.13, l'agent perçoit 25 attributs, dont 5 nécessitent un nombre important de classeurs pour être anticipés correctement. Dans ce cas, le nombre de classeurs découverts par MACS est bien plus important que pour YACS (voir figure 5.15), et la solution globale perd en intelligibilité. Toutefois, cet inconvénient est en partie compensé par la possibilité de considérer cinq sous-systèmes.

Vitesse d'apprentissage

Durant les 1 000 premiers pas de temps des expériences, pour chacun des environnements testés, le pourcentage de connaissance croît très rapidement avant de ralentir. De plus, le modèle complet est appris beaucoup plus rapidement dans MACS que dans YACS.

Le très rapide gain en connaissances au début de l'apprentissage tient au fait que de nombreuses régularités ne nécessitent qu'une seule spécialisation pour être modélisées. Ces régularités interviennent, par exemple, dans les transitions impliquant une action de rotation. De plus, dans ce cas, malgré une exploration aléatoire, les exemples pertinents pour apprendre ces transitions sont nombreux. En effet, pour chaque attribut, il y a plusieurs transitions qui permettent d'apprendre à prédire sa nouvelle valeur.

Les anticipations liées à la topologie particulière de chaque environnement nécessitent plusieurs spécialisations successives. MACS met donc plus de temps à les caractériser. De plus, à mesure que les classeurs sont plus spécialisés, les exemples pertinents sont plus rares. Ainsi, en raison de l'exploration aléatoire, le système expérimente beaucoup de transitions qui ne lui apportent aucune information supplémentaire, et l'apprentissage est plus lent.

Pour MACS, malgré cette exploration aléatoire, la vitesse de l'apprentissage d'un modèle complet semble linéaire par rapport à la taille des environnements testés. Par contre, la topologie particulière de Maze288 pose problème à YACS (voir section 4.4.1). La figure 4.18 montre que MACS ne souffre pas du problème évoqué à la section 4.3.3. Ainsi, la spécialisation immédiate des classeurs qui oscillent permet à MACS d'être moins gêné que YACS par la topologie de Maze288, telle que certains sous-graphes situations/actions sont faiblement connectés aux autres sous-graphes.

Afin d'accélérer la convergence de MACS, nous montrons, dans le chapitre suivant, comment utiliser des algorithmes de programmation dynamique dans une architecture *Dyna*, afin de guider l'exploration de l'environnement qui a reposé ici sur une politique aléatoire. Cette exploration active se fonde sur les transitions déjà modélisées, et sur une récompense interne qui dépend du gain en information attendu dans chaque situation. Ce gain est fonction du nombre d'évaluations de chaque classeur dont la partie `Condition` est appariée avec la situation considérée. Avec un tel mécanisme, nous montrons que l'agent peut utiliser un modèle partiel pour accélérer l'apprentissage du modèle complet, en explorant activement son environnement grâce à la rétro-propagation du gain en information attendu dans chaque situation.

4.5 Discussion

Par rapport à ACS, qui nous a tenu lieu de référence dans un premier temps, YACS décorrèle les parties `Condition` et `Effet` des classeurs. MACS va plus loin dans ce sens, en décorrélant les attributs de la partie `Effet`. Il propose ainsi une nouvelle manière de considérer la question de l'anticipation dans les systèmes de classeurs. Mais ici, cette décorrélation ne provient pas des mécanismes, mais du formalisme lui-même.

Lorsque l'on choisit de traiter un type de problème dans lequel les situations sont composées d'attributs, si l'on cherche à anticiper des situations, on peut, comme dans MACS, anticiper les attributs séparément. Le modèle des transitions est alors composé de plusieurs modèles partiels, chacun composé de classeurs anticipant le même attribut. Ce faisant, on gagne une plus grande capacité d'expression et il devient possible de modéliser facilement des régularités impliquant plusieurs attributs. Dès lors, un classeur ne modélise plus une transition à lui seul. En effet, une transition implique tous les attributs, et il devient nécessaire d'utiliser plusieurs classeurs pour pouvoir anticiper.

Le problème consistant à apprendre un modèle de son environnement revient à chercher la fonction de transition $T\ : s_1 \times ... \times s_d \times a_1 \times ... \times a_e \rightarrow s_1 \times ... \times s_d$, les moyens d'y parvenir sont

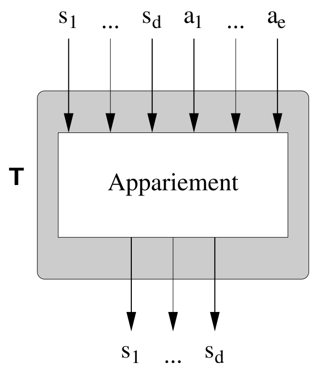

FIG. 4.20: *Architecture de YACS.*

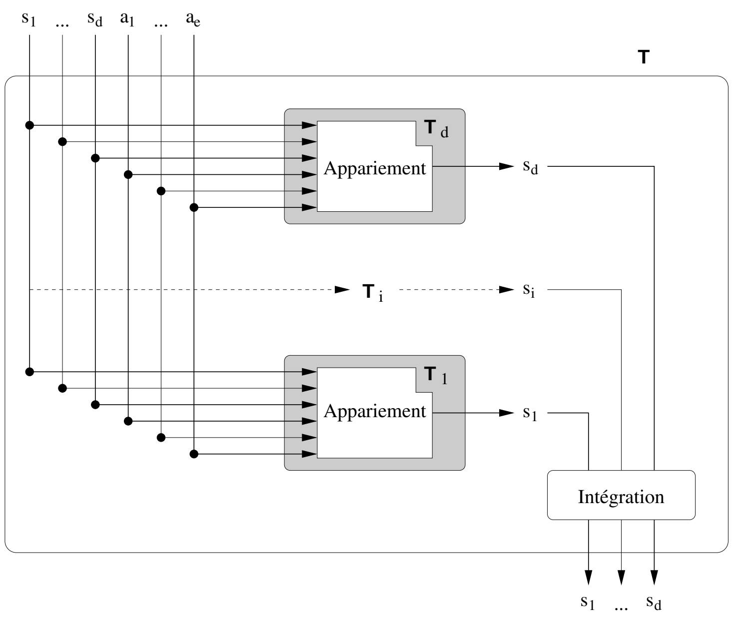

FIG. 4.21: *Architecture de MACS.*

différents. Dans YACS, comme dans ACS, chaque classeur définit une fonction partielle par le jeu des parties `Condition`, qui restreignent le domaine de validité du classeur à un sous-ensemble de $s_1 \times ... \times s_d \times a_1 \times ... \times a_e$. L'ensemble de destination de chacune de ces fonctions partielles reste un sous-ensemble de $s_1 \times ... \times s_d$.

Dans MACS, par contre, chaque classeur ne se prononce que sur un seul attribut s_i. Ainsi, l'ensemble des classeurs prédisant une valeur pour l'attribut s_i définit une fonction $T_i\ : s_1 \times ... \times s_d \times a_1 \times ... \times a_e \to s_i$. MACS peut donc être vu comme un ensemble de d modules, chacun définissant une fonction T_i. L'ensemble des fonctions T_i permet de recomposer la fonction $T\ : s_1 \times ... \times s_d \times a_1 \times ... \times a_e \to s_1 \times ... \times s_d$. En utilisant des classeurs mettant en relation un seul attribut avec tous les autres, plutôt que des classeurs cherchant à modéliser des transitions complètes (avec tous ses attributs), MACS suggère une architecture comme celle illustrée par la figure 4.21, plutôt qu'une architecture comme celle illustrée par la figure 4.20.

Dans ce chapitre, nous avons décrit MACS, qui offre de nouvelles possibilités de généralisation par rapport à YACS et à ACS. De plus, bien qu'utilisant des heuristiques et des estimations très semblables à celles de YACS, il offre des performances accrues pour l'apprentissage d'un modèle du monde. Par ailleurs, si dans les expériences menées jusqu'ici, la politique d'exploration était aléatoire, dans le chapitre suivant, nous montrons comment il est possible d'utiliser l'apprentissage latent de MACS dans une architecture de type *Dyna*, afin d'apprendre rapidement une politique. Nous définissons deux politiques d'exploration et une politique permettant de maximiser la récompense. Nous les organisons de manière hiérarchique pour permettre à la fois une exploration suffisante et un comportement optimal.

Chapitre 5

Le modèle du monde et la politique comportementale avec MACS

Jusqu'ici, nous avons étudié l'apprentissage latent dans le cadre des systèmes de classeurs. Cet apprentissage est caractérisé par le fait qu'il est réalisable en l'absence de récompense. Pour reprendre l'exemple utilisé dans l'introduction, le clown Auguste peut apprendre les conséquences de ses actions indépendamment de la récompense, et construire un modèle comme celui de la figure 5.1. Dans ce chapitre, nous nous intéressons à l'utilisation d'un modèle des transitions dans l'apprentissage d'une politique comme celle de la figure 5.2. La politique obtenue dépend de la récompense immédiate. Celle de la figure 5.2 est la politique qui permet à Auguste de se trouver le plus rapidement possible en position d'entarter le clown blanc. Définir une autre récompense, c'est modifier les objectifs de l'agent, et donc le conduire à construire une autre politique.

Dans ce chapitre, nous utilisons trois récompenses immédiates :

- la première récompense les améliorations du modèle. Elle permet une exploration active de l'environnement ;
- la seconde est la récompense environnementale. Elle spécifie les objectifs de l'agent ;
- la dernière récompense l'exploration des situations qui n'ont pas été rencontrées depuis longtemps. Elle entraîne une exploration systématique et régulière de toutes les situations.

Chacune de ces récompenses immédiates donne lieu à une politique différente, qu'il s'agit de combiner avec les autres pour permettre de sélectionner, à chaque pas de temps, l'action qui satisfait le mieux un compromis entre les trois critères.

Pour la construction de chacune de ces politiques, le modèle de l'environnement est utilisé pour accélérer l'apprentissage de valeurs de renforcement. Pour ce faire, nous utilisons une architecture de type *Dyna* (voir section 1.3.2).

5.1 Politique d'exploration active avec MACS

L'objectif de l'exploration active est de pourvoir l'agent d'une politique lui permettant de maximiser la pertinence des informations tirées de la boucle sensori-motrice. De cette manière,

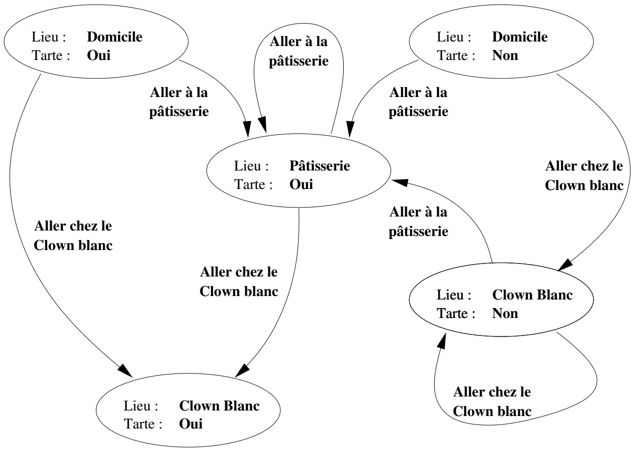

FIG. *5.1: Modèle de l'environnement d'Auguste.* Ce modèle spécifie les conséquences de chacune des actions, mais n'apporte aucune information quand à la source de récompense.

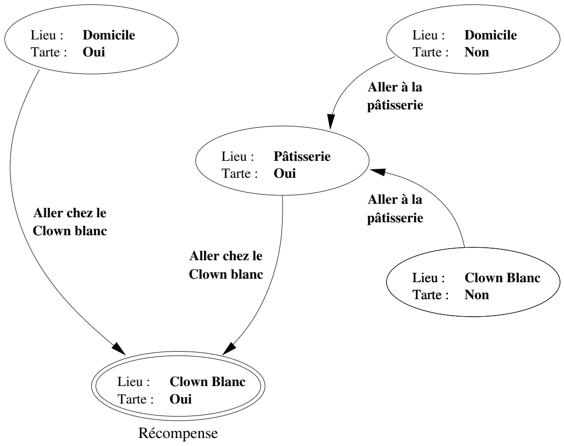

FIG. *5.2: Politique d'Auguste.* Du modèle complet, la politique ne retient que les transitions qui permettent de maximiser la récompense. Dans une situation donnée, l'agent entreprend la situation spécifiée.

l'agent choisit des actions qui lui permettent d'améliorer plus vite son modèle de l'environnement.

5.1.1 La récompense interne immédiate

Pour guider le comportement de l'agent de manière à ce qu'il cherche à améliorer son modèle, nous définissons une fonction de récompense interne $i \ : S \times A \rightarrow \Re$ qui estime le gain immédiat en information, étant donné une situation et une action. Cette fonction permet à MACS de choisir les actions qui maximisent le cumul à long terme des gains immédiats en information. La politique correspondante permet d'effectuer une exploration active de l'environnement.

Dans une situation particulière s_t, lorsque le système doit choisir une action pour maximiser le gain en information, MACS sélectionne tous les classeurs dont la condition est appariée avec s_t. Les conséquences de l'action suggérée par chacun de ces classeurs pourront être évaluées au pas de temps suivant : soit le nombre d'anticipations correctes `g` augmentera, soit le nombre d'erreurs d'anticipations `b` augmentera.

À chacun de ces classeurs appariés avec s_t, MACS associe un niveau d'évaluation $l \in [0, 1]$. L'exploration active a pour but d'augmenter les niveaux d'évaluation, pour que les classeurs soient évalués aussi rapidement que possible. Le niveau d'évaluation associé à un classeur dépend du nombre de succès et d'erreurs d'anticipations du classeur, mais aussi des paramètres rendant compte du nombre d'évaluations qui sont nécessaires pour déclarer un classeur oscillant, inadéquat ou fiable :

– si $b > 0$ et $g > 0$, alors le classeur oscille, mais il peut être utile de l'évaluer plus avant de manière à garantir une meilleure fiabilité des estimations utilisés dans le processus de

Classeurs			(g, b)	Niveaux d'évaluation
[0###]	[0]	[??0?]	$(3, 1)$	1.0
[#1##]	[0]	[???1]	$(1, 1)$	0.5
[##1#]	[0]	[?1??]	$(2, 1)$	0.75
[###0]	[0]	[0???]	$(1, 2)$	0.75
[###0]	[0]	[1???]	$(2, 0)$	0.5

Hypothèses			Niveaux d'évaluation	Gains d'info.
s_t	a_t	s_{t+1}		
[0110]	[0]			
		[0110] ?	$1.0 \times 0.5 \times 0.75 \times 0.75 = 0.28125$	0.71875
		[0111] ?	$1.0 \times 0.5 \times 0.75 \times 0.5 = 0.1875$	0.8125

TAB. *5.1: Calcul des gains d'information immédiats*

spécialisation. Dans ce cas, $l = \min((b + g)/\theta_e, 1)$;

- si $b = 0$ et $g > 0$, alors le classeur a toujours bien anticipé mais peut avoir besoin d'être évalué plus avant pour être déclaré fiable. Dans ce cas, $l = \min(g/\theta_e, 1)$;
- si $b > 0$ et $g = 0$, alors le classeur n'a jamais bien anticipé mais il peut être nécessaire de l'évaluer davantage avant de le supprimer. Dans ce cas, $l = \min(b/\theta_e, 1)$;
- si $b = g = 0$, alors le classeur n'a jamais été évalué et $l = 0$.

Donc, le niveau l est borné entre 0 et 1. Il est égal à 1 si le classeur n'a plus besoin d'être évalué avant d'être supprimé, spécialisé ou généralisé. Il est strictement inférieur à 1 lorsque le classeur doit encore être évalué. Le seuil θ_e représente le nombre d'évaluations nécessaires pour prendre une décision concernant un classeur.

Comme dans le tableau 5.1 avec l'action `[0]`, les classeurs appariés avec s_t sont groupés par actions semblables. Pour chacune des actions possibles, MACS construit l'ensemble $S_{s_t,a}$ de situations susceptibles de survenir au pas de temps suivant, d'après le modèle de l'environnement. Dans le cas où les classeurs ne sont pas fiables, cet ensemble peut contenir plusieurs situations. Seules les situations déjà rencontrées et présentes dans l'ensemble P, sont considérées dans $S_{s_t,a}$. Chaque triplet (s_t, a, s_{t+1}), tel que $s_{t+1} \in S_{s_t,a}$, est une des transitions qui pourraient être observées si l'agent entreprenait l'action a dans la situation s_t.

Nous définissons le niveau d'évaluation associé à une telle transition (s_0, a, s_1) comme le produit des niveaux d'évaluation de chacun des classeurs impliqués dans la prédiction de cette transition :

$$l(s_0, a, s_1) = \prod_{c \approx (s_0,a,s_1)} l_c$$

Le symbole $\approx$ est utilisé pour représenter l'appariement des parties d'un classeur avec l'action et les situations successives considérées. Les classeurs c appariés avec (s_0, a, s_1) sont tels que leur partie `Condition` est appariée avec s_0, leur partie `Action` est appariée avec a, et leur partie `Effet` est appariée avec s_1. Nous définissons le gain en information associé à cette transition comme étant :

$$R_i(s_0, a, s_1) = 1 - l(s_0, a, s_1)$$

Nous définissons le gain en information associé à une situation et à une action comme le gain maximal parmi ceux qui sont associés à chacune des situations anticipées pour cette action :

$$R_i(s_0, a) = \max_{s1 \in S_{s_0,a}} R_i(s_0, a, s_1)$$

Si le modèle ne fournit pas à MACS au moins une situation anticipée s_1, à cause de son incomplétude, alors $R_i(s_0, a)$ prend la valeur par défaut 1, qui est la valeur maximale pour le gain immédiat en information.

Cette valeur $R_i(s_0, a)$ est utilisée comme une récompense immédiate dans le processus d'élaboration d'une politique d'exploration active. Elle est calculée en fonction du seul modèle, et ne dépend pas directement de l'environnement. C'est donc une récompense interne.

5.1.2 Construction d'une politique comportementale d'exploration active

Pour réaliser une exploration active, MACS doit maximiser le cumul des gains en information sur le long terme. MACS utilise ses capacités d'anticipation de manière à déterminer comment agir pour obtenir des informations pertinentes dans le futur, même s'il se trouve dans une situation telle qu'aucune récompense interne immédiate n'est possible.

Grâce à son modèle de l'environnement et aux récompenses internes, MACS est en mesure de déterminer une politique grâce à des méthodes de programmation dynamique. Il rétro-propage donc la récompense interne immédiate en utilisant une variante de l'algorithme Value Iteration (voir section 1.2.2). Cependant, deux problèmes se posent dans la mesure où la détermination de la politique utilise deux informations très instables et pas toujours fiables. Conformément à une architecture *Dyna*, MACS utilise donc un modèle de l'environnement pour, à chaque pas de temps, mettre à jour plusieurs valeurs associées aux situations.

Le premier problème vient de ce que, durant la période d'apprentissage, MACS utilise le modèle incomplet qu'il est en train de construire pour aller chercher les informations qui lui permettront de raffiner plus vite ce même modèle. L'utilisation d'un modèle incomplet et pas toujours fiable peut occasionner des problèmes. Par exemple, il est possible qu'un agent définisse une politique d'exploration en prenant en compte des transitions qui ne peuvent pas être observées dans la réalité. C'est souvent le cas lorsque les classeurs sont encore trop généraux. Dans ce cas, il peut arriver que la politique construite à partir du modèle et de la fonction de récompense immédiate mène l'agent à répéter cycliquement la même séquence d'actions. En effet, si les actions entreprises par l'agent ne lui permettent pas de remettre en cause les transitions problématiques, il peut se trouver piégé dans un optimum local, sans possibilité de reconsidérer les éléments du modèle qui l'y ont amené. En conséquence, le calcul de la politique doit se faire avec prudence, puisqu'il est fondé sur des informations manquant de fiabilité.

Le second problème est que, pendant l'apprentissage, le modèle des transitions est sans cesse amélioré et les récompenses internes changent sans arrêt. Donc, la politique d'exploration de l'agent est très instable au cours des pas de temps successifs. Toutefois, malgré cette instabilité, il n'est pas souhaitable de reconstruire entièrement une politique optimale à chaque pas de temps, dans la mesure où nous voulons que l'agent reste réactif. C'est pour cette raison que nous utilisons des méthodes itératives de planification issues de la programmation dynamique. La particularité de ces méthodes itératives est qu'elles peuvent donner un résultat intermédiaire exploitable à n'importe quelle étape du calcul. À chaque pas de temps, MACS met à jour chacune des valeurs associées aux différentes situations rencontrées. Sans jamais être tout à fait optimale, la politique résultante reste cependant acceptable, dans la mesure où l'exploration active permet à l'agent d'apprendre un modèle plus vite que s'il suivait une politique aléatoire.

À chaque situation de l'ensemble P des situations déjà rencontrées, est associée une valeur représentant le cumul dégressif des gains en informations attendus à partir de cette situation. Ainsi, l'ensemble P sert à stocker, d'un pas de temps sur l'autre, les valeurs calculées itérativement par l'algorithme de programmation dynamique. Les transitions sont déduites de la liste des classeurs grâce à la méthode d'intégration (voir section 4.2) avec les récompenses internes immédiates qui leur sont associées.

À chaque pas de temps, MACS simule une prise de décision à partir de chaque situation s_0 de l'ensemble P. Pour chaque action possible a, la récompense immédiate associée à s_0 et a est $i(s_0, a)$ (voir section 5.1.1). La récompense attendue associée à chaque transition (s_0, a, s_1) est la valeur $V_i(s_1)$, atténuée par un facteur γ, associée à s_1 dans l'ensemble P.

Construire une politique sur la base d'un modèle imparfait peut mener à une politique très sous-optimale, voire à des comportements cycliques et sans fin. Pour éviter de tels problèmes, MACS est prudent vis-à-vis des récompenses attendues.

Étant donné une situation s_0 et une action a, nous définissons la récompense attendue en termes de gain d'information comme :

$$E_i(s_0, a) = \min_{s_1 \in S_{s_0,a}} V_i(s_1)$$

Le *min* dans cette équation reflète la prudence de MACS. En effet, pour chaque action, il considère la interne minimale qu'il est en droit d'attendre, en fonction du modèle. C'est cette récompense minimale qu'il cherche à maximiser en sélectionnant l'action appropriée. De cette information, MACS déduit la qualité associée à la situation s_0 et à l'action a en utilisant une variante des équations de Bellman[35] (voir section 1.2.2) :

$$Q_i(s_0, a) = R_i(s_0, a) + \gamma E_i(s_0, a)$$

Dès lors, MACS met à jour la nouvelle valeur associée à s_0 :

$$V_i(s_0) = \max_{a \in A} Q_i(s_0, a)$$

[35]La différence avec les équations de Bellman originales tient dans l'utilisation de la valeur minimum comme récompense attendue, pour traduire la prudence de MACS.

Le facteur γ est le facteur d'amortissement. Il joue le même rôle que dans l'équation 1.2.

Un système d'apprentissage par renforcement qui ne dispose pas d'un modèle de l'environnement, comme *Q-learning*, ne peut mettre à jour les valeurs et les qualités que lorsqu'il agit effectivement dans l'environnement (voir section 1.3.1). Le modèle permet d'anticiper les conséquences de ses actions, et de simuler des actions pour mettre à jour plusieurs valeurs par pas de temps.

La politique est déterminée en choisissant de manière déterministe, pour chaque situation s, l'action a telle que $Q_i(s, a)$ est maximale.

5.2 Politique de maximisation des récompenses reçues avec MACS

La politique décrite à la section précédente permet d'explorer l'environnement de manière active, de manière à ce qu'un modèle fiable de l'environnement soit appris plus vite que si l'exploration était aléatoire. Dans cette section, nous décrivons comment MACS apprend une politique lui permettant de remplir les objectifs tels qu'ils sont définis par la récompense environnementale. La première politique permet donc à auguste d'apprendre rapidement le modèle de la figure 5.1. La politique correspond à celle de la figure 5.2. Elle permet à Auguste de maximiser sa récompense, c'est-à-dire d'entarter le clown blanc au plus vite.

À chaque pas de temps, MACS reçoit de son environnement une récompense scalaire r_t et une nouvelle situation s_t, comme résultat de l'action a_{t-1} entreprise alors que la situation s_{t-1} était perçue. Cette récompense environnementale immédiate est associée à s_t dans l'ensemble des situations perçues P. Nous la notons $R_p(s_t)$. Cette fonction R_p représente les buts du système, définis par les récompenses environnementales. Ici encore, nous utilisons des techniques de planification itératives pour permettre à MACS d'utiliser son modèle de l'environnement pour remplir ses objectifs.

Comme dans la section 5.1.2, MACS simule plusieurs actions à chaque pas de temps. Quand MACS simule une action à partir de la situation s_0, il utilise le modèle des transitions fourni par la liste des classeurs et le mécanisme d'intégration. Avec ces informations, MACS calcule, pour chaque action possible a, l'ensemble des situations anticipées s_1. Une valeur de récompense attendue $V_p(s)$ est associée à chaque situation de l'ensemble P. Cette valeur mesure à quel point atteindre la situation correspondante permettra de maximiser le cumul des récompenses futures.

Le processus d'apprentissage par renforcement proprement dit met à jour ces valeurs de façon itérative, grâce au modèle des transitions et aux récompenses immédiates. Ici encore, le processus d'apprentissage est prudent, puisque durant l'apprentissage latent, le modèle est incomplet et manque de fiabilité.

En premier lieu, étant donné toutes les transitions possibles à partir de s_0, MACS calcule les qualités associées aux actions :

$$Q_p(s_0, a) = \min_{s_1 \in S_{s_0,a}} [R(s_1) + \gamma V_p(s_1)]$$

Ici encore, le facteur γ est un facteur d'amortissement, qui joue le même rôle que dans l'équation 1.3. Grâce à ces valeurs, MACS met à jour la valeur associée à la situation considérée s_0 :

$$V_p(s_0) = \max_{a \in A} Q_p(s_0, a)$$

Pendant la phase de sélection de l'action, quand la situation perçue est s_t, MACS choisit l'action propre à maximiser $Q_p(s_t, a)$. De cette manière, MACS construit une politique d'exploitation qui lui permet de choisir la meilleure action pour remplir ses objectifs, tels qu'ils sont définis par la récompense environnementale.

5.3 Politique d'exploration systématique avec MACS

Nous définissions ici une nouvelle récompense immédiate pour permettre à l'agent de visiter régulièrement toutes les situations possibles. De manière assez similaire à *DynaQ+*, une valeur $R_x(s)$ est associée à chaque situation déjà perçue s. Une valeur $R_x(s)$ est plus importante lorsque s n'a pas été perçue depuis un plus grand nombre de pas de temps. Lorsque MACS perçoit une nouvelle situation s, $R_x(s)$ est mise à 0. Chacune des autres situations est augmentée, avec une probabilité p_{xd}, selon une règle de Widrow-Hoff de taux d'apprentissage β_{xd} :

$$R_x(s) \leftarrow (1 - \beta_{xd}) R_x(s) + \beta_{xd}$$

Les situations différentes de s ne sont pas augmentées systématiquement parce que, ce faisant, dans les tous premiers pas de temps d'une simulation, alors que l'agent n'a encore expérimenté qu'une infime partie des situations, cela peut engendrer des boucles comportementales faisant répéter à l'agent de manière cyclique les mêmes séquences d'actions. La mise à jour probabiliste des valeurs permet d'éviter cet écueil.

Comme dans les sections précédentes, MACS simule plusieurs actions à chaque pas de temps. Quand MACS simule une action avec la situation s_0 comme point de départ, il utilise le modèle des transitions fourni par la liste des classeurs et le mécanisme d'intégration. Avec ces informations, MACS calcule, pour chaque action possible a, l'ensemble $S_{s_0,a}$ des situations anticipées s_1. Une valeur de récompense attendue $V_x(s)$ est associée à chaque situation de l'ensemble P. Cette valeur indique dans quelle mesure la situation correspondante permettra, à l'avenir, d'éviter de percevoir les mêmes situations que celles qui ont été perçues récemment.

Étant donné une situation s_0 et une action a, nous définissons la qualité associée à la situation s_0 et à l'action a comme :

$$Q_x(s_0, a) = \max_{s_1 \in S_{s_0,a}} R_x(s_1) + \gamma V_x(s_1)$$

Dès lors, MACS met à jour la nouvelle valeur associée à s_0 :

$$V_x(s_0) = \max_{a \in A} Q_x(s_0, a)$$

Pendant la phase de sélection de l'action, quand la situation perçue est s_t, MACS choisit l'action propre à maximiser $Q_x(s_t, a)$. De cette manière, MACS construit une politique d'exploration systématique.

5.4 Compromis entre exploration et exploitation avec MACS

Dans les deux sections précédentes, nous avons décrit comment MACS construit trois politiques : deux pour l'exploration et la troisième pour l'exploitation. Nous abordons ici le problème de l'arbitrage entre ces trois politiques, de manière à produire un comportement global propre à combiner exploration active et exploitation.

Les trois politiques sont indépendantes, et l'arbitrage se fait durant la phase de sélection de l'action. Lorsque MACS perçoit une situation s_t, il calcule les qualités $Q_i(s_0, a)$, $Q_p(s_0, a)$ et $Q_x(s_0, a)$ respectivement associées à la maximisation de l'information, à la poursuite des buts et à l'exploration systématique, avant de les agréger. En ce qui concerne le compromis exploration/exploitation, il n'est pas souhaitable d'agréger ces trois critères en calculant une qualité globale comme somme pondérée des trois qualités élémentaires. En effet,

- s'il y a beaucoup de gains immédiats d'information à attendre, le cumul de ces gains au long terme peut être très important et sans commune mesure avec la récompense environnementale attendue ;
- la quantité de récompense environnementale ne peut pas être connue avant que le processus d'apprentissage ait atteint toutes les sources de récompense.

Il serait donc difficile de définir à l'avance des poids adéquats pour chacun des critères. Toutefois, il reste possible de définir une hiérarchie entre les critères. Puisque l'optimalité de la politique d'exploitation dépend de la fiabilité du modèle de l'environnement, la recherche d'informations améliorant ce modèle est prioritaire. Vient ensuite la maximisation de la récompense, puis l'exploration systématique. En conséquence, le choix de l'action opère de la sorte (nous notons $A_{\backslash\{a_0\}}$ l'ensemble A privé de l'élément a_0) :

- s'il existe une action a_0, telle qu'aucune autre action a'_0 n'est telle que $Q_i(s_t, a_0) < Q_i(s_t, a'_0)$, et s'il existe au moins une autre action a''_0 telle que $Q_i(s_t, a_0) > Q_i(s_t, a''_0)$, alors l'action choisie est a_0 ;
- sinon, s'il existe une action a_1, telle qu'aucune autre action a'_1 n'est telle que $Q_p(s_t, a_1) < Q_p(s_t, a'_1)$, et s'il existe au moins une autre action a''_1 telle que $Q_p(s_t, a_1) > Q_p(s_t, a''_1)$, alors l'action choisie est a_1 ;
- sinon, l'action choisie a_2 est telle que $Q_x(s_t, a_2) = \max_{a \in A} Q_x(s_t, a)$.

Autrement dit, si toutes les actions sont équivalentes concernant la maximisation de l'information, MACS examine l'optimalité des actions concernant la maximisation de la récompense environnementale. Si les différentes possibilités sont encore équivalentes selon ce deuxième critère, MACS cherche à explorer l'environnement de manière systématique.

5.5 Étude expérimentale

5.5.1 Exploration active

Dans cette section, nous détaillons des expériences pour valider expérimentalement l'exploration active de MACS en la confrontant à l'exploration aléatoire. Ces tests utilisent les mêmes environnements que dans la section 4.4.1. Ils présentent des moyennes sur 100 simulations. Les figures 5.3, 5.5, 5.7 et 5.9 montrent l'évolution moyenne du pourcentage de connaissance pour Maze228, Maze252, Maze288 et Maze324. Les figures 5.4, 5.6, 5.8 et 5.10 montrent l'évolution moyenne du nombre de classeurs pour les mêmes expériences.

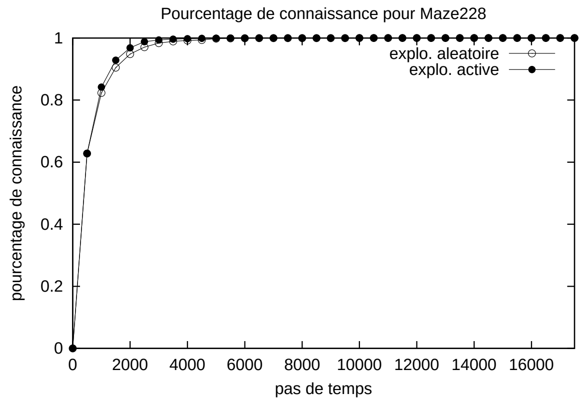

Fig. *5.3: MACS en exploration aléatoire et active : évolution du pourcentage de connaissance pour Maze228.*

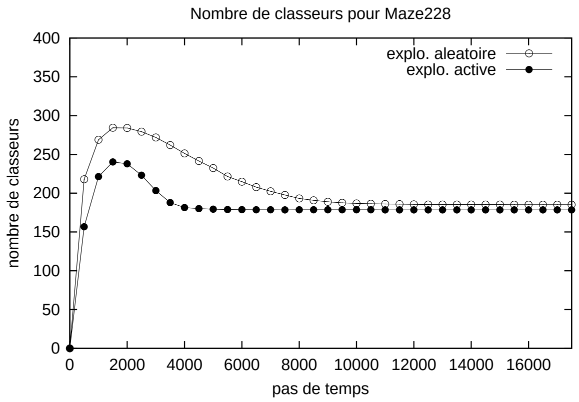

Fig. *5.4: MACS en exploration aléatoire et active : évolution du nombre de classeurs pour Maze228.*

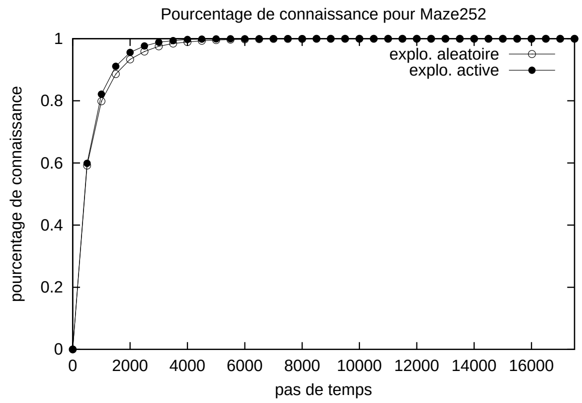

Fig. *5.5: MACS en exploration aléatoire et active : évolution du pourcentage de connaissance pour Maze252.*

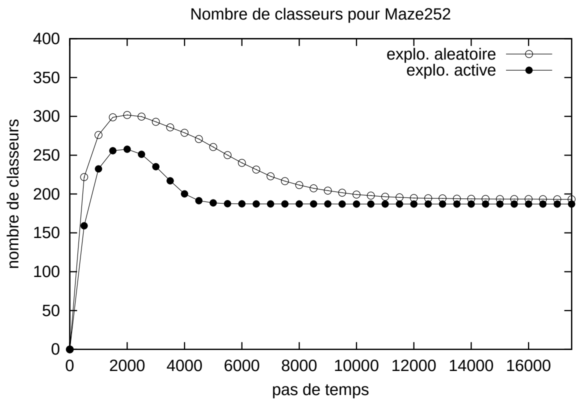

Fig. *5.6: MACS en exploration aléatoire et active : évolution du nombre de classeurs pour Maze252.*

Le tableau 5.2 résume les résultats expérimentaux obtenus au cours des différentes expériences. Ils présentent la moyenne et l'écart type associé, sur les 100 expériences, du temps mis par MACS en exploration aléatoire et en exploration active, pour atteindre une connaissance de 0.99 sur l'environnement. Ils montrent aussi la moyenne du nombre de classeurs nécessaires pour modéliser les interactions entre le système et son environnement.

Pour chaque environnement, des tests de Wilcoxon unilatéraux ont servi à tester statistiquement l'hypothèse de l'égalité des temps de convergence de MACS, avec ou sans exploration active, contre l'hypothèse selon laquelle les temps de convergence sont inférieurs avec l'explo-

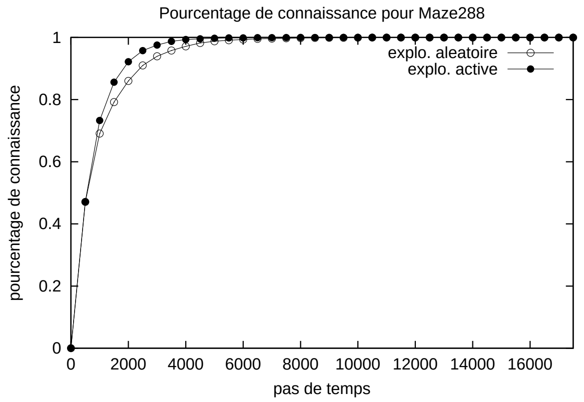

FIG. 5.7: *MACS en exploration aléatoire et active : évolution du pourcentage de connaissance pour Maze288.*

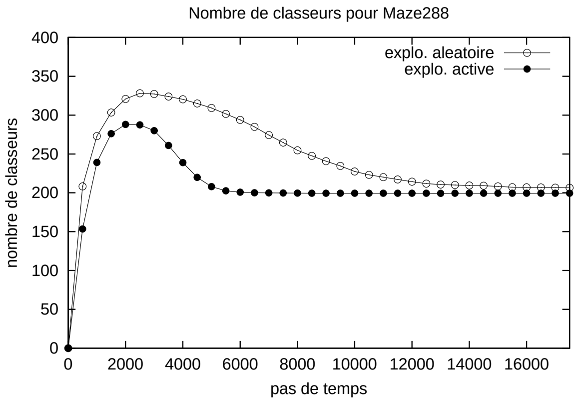

FIG. 5.8: *MACS en exploration aléatoire et active : évolution du nombre de classeurs pour Maze288.*

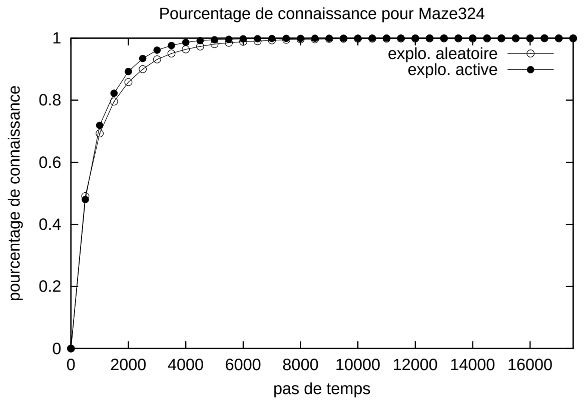

FIG. 5.9: *MACS en exploration aléatoire et active : évolution du pourcentage de connaissance pour Maze324.*

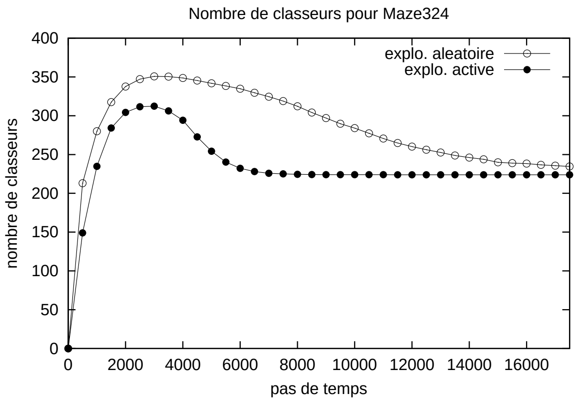

FIG. 5.10: *MACS en exploration aléatoire et active : évolution du nombre de classeurs pour Maze324.*

ration active. Les p-valeurs associées à chaque couple d'échantillons sont toutes inférieures à 10^{-5}. Les tests de Wilcoxon acceptent donc, à un seuil de 10^{-5}, l'hypothèse selon laquelle, pour chaque environnement testé, l'exploration active accélère l'apprentissage latent du modèle de l'environnement.

Les figures 5.11 et 5.12 reprennent ces résultats et présentent les relations entre la taille de l'environnement d'une part, et le nombre de classeurs et le temps de convergence vers un modèle complet et fiable d'autre part.

5.5.2 Prise en compte de problèmes plus complexes

L'exploration aléatoire est particulièrement inefficace quand les environnements sont de taille importante. Les bénéfices attendus de l'exploration active sont donc très importants dans ce cas.

Nous reproduisons ici une expérience menée dans un plus grand environnement, dont la

Expérience	Nombre de classeurs (moyenne)	Temps de convergence	
		Moyenne	Écart type
Maze228 - MACS aléatoire	185.2	2990	786
Maze228 - MACS explo. active	178.5	2524	560
Maze252 - MACS aléatoire	193.3	3680	987
Maze252 - MACS explo. active	187.2	3015	534
Maze288 - MACS aléatoire	206.4	4944	1403
Maze288 - MACS explo. active	199.4	3717	691
Maze324 - MACS aléatoire	231.1	5508	1315
Maze324 - MACS explo. active	223.9	3950	771

TAB. *5.2: Synthèse des résultats expérimentaux concernant la comparaison entre MACS avec exploration active et MACS sans exploration active*

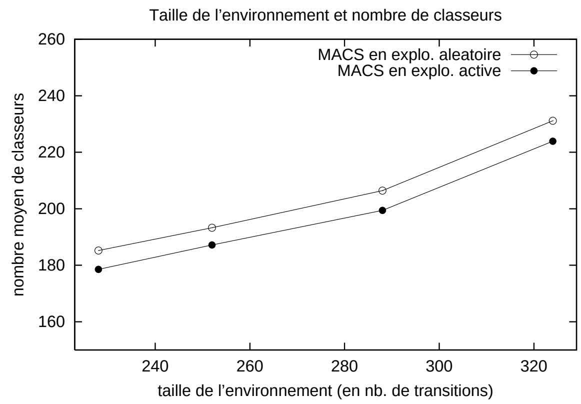

FIG. *5.11: Nombre de classeurs obtenus en fonction de la taille de l'environnement*

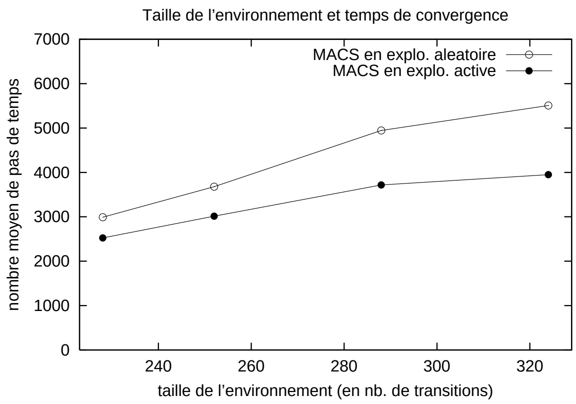

FIG. *5.12: Temps de convergence en fonction de la taille de l'environnement*

topologie est illustrée par la figure 5.13. Pour rendre cet environnement markovien, nous laissons à MACS l'opportunité de percevoir la présence ou l'absence de murs jusqu'à deux cases de distance. Ainsi, dans cette expérience, MACS doit gérer 25 attributs (voir figure 5.14) et modéliser 2232 transitions.

La figure 5.15 montre l'évolution du nombre de classeurs dans cette expérience, et la figure 5.16 montre l'évolution du pourcentage de connaissance. MACS est capable d'apprendre un modèle complet de l'environnement dans ce problème plus complexe que les problèmes précédents, tant en ce qui concerne sa taille, qu'en ce qui concerne le nombre des attributs.

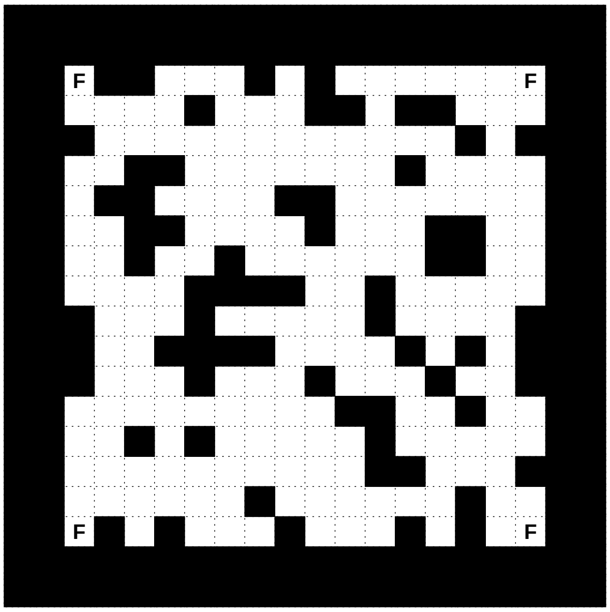

FIG. *5.13: Topologie de l'environnement Maze2232*

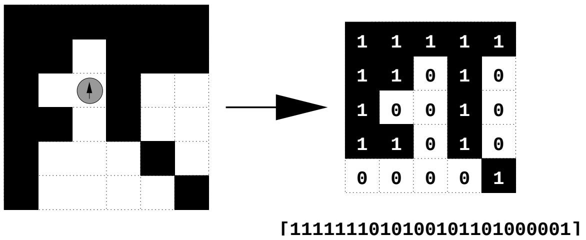

FIG. *5.14: Situation perçue dans Maze2232.* L'agent perçoit les 25 attributs correspondant aux cases autour de lui. Les attributs sont considérés de gauche à droite, et du haut vers le bas.

5.5.3 Maximisation de la récompense et exploration systématique

Les environnements Maze216C, Maze228C et Maze312C

Pour illustrer l'intérêt de l'apprentissage latent dans un processus d'apprentissage par renforcement, nous utilisons des environnements changeants. Nous proposons les environnements Maze216C, Maze228C et Maze312C respectivement illustrés par les figures 5.17, 5.18 et 5.19. Dans chacun de ces environnements, l'agent perçoit `0` pour les cases vides comme pour celle avec de nourriture. Il perçoit toujours `1` pour une case occupée par un obstacle. Au début de l'expérience, la nourriture se trouve dans la case marquée d'un `F` cerclé. Après que l'agent l'ait trouvée 20 fois, elle disparaît de cet endroit pour apparaître dans la case marquée d'un `F` non cerclé. Tous les 20 essais, la nourriture change ainsi de place.

Ces expériences sont complémentaires de celles menées par Sutton et Barto (1998) pour illus-

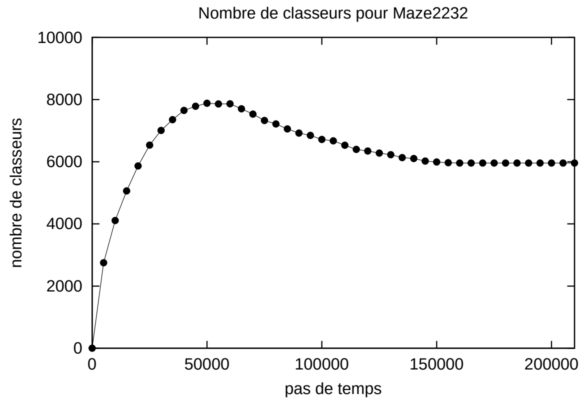

FIG. *5.15: Évolution du nombre de classeurs dans Maze2232*

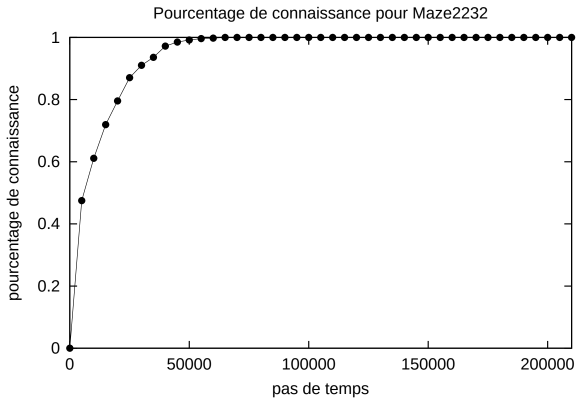

FIG. *5.16: Évolution du pourcentage de connaissance dans Maze2232*

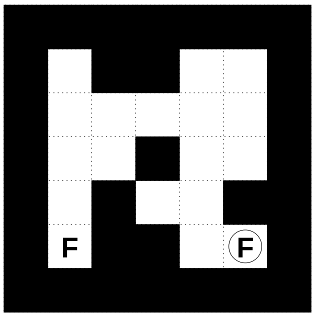

FIG. *5.17: Maze216C*

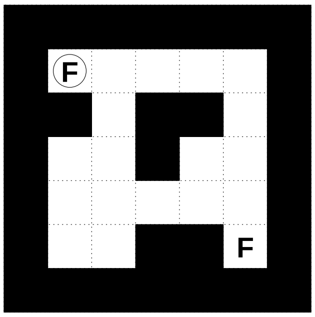

FIG. *5.18: Maze228C*

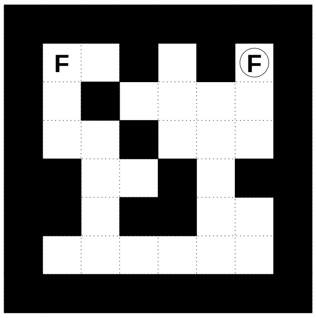

FIG. *5.19: Maze312C*

trer l'intérêt de l'architecture *Dyna* et de l'apprentissage latent. Les expériences de Sutton et Barto montrent comment une architecture *Dyna* permet de découvrir des chemins alternatifs lorsqu'un chemin est bloqué. Ici, nous montrons comment l'utilisation d'un modèle de l'environnement permet une réadaptation rapide, lorsque la source de récompense change.

Dans de tels environnements, la réadaptation d'un système qui n'est pas doté de la capacité à anticiper, comme *Q-learning* ou XCS, est très laborieuse puisque tout est à réapprendre chaque fois que la récompense change de place. Lorsqu'un tel agent a appris la politique correspondant à la partie droite de la figure 5.20, par exemple, il peut mettre longtemps à trouver la nouvelle source de récompense. En attendant, les récompenses immédiates sont toutes nulles. Le facteur d'amortissement γ des équations de Bellman conduit donc toutes les qualités à diminuer en tendant vers 0. Ne pouvant mettre à jour qu'une seule qualité à chaque pas de temps, un grand nombre d'actions successives seront nécessaires à *Q-learning* pour que les qualités s'annulent, comme dans la partie centrale de la figure. Avant cela, les qualités conduisent à aller en sens contraire à celui qui permettrait de découvrir la nouvelle source de récompense. Sa découverte ne peut être que fortuite, grâce à une politique stochastique comme celle présentée dans la section 1.3.1. Pour s'adapter à un changement de la source de récompense, un système comme

Q-learning devrait d'abord « oublier » une politique avant d'en apprendre une nouvelle, ce nouvel apprentissage étant lui même aussi long que celui de la première politique. La réadaptation serait donc plus lente que l'adaptation initiale.

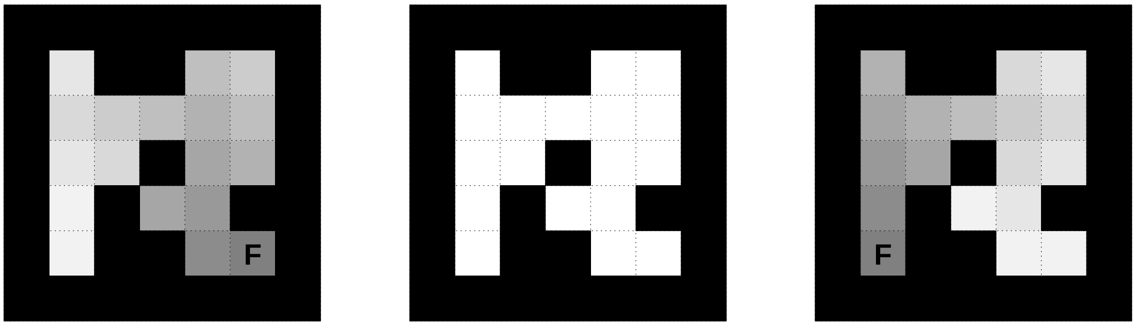

Fig. *5.20:* Lorsque la source de récompense est à droite, l'agent doit suivre la politique qui lui permet d'aller des cases claires vers les cases sombres. Lorsque la source de nourriture disparaît, il n'y a plus d'informations sur la récompense immédiate et la politique devrait être aléatoire. Lorsque la source de récompense est identifiée à gauche, l'agent doit construire une nouvelle politique.

En revanche, un agent capable d'anticiper peut mettre à jour sa politique plus rapidement. Dès que la source de nourriture disparaît, l'absence de récompense est rétro-propagée, et la politique est mise à jour conformément à la partie centrale de la figure 5.20, c'est-à-dire que toutes les valeurs s'annulent. Ceci lui permet de trouver plus rapidement la nouvelle source de nourriture. Il peut alors calculer très rapidement la nouvelle politique grâce à son modèle de l'environnement.

Résultats expérimentaux

Les figures 5.21, 5.22 et 5.23 illustrent respectivement le temps mis par MACS pour achever les essais successifs dans Maze216C, Maze228C et Maze312C. Ces résultats sont des moyennes sur 100 expériences. À chaque fois, MACS utilise tous les niveaux de son système de décision hiérarchique.

Dans chacune de ces expériences, les mêmes paramètres que précédemment sont utilisés, c'est-à-dire que les taux d'apprentissage des estimations concernant l'apprentissage latent sont de 0.1. Le seuil de nombre d'évaluations θ_e est égal à 5 (voir la section 4). Les facteurs d'amortissement γ des politiques sont de 0.5. La probabilité d'augmenter l'attrait d'une situation tant qu'elle n'est pas perçue est 0.1. Cet attrait augmente selon un taux d'apprentissage de 0.01 et se trouve réduit à 0 dès que la situation correspondante est perçue.

Ces résultats montrent la rapide réadaptation de MACS à chaque changement de la position de la nourriture. Tous les 20 essais, un essai plus long est le résultat d'un changement de la position de la source de renforcement. Quand cela arrive, toutes les récompenses immédiates associées aux situations deviennent égales à 0 et les récompenses attendues, grâce à l'utilisation de l'anticipation, tombent elles aussi à 0.

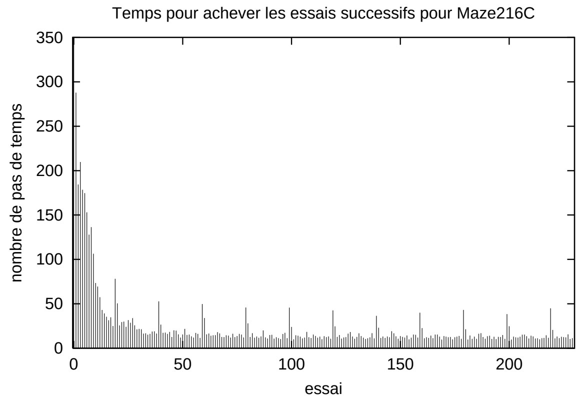

FIG. *5.21: MACS en exploration et exploitation simultanée : nombre de pas de temps nécessaires pour atteindre le but au cours des essais successifs dans Maze216C.*

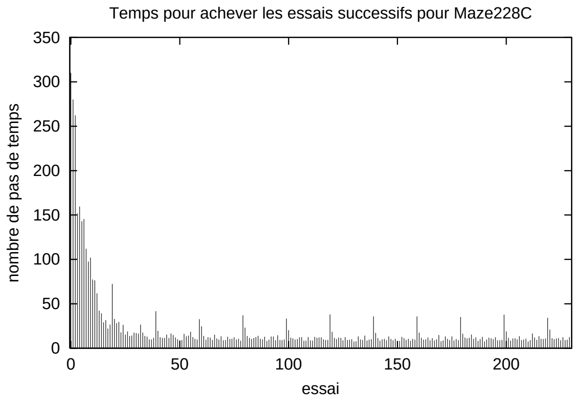

FIG. *5.22: MACS en exploration et exploitation simultanée : nombre de pas de temps nécessaires pour atteindre le but au cours des essais successifs dans Maze228C.*

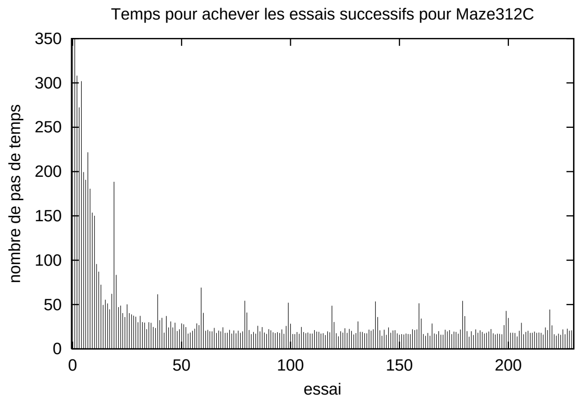

FIG. *5.23: MACS en exploration et exploitation simultanée : nombre de pas de temps nécessaires pour atteindre le but au cours des essais successifs dans Maze312C.*

Dès lors, la politique est réinitialisée, et MACS entreprend d'explorer son environnement de manière systématique. Sans cette exploration systématique, le nombre de pas de temps nécessaires pour retrouver la nouvelle source de récompense est nettement plus long.

Dès que MACS identifie la nouvelle source de récompense, quelques pas de temps sont nécessaires pour mettre à jour la politique grâce aux capacités d'anticipation. C'est pourquoi les essais suivant immédiatement la découverte de la source de nourriture sont légèrement plus longs que leurs suivants. Dès que la politique est à jour, le comportement de l'agent est à nouveau optimal. Les légères variations tiennent au fait que chaque essai débute dans une case aléatoire, et que les résultats présentés sont moyennés sur 100 expériences seulement.

Les figures 5.24, 5.26 et 5.28 présentent l'évolution moyenne, sur 100 expériences, du pourcentage de connaissance dans Maze216C, Maze228C et Maze312C. Les figures 5.25, 5.27 et 5.29

présentent l'évolution moyenne du nombre de classeurs pour les mêmes expériences.

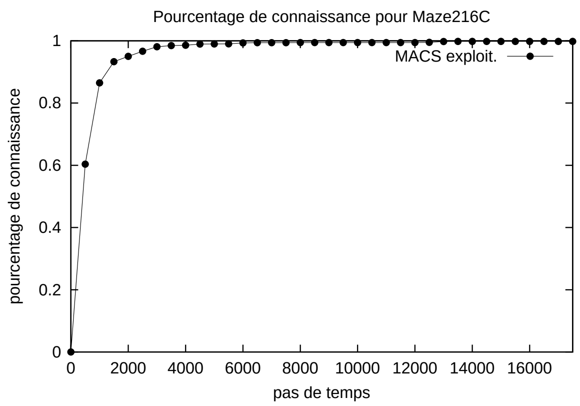

FIG. *5.24: MACS en exploration et exploitation simultanée : évolution du pourcentage de connaissance pour Maze216C.*

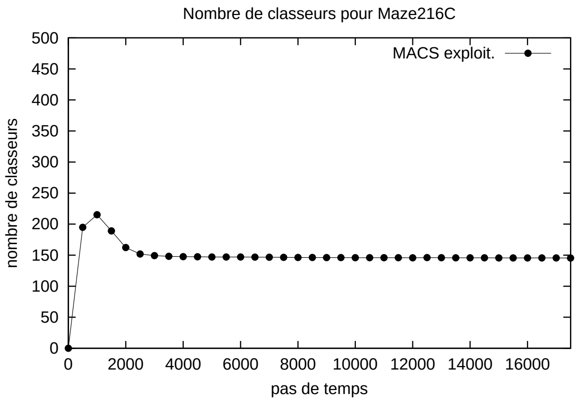

FIG. *5.25: MACS en exploration et exploitation simultanée : évolution du nombre de classeurs pour Maze216C.*

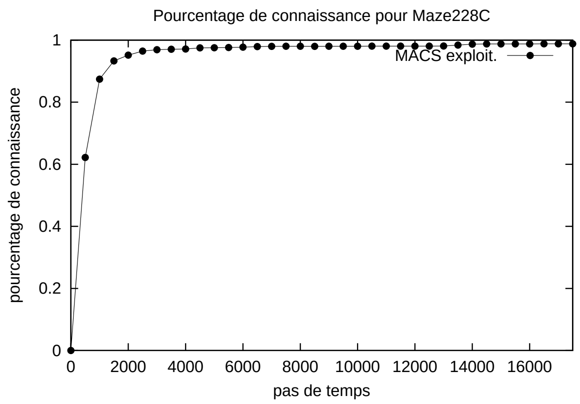

FIG. *5.26: MACS en exploration et exploitation simultanée : évolution du pourcentage de connaissance pour Maze228C.*

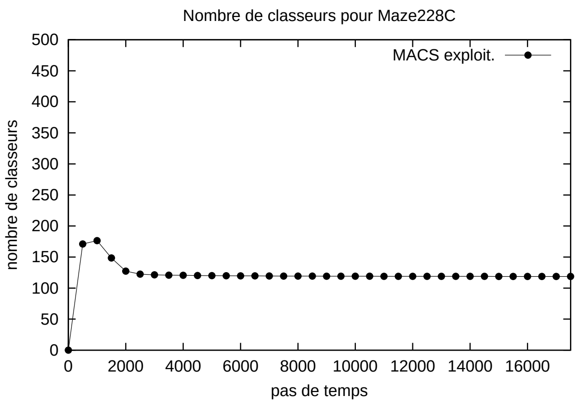

FIG. *5.27: MACS en exploration et exploitation simultanée : évolution du nombre de classeurs pour Maze228C.*

5.6 Discussion

Dans ce chapitre, nous avons intégré l'apprentissage latent de MACS à une architecture classique de type *Dyna*. Comme dans d'autres architectures *Dyna*, un module apprend un modèle de l'environnement et un autre module apprend les valeurs de renforcement. En utilisant plusieurs récompenses immédiates, nous avons pu définir trois politiques différentes, en utilisant à chaque fois une variante de *Value Iteration*.

Comme dans *DynaQ+*, une de ces politiques utilise la récompense de l'environnement. Comme

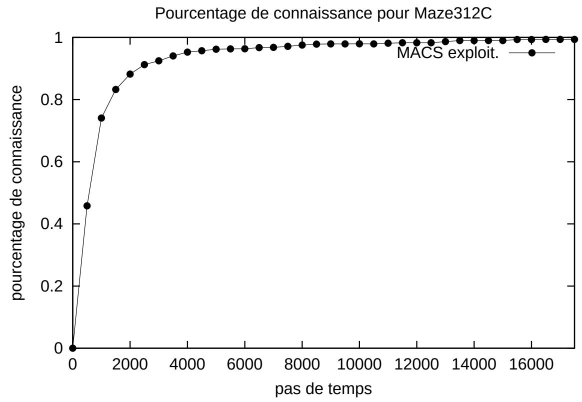

FIG. *5.28: MACS en exploration et exploitation simultanée : évolution du pourcentage de connaissance pour Maze312C.*

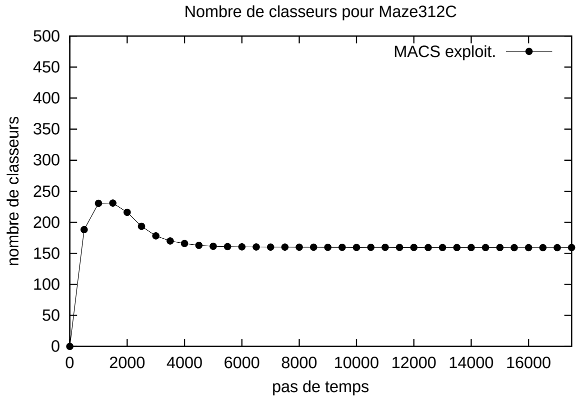

FIG. *5.29: MACS en exploration et exploitation simultanée : évolution du nombre de classeurs pour Maze312C.*

le montre la rapide réadaptation de MACS lorsque la source de récompense change de place, l'utilisation du modèle de l'environnement permet d'accélérer l'apprentissage de cette politique, qui permet à l'agent de remplir les objectifs qui lui ont été assignés. Une autre utilise une récompense interne associée à chaque situation, et qui dépend du temps écoulé depuis la dernière fois que la situation a été perçue. Cette politique permet de visiter fréquemment tous les états de l'environnement. Chacune de ces deux politiques a un équivalent dans *DynaQ+*.

La troisième politique utilisée dans MACS, celle d'exploration active, est plus originale. Pour favoriser l'exploration, *DynaQ+* utilise un mécanisme semblable à la politique d'exploration systématique que nous avons définie. Ainsi, l'agent cherche à visiter le plus souvent possible chaque état de l'environnement. Dans MACS, les niveaux d'évaluation l (voir section 5.1.1) sont utilisés pour déterminer les parties du modèle qui n'ont pas été suffisamment évaluées. La récompense interne qui en dépend incite l'agent à évaluer les classeurs qui ne l'ont pas encore été suffisamment. La politique d'exploration qui en résulte est très différente d'une politique d'exploration systématique dans la mesure où elle prend en compte les insuffisances du modèle pour chercher explicitement à les combler. Cette exploration active permet de choisir les actions de manière déterministe (voir section 5.1.2), et d'explorer l'environnement efficacement. L'agrégation hiérarchique permet d'utiliser aussi une politique déterministe pour l'exploitation, et d'obtenir un comportement optimal, pour peu que le modèle de l'environnement soit fiable.

La méthode d'agrégation des différentes politiques dans MACS est elle aussi originale par rapport à celle employée dans *DynaQ+*. Dans ce dernier système, les qualités et les valeurs pour chaque politique sont sommées pour obtenir une qualité unique proposant un compromis entre les deux critères. C'est de cette qualité que dépend la politique. Pourtant, la qualité correspondant à l'exploration n'est jamais nulle, et la politique est toujours influencée par l'exploration. L'agent n'agit donc jamais de manière à maximiser le cumul de ses récompenses. Dans MACS, par contre, si le modèle de l'environnement est fiable et si tous les classeurs ont été suffisamment évalués, il n'y a plus d'information à gagner, et la politique est optimale par rapport à la récompense de l'environnement.

En outre, l'ordre de grandeur des récompenses externes n'est pas connu à l'avance et, à mesure que l'environnement est plus grand, les valeurs pour l'exploration sont plus importantes, puisque l'agent ne peut pas visiter tous les états aussi fréquemment. Pour MACS, nous avons préféré une méthode d'agrégation hiérarchique. L'ordre de grandeur des valeurs pour chaque politique est alors indifférent et, lorsque le modèle est complet et fiable, la politique d'exploitation prend tout à fait le pas sur la politique d'exploration, si bien que le comportement est aussi optimal que le permet la qualité du modèle.

Nous avons donc utilisé une architecture *Dyna* pour utiliser les résultats de l'apprentissage latent afin d'accélérer l'apprentissage de la politique. Dans ce type de systèmes, la qualité de la politique est très dépendante de la fiabilité du modèle de l'environnement. Si toutes les récompenses immédiates sont rétro-propagées selon un modèle inadéquat, la politique résultante sera sous-optimale. Pourtant, dans les mécanismes d'apprentissage latent présentés au chapitre 4, nous avons fait l'hypothèse que l'environnement était déterministe et markovien. Pour les mêmes raisons que YACS (voir section 3.4.1), MACS n'est pas capable de prendre en compte les problèmes stochastiques ou non-markoviens. Dans ces deux cas, MACS est incapable de construire un modèle environnemental fiable. Et dans des problèmes réels comme ceux auxquels peut faire face un robot, les caractères déterministes et markoviens sont rarement vérifiés. Dans le chapitre suivant, nous discutons de la prise en compte d'environnements non-markoviens ou stochastiques.

Chapitre 6

Discussion

Dans ce mémoire, les environnements utilisés pour tester successivement YACS et MACS sont d'une taille comparable à celle des environnements généralement utilisés dans le cadre des systèmes de classeurs. Pourtant, les gains en vitesse d'apprentissage de YACS par rapport à ACS, puis ceux de MACS par rapport à YACS, et encore ceux apportés à MACS par l'exploration active, sont tels que l'on peut espérer traiter des problèmes plus complexes. Avec Maze2232 (voir section 5.5.2), nous proposons un problème d'une taille plus grande que ceux habituellement traités par les systèmes de classeurs. Des expérimentations complémentaires seraient nécessaires pour étudier précisément le comportement de MACS dans des environnements complexes. Néanmoins, nous avons montré que, pour un problème de la taille de Maze2332, MACS peut apprendre un modèle complet.

Remarquons également que, dans la section 3.3.1, nous avons présenté les environnements de type « *Wilson Woods* » comme représentant des automates à états finis particuliers. Puisqu'aucune spécificité de ce type d'automates n'est exploitée par les mécanismes de MACS, les résultats obtenus doivent être généralisables à tous types d'automates, comme ceux faisant intervenir des clowns et des pâtisseries, par exemple.

Quoi qu'il en soit, indépendamment du fait que YACS et MACS offrent des performances accrues par rapport à ACS, ce qui nous a particulièrement intéressé dans le cadre de cette thèse, ce sont les avancées conceptuelles que ces systèmes apportent dans le cadre général des systèmes de classeurs. YACS utilise le même formalisme qu'ACS, mais simplifie ses mécanismes et pose différemment le problème de la fiabilité des classeurs à anticipation (voir section 3.4.2). L'originalité de MACS, discutée dans la section 4.5, réside dans ses anticipations partielles et dans l'aspect modulaire qui en résulte. MACS pose ainsi autrement le problème de l'anticipation dans les systèmes de classeurs.

Dans ce chapitre, nous examinons les principales limitations de l'approche de MACS, qui concernent la prise en compte de l'incertain, qu'il soit dû à un caractère stochastique ou à des problèmes d'ambiguïté. Nous examinerons au chapitre suivant les avantages et les perspectives offertes avec une approche telle que MACS dans le cadre plus général de l'approximation de fonctions.

6.1 MACS et les problèmes markoviens stochastiques

Un environnement est stochastique si, compte tenu de l'état de l'agent dans son environnement et de la situation perçue associée, entreprendre une action particulière peut mener indifféremment à des situations perçues diverses. Il y a deux facteurs d'incertitude :

- le bruit sur les actions a pour conséquence que l'action réellement effectuée n'est pas la même que l'action décidée. C'est, par exemple, le cas si les effecteurs sont déficients. Un agent décidant d'avancer dans une direction donnée peut ainsi se diriger parfois dans une autre direction. Dans ce cas, les actions décidées n'ont pas toujours les mêmes conséquences et les situations perçues, après avoir tenté une action particulière dans une certaine situation, ne sont pas toujours les mêmes. En résulte une nécessaire prise de décision dans l'incertain ;
- le bruit sur les perceptions a pour conséquence de modifier parfois les situations telles qu'elles seraient normalement perçues. Dès lors, alors que l'agent devrait percevoir une certaine situation, il en perçoit parfois d'autres sans que ce soit prévisible. Ici encore, les conséquences perceptibles des actions entreprises sont incertaines.

Dans de tels environnements, tous les classeurs anticiperaient parfois bien, et parfois mal, l'issue des actions étant toujours incertaine. En conséquence, le mécanisme de spécialisation opérerait à chaque fois qu'un classeur serait suffisamment évalué. MACS produirait donc un ensemble de classeurs avec des parties `Condition` complètement spécialisées et en outre, aucun de ces classeurs ne serait fiable.

Les estimations utilisées par MACS offrent toutefois une certaine robustesse au bruit puisque leur mise à jour utilise des règles de Widrow-Hoff. Ce sont les heuristiques qui sont en cause. Pour traiter des problèmes stochastiques, il est donc possible de renoncer aux heuristiques utilisées dans MACS, mais de conserver tout de même les estimations présentées dans les sections 4.3.3 et 4.3.4. Ces estimations donnent une indication sur la manière de conduire la construction du modèle. Elles pourraient être utilisées comme biais dans des algorithmes génétiques, par exemple, comme cela a été fait par Dorigo (1994) avec des estimations moins précises, qui permettaient d'indiquer un classeur à spécialiser, mais pas l'attribut à spécialiser[36]. Plutôt que d'utiliser des mutations aléatoires, on pourrait par exemple utiliser les estimations pour introduire des mutations dont on a de bonnes raisons de penser qu'elle sont souhaitables.

Quoi qu'il en soit, les difficultés concernant les aspects stochastiques ne sont pas spécifiques à l'approche particulière de MACS pour l'anticipation. Les mêmes problèmes ont déjà été évoqués pour YACS à la section 3.4.1.0. Les questions soulevées dans la section suivante, par contre, sont spécifiques aux anticipations partielles de MACS.

[36]Des biais sont d'ailleurs déjà utilisés dans des systèmes de classeurs comme XCS, avec les mécanismes de couverture et de subsomption.

6.2 MACS et les problèmes non-markoviens

6.2.1 Les ambiguïtés sur les perceptions

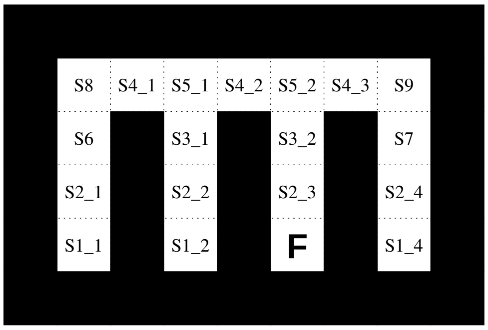

FIG. *6.1: L'environnement Maze10*

Dans le cas d'un agent situé doté d'une perception locale de son environnement, le non-déterminisme n'est pas le seul facteur d'indétermination. En effet, il se peut que, dans différents états, les situations perçues par l'agent soient différentes, indépendamment d'un possible bruit sur les perceptions. La figure 6.1 représente un environnement posant ce type de problèmes. Cet environnement a été proposé par Lanzi (1998b), et il a également été étudié par Landau et al. (2002). Dans ces environnements, attendu que l'agent ne perçoit que les huit cases autour de lui, il perçoit la même situation dans chaque groupe d'états dont le premier indice est le même. C'est par exemple le cas des états marqués S4_1, S4_2 et S4_3. Entreprendre certaines actions dans chacun de ces états peut mener à des états et des perceptions différents. Le problème vient du fait que l'agent manque d'information pour distinguer ces différents états. D'une manière ou d'une autre, il doit avoir recours à des informations passées pour lever l'ambiguïté sur ces états.

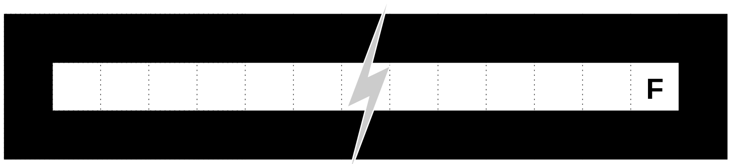

FIG. *6.2: Ambiguïtés vis-à-vis du modèle et de la fonction de récompense.*

Si les ambiguïtés peuvent être problématiques pour l'apprentissage direct d'une politique, les méthodes d'apprentissage par renforcement fondées sur la construction d'un modèle peuvent s'y révéler encore plus sensibles. En effet, avec une méthode directe, seule compte l'optimalité de l'action. Tant que les qualités des couples situation-action faisant intervenir des situations ambiguës restent adéquates, relativement aux valeurs des autres situations, l'impossibilité d'anticiper la situation future ne pose pas de problème.

L'environnement de la figure 6.2 illustre ce point. Dans cet environnement, l'agent est situé dans une case vide et perçoit les huit cases qui l'entourent, dans le sens des aiguilles d'un montre, en partant de la case au-dessus de lui. Ses actions correspondent à des mouvements d'une case vers une case adjacente. Il reçoit une récompense lorsqu'il rejoint la case marquée F.

Cet environnement comporte une ambiguïté dans la mesure où, si l'agent perçoit la situation commune à toutes les cases du centre et s'il entreprend une action vers la droite, il lui est impossible, sans autre information que sa situation, de déterminer sa situation future. En effet, selon la case dans laquelle il se trouve, il pourra atteindre une case similaire ou bien la case du coin droit, marquée F. Construire un modèle fiable d'un tel environnement peut donc se révéler difficile. Pourtant, si l'agent ne cherche pas à construire un modèle, mais cherche directement une politique, il pourra déterminer que l'action optimale est toujours la même : un mouvement vers la droite. Ainsi, dans cet exemple, un certain nombre de cases sont ambiguës (toutes celles du centre), mais un système d'apprentissage par renforcement direct, comme le *Q-learning*, pourra trouver une politique optimale, même s'il est impossible de construire un modèle de l'environnement sans avoir recours à une mémoire du passé.

Dans la suite de ce chapitre, nous passons en revue un certain nombre de techniques propres à prendre en compte les ambiguïtés dans les situations. Ces exemples abordent toujours des problèmes où, non seulement la situation future est difficile à anticiper, mais où, en outre, l'action optimale est différente selon les états caractérisés par les mêmes situations. C'est par exemple le cas de Maze10 (voir figure 6.1). Ici, dans les états ambigus S2_1, S2_2, S2_3 et S2_1, non seulement il est difficile d'anticiper sa situation future après un mouvement vers le haut mais, en plus, l'action optimale n'est pas toujours la même. En S2_1, S2_2 et S2_1, il convient de se diriger vers le haut, alors qu'en S2_3, il faut se diriger vers le bas.

6.2.2 Les POMDP

Au lieu de décrire un environnement dans lequel certains états sont ambigus comme un processus de décision markovien (voir section 1.2.2), on a coutume de recourir au formalisme des processus de décision markoviens partiellement observables (POMDP[37]). Un tel processus est caractérisé par un processus de décision markovien classique comprenant les éléments $< S, A, T, R >$ décrits dans la section 1.2.2, auquel on ajoute un ensemble discret de symboles de Ω et une fonction d'observation $O\ : S \rightarrow \Pi(\Omega)$.

Dans le cadre informatique, S est souvent un vecteur d'observations et Ω une projection de S, représentative de ce que l'agent perçoit. La difficulté de l'observation partielle provient de ce que l'agent peut avoir des perceptions identiques pour le même symbole Ω dans deux états différents de son environnement.

Le problème de la prise de décision dans un POMDP (Kaelbing *et al.*, 1996) consiste à supposer que l'agent ne sait pas dans quel état il est - mais connaît un modèle du processus markovien sous-jacent à ses observations - puis à introduire l'état dans lequel l'agent croit être

[37] Partially Observable Decision Process (Kaelbing *et al.*, 1996)

(*belief state*). Cet état représente l'état interne de l'agent. On sait alors, en théorie, trouver l'alternative optimale, parmi toutes les séquences d'actions possibles, en utilisant des algorithmes de programmation dynamique sur ce POMDP.

En pratique, pour contourner un problème d'explosion combinatoire, des heuristiques spécifiques, comme l'algorithme *Witness* (Littman *et al.*, 1995), évitent d'engendrer les séquences qui ont peu de chance d'être optimales. Plus récemment, Hansen (1998) propose un algorithme d'itération sur les lois de commande (*Policy Iteration*) qui construit directement des automates représentant la commande optimale. Ces heuristiques présentent l'inconvénient de présupposer la connaissance du MDP[38] sous-jacent. Cette hypothèse suppose une modélisation préalable de l'environnement de l'agent et rend le cadre classique de résolution des POMDP mal adapté aux types de problèmes qui nous intéressent ici : ceux pour lesquels le concepteur n'a pas de modèle de l'environnement.

6.2.3 L'apprentissage dans les problèmes partiellement observables

Dans les sections suivantes, nous examinons différentes techniques utilisées dans l'apprentissage par renforcement, pour adjoindre aux situations ambiguës des informations concernant le passé de l'agent. Nous distinguons quatre types de solutions à la levée des ambiguïtés :

- les perceptions de l'agent ne sont plus constituées de ses seules perceptions immédiates, mais aussi d'informations concernant les situations et actions passées. Par exemple, cela a été fait par McCallum (1996) ;
- les règles comportementales sont liées les unes aux autres pour former des séquences qui évitent de fonder la décision sur les situations ambiguës. C'est ce qui a été fait par Tomlinson et Bull (2000) . Le mécanisme d'ACS pour aborder les problèmes non-markoviens utilise aussi un mécanisme de chaînage d'actions.
- des états internes explicites sont ajoutés aux perceptions. Ils deviennent de nouveaux attributs, comme c'est le cas dans les travaux de Lanzi (1998a) ;
- l'environnement partiellement observable est découpé à l'aide de modules en plusieurs environnements totalement observables. Cette solution a été utilisée par Wiering et Schmidhuber (1997), puis par Sun et Peterson (2000).

Dans la suite, nous montrons comment les deux premières solutions appartiennent à la même famille et nous mettons leurs limites en évidence. Nous montrons ensuite comment l'ajout d'états internes permet de résoudre les problèmes d'ambiguïté, puis dans quelle mesure un partitionnement du problème permet d'améliorer encore cette solution.

6.2.4 Conserver une mémoire explicite du passé

Un moyen de lever les ambiguïtés sur les situations est d'y adjoindre une information explicite sur les dernières actions et situations perçues. En effet, si une situation ne permet pas seule de

[38] *Markov Decision Process*, voir section 1.2.2

distinguer un état d'un autre, lui adjoindre des informations concernant le passé peut suffire à rendre cette distinction possible.

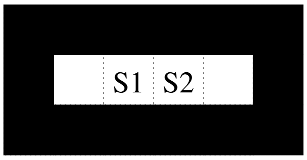

FIG. *6.3: Environnement non-markovien de niveau 1.*

Dans l'environnement présenté dans la figure 6.3, si l'agent perçoit les huit cases adjacentes, et en considérant que son orientation est fixe, alors il perçoit la même situation dans l'état S1 et dans l'état S2. Pourtant, si sa dernière action était un déplacement vers l'est, alors dans le premier cas sa perception correspondait au coin gauche alors que, dans le second, la situation perçue était celle d'une des cases au centre. Cette information explicite concernant le passé permet à l'agent de distinguer situations ambiguës.

McCallum a proposé l'algorithme *U-Tree* (McCallum, 1996) permettant de construire un arbre de décision intégrant de l'information sur les actions et situations passées de manière à distinguer les situations ambiguës. En effet, dès lors que la politique de l'agent est stable, alors le recours à ce type de mémoire permet de distinguer les situations les unes des autres et de choisir l'action adéquate. L'agent doit prendre en compte les séquences situation perçue-action à rebours, jusqu'à un point du passé où une telle séquence n'est pas ambiguë. L'agent doit donc adjoindre aux situations autant de pas de temps qu'il est nécessaire, pour considérer la dernière séquence non ambiguë, c'est-à-dire celle faisant intervenir une action vers l'est ou vers l'ouest.

Nous avons montré (Sigaud et Gérard, 1999) que ce type de méthode est mal adapté dans le cas où l'agent doit gérer un compromis entre l'exploration et l'exploitation. En effet, si la politique de l'agent est stable, alors le recours à ce type de mémoire permet de distinguer les situations les unes des autres et de choisir l'action adéquate. Dans les expériences que nous avons menées, nous avons constaté que, tant que la politique n'est pas stable, ce qui est le cas tout le long de l'apprentissage, l'agent peut avoir besoin de recourir à une mémoire arbitrairement longue. Dans cet exemple, si le processus d'exploration mène l'agent à répéter des actions nord ou sud, l'amenant à se cogner sans cesse dans un mur, alors le nombre de pas de temps à adjoindre à la situation ambiguë peut être arbitrairement long.

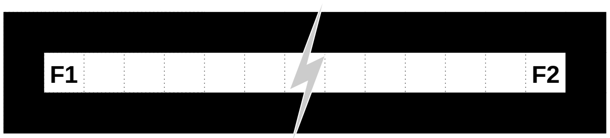

FIG. *6.4: Environnement non-markovien de niveau arbitraire.*

Un autre exemple illustre les limitations de ce type de techniques fondé sur une mémoire de toutes les expériences passées successives, jusqu'à une certaine durée. Considérons en effet l'environnement illustré par la figure 6.4. Ici, une seule source de nourriture (F1 ou F2) est présente à la fois. Dès que l'agent rencontre F1, celle-ci disparaît et F2 apparaît (et inversement) et le but de l'agent change. Dans ce cas, la mémoire requise est très limitée. En principe, lorsque l'agent se trouve dans une case ambiguë, il lui suffit de se rappeler si la dernière fois qu'il a atteint une des extrémités du couloir, il s'agissait de l'extrémité de gauche ou non. Ici, l'état interne propre à distinguer les perceptions ambiguës peut ne prendre que deux valeurs.

Or, en utilisant une mémoire des pas de temps précédents, si la longueur de couloir est n, dans le meilleur des cas, l'agent devra faire référence à sa situation à $n-2$ pas de temps dans le passé. Si n est grand, alors la chaîne de Markov nécessaire à la résolution de ce problème sera elle aussi très grande, alors même qu'un seul état interne prenant deux valeurs suffit à résoudre le problème, quel que soit n.

6.2.5 Créer des séquences de classeurs ou d'actions

Avec CXCS (Tomlinson et Bull, 2000), Tomlinson propose d'ajouter à XCS un mécanisme pour lier les classeurs les uns aux autres. Chaque classeur est lié à un classeur qui le précède et à un autre qui le suit. Ces liens forment des groupes de classeurs, des corporations qui sont traités comme des classeurs ordinaires. Les liens sont renforcés en fonction de la récompense environnementale et permettent d'activer des classeurs en séquence. Si, pendant une séquence, une situation ambiguë survient, la séquence continuera d'être exécutée en ignorant cette situation.

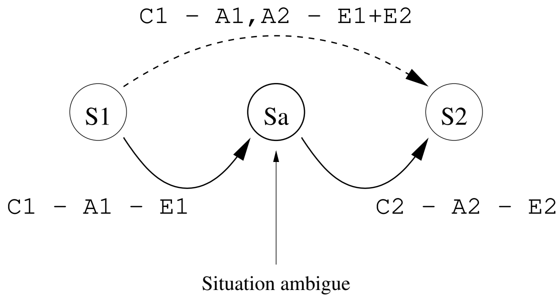

FIG. *6.5: ACS et les problèmes non-markoviens.*

ACS utilise un mécanisme similaire pour former des chaînes d'actions. Une situation ambiguë est identifiée lorsque tous les mécanismes de spécialisation d'ACS ont échoué à rendre un classeur fiable. Dans ce cas, la partie `Condition` d'un tel classeur est considérée comme ambiguë et le classeur doit être lié à un classeur qui le précéderait, de manière à passer outre la situation ambiguë. Un tel cas est illustré par la figure 6.5. Les situations $S1$ et $S2$ ne sont pas ambiguës mais la situation Sa l'est. Le classeur `C1 - A1 - E1` est fiable, mais à cause de l'ambiguïté, `C2 - A2 - E2` ne l'est pas. Dès qu'un succès dans l'évaluation du classeur problématique est précédé de l'activation du classeur fiable, un nouveau classeur est créé, qui synthétise les deux classeurs.

Dans ce classeur `C1 - A1,A2 - E1+E2`, la partie `Action` est une séquence composée des actions `A1` et `A2`, et la partie `Effet` est une synthèse des effets successifs `E1` et `E2` obtenue en appliquant la fonction *passthrough*[39] à E1 et E2.

Cette solution autorise le chaînage d'un nombre quelconque d'actions, de manière à prendre en compte des environnements avec beaucoup de situations ambiguës. Pourtant, le chaînage d'actions impose de créer des chaînes d'actions aussi longues que le nombre de situations ambiguës successives. Le problème subséquent est identique à celui de la solution consistant à conserver une mémoire explicite du passé. Les solutions présentées plus loin permettent de résoudre ce problème par l'introduction d'états internes au lieu d'une mémoire explicite.

6.2.6 Augmenter les perceptions avec des états internes

Une autre approche pour distinguer les situations ambiguës consiste à découvrir comment utiliser et comment positionner correctement des états internes qui seront utilisés, en plus des perceptions courantes, pour décider de l'action. Dans le cadre de systèmes de classeurs, c'est ce qui a été fait par Lanzi avec XCSM (Lanzi, 1998a).

L'utilisation de classeurs avec seulement une condition et une action ne permet pas d'agir de manière optimale dans un environnement partiellement observable. Il est nécessaire d'ajouter aux classeurs des informations sur le passé. Pour ce faire, XCSM gère un registre de mémoire interne composé d'un certain nombre de bits de mémoire qui, ensemble, représentent explicitement l'état interne de l'agent. La taille de ce registre est fixée *a priori*.

De manière à utiliser cette mémoire pour lever les ambiguïtés sur les situations, une condition interne est ajoutée à la condition externe concernant les situations. Pour que l'action d'un classeur soit sélectionnée, sa condition interne doit être appariée avec le registre de mémoire et sa condition externe doit être appariée avec la situation. L'action préconisée par le classeur ne sera choisie que si ces deux conditions sont remplies simultanément.

De manière à modifier cet état interne, on adjoint aux classeurs une action interne qui est exécutée en même temps que l'action externe. Cette action a pour effet de changer la valeur du registre de mémoire afin de modifier son état interne.

Dans XCSM, l'utilisation et la modification de ce registre de mémoire est dévolue à un algorithme génétique, comme c'est souvent le cas dans le cadre des systèmes de classeurs. Le problème de ce type d'approche est que, souvent, l'état interne est très instable. En effet, la seule contrainte sur la dynamique de l'état interne est qu'il soit bien positionné au moment où survient une situation ambiguë. Le reste du temps, sa valeur importe peu et il a tendance à beaucoup osciller. Si cela ne nuit pas à l'efficacité, cela rend par contre le résultat du processus d'apprentissage moins intelligible, ce qui est une qualité attendue d'un système à base de règles.

[39] voir section 2.4.4

6.2.7 Découper un problème partiellement observable

Un moyen de contraindre la stabilité de l'état interne, pour mieux en comprendre le fonctionnement *a posteriori*, est l'utilisation de systèmes modulaires consistant à découper un problème non-markovien en plusieurs problèmes markoviens. Cette approche a été utilisée par Wiering et Schmidhuber (1997) puis par Sun et Peterson (2000).

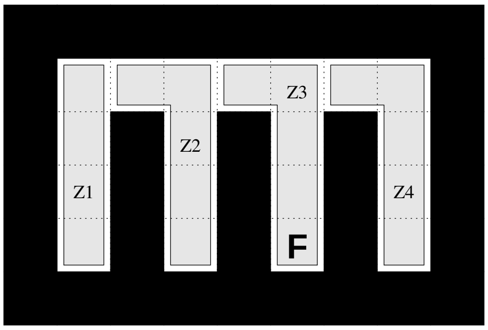

FIG. *6.6: Découpage de Maze10 en zones markoviennes.*

Dans l'exemple de Maze10, par exemple, on peut définir quatre zones de l'environnement telles qu'à l'intérieur de chacune d'elles, il n'y ait pas d'ambiguïté qui interdise de trouver une politique optimale sans recourir à une mémoire. Ces zones sont illustrées par la figure 6.6.

Wiering et Schmidhuber (1997) ainsi que Sun et Peterson (2000) proposent chacun un système modulaire tel que chaque module trouve une politique optimale à l'intérieur d'une zone. Le nombre initial de modules (et donc de zones, voir figure 6.6) est fixé au début de l'expérience par l'expérimentateur. Pour définir une politique optimale à l'intérieur d'une zone, chaque agent utilise l'algorithme *Q-learning*. Dans les travaux de Wiering et Schmidhuber (1997), comme dans ceux de Sun et Peterson (2000), un seul module est actif à chaque pas de temps. La politique suivie par le système est celle indiquée par l'agent actif. Le problème central pour le système modulaire consiste alors à apprendre comment changer le module de manière à avoir un comportement optimal. Chacun des deux systèmes effectue simultanément un apprentissage des politiques partielles, et un apprentissage de la meilleure façon de passer d'un module à un autre.

Dans *HQ-learning* (Wiering et Schmidhuber, 1997), les modules sont ordonnés et activés en séquence selon un ordre préétabli. Chacun d'eux, en plus de sa table de qualités concernant l'optimalité des actions, gère une seconde table de qualités. Cette dernière permet de décider quand le module courant doit devenir inactif au profit du module suivant. L'apprentissage de ces nouvelles qualités se fait en fonction de la récompense attribuée au système global. Puisque les modules sont activés séquentiellement et selon un ordre préétabli, ce type de système ne peut résoudre que des problèmes dans lesquels l'état de départ est toujours le même.

Le mécanisme utilisé par Sun et Peterson (2000) est plus général puisqu'il ne suppose pas la définition *a priori* d'un ordre entre les modules. À chaque pas de temps, chaque module fait une enchère. Celui qui fait l'enchère la plus haute prend la main. Les enchères proposées par chaque module sont données par une fonction de la récompense globale du système. Si, dans cette architecture, les modules ne sont pas ordonnés, leur nombre reste spécifié à l'avance et suppose une connaissance du problème de la part du concepteur.

Dans chacun de ces systèmes, un seul module est actif à la fois. Selon le module actif, le système ne choisira pas le même comportement. L'activité des différents modules est donc un état interne du système, dans un sens très proche de celui développé par Lanzi. En outre, les mécanismes d'arbitrage et de passage de relais entre les modules sont tels que chaque module reste actif un certain nombre de pas de temps consécutifs. Ce type de découpage permet donc de gérer des états internes en assurant leur stabilité, de manière à ce que les états internes soient plus intelligibles.

Donc, ce type de méthode, qui consiste à découper un problème non-markovien en plusieurs sous-problèmes markoviens, semble offrir à la fois des vertus de compacité, puisque l'on n'a pas recours à une mémoire sur les pas de temps précédent, et des vertus d'intelligibilité, puisque l'état interne reste stable. De ce point de vue, cette solution consistant à découper un problème non-markovien en sous-problèmes non-markoviens nous semble la meilleure par rapport à toutes celles qui ont été exposées précédemment.

Toutefois, les systèmes existant à ce jour et qui utilisent cette technique, nécessitent une spécification préalable du nombre d'états interne (du nombre de modules) mais, surtout, ils sont dévolus à un apprentissage par renforcement direct, sans modélisation de l'environnement par un apprentissage latent. Dans la suite, nous discutons de la réalisation d'un système modulaire d'apprentissage latent, qui ne nécessite pas de spécification préalable du nombre de modules.

6.2.8 MACS et les problèmes non-markoviens

Identification d'un problème non-markovien

Repérer le caractère non-markovien d'un environnement est facile avec des systèmes d'apprentissage latent comme MACS. En effet, lorsque que la partie `Condition` d'un classeur est complètement spécialisée mais que le classeur continue à osciller, c'est qu'il y a un problème d'information cachée. S'il est complètement spécialisé, c'est qu'il considère une situation particulière au lieu d'un ensemble de situations généralisées. Dès lors, s'il oscille, c'est que, dans cette situation, l'action du classeur mène parfois à une situation, parfois à une autre. Cette oscillation peut être le fait du caractère stochastique d'un environnement ou bien de son caractère non-markovien.

Dans le cas de MACS, si un classeur oscille, l'environnement est considéré non-markovien. Ce moyen d'identifier des ambiguïtés ne fonctionne que si l'environnement est déterministe. Sans cette hypothèse, discriminer les fluctuations dues à un caractère stochastique des fluctuations dues à un caractère partiellement observable de l'environnement est difficile.

Une fois le caractère ambigu de certaines situations établi, il faut prendre en compte cette information pour apprendre à agir de manière optimale, malgré tout. Dans la suite, nous montrons comment les anticipations partielles de MACS sont un obstacle à la fois à l'utilisation de la technique de chaînage d'actions utilisée par ACS, et découpage du problème non-markovien en sous-problèmes markoviens.

Création de chaînes d'actions

Pour aborder des problèmes non-markoviens, la première solution proposée par les systèmes de classeurs à anticipation est celle développée dans ACS. Elle consiste à construire des chaînes d'actions, comme expliqué dans la section 6.2.5, en agrégeant des classeurs. Pour ce faire, il faut que chaque classeur représente une ou plusieurs transitions. Dès lors, il est possible d'utiliser l'effet du premier classeur comme condition du second.

Or, dans MACS, les parties `Effet` ne représentent que des anticipations partielles et ne permettent pas de définir une situation initiale hypothétique propre à coïncider avec la partie `Effet` du second classeur. Dès lors, MACS complique considérablement le chaînage d'actions tel qu'il est fait dans ACS. Si les anticipations partielles permettent de représenter de nouvelles régularités dans les interactions entre l'agent et son environnement, elles constituent un obstacle à la création de séquences d'actions.

Définition d'un partitionnement des états

Plutôt que de procéder par chaînage d'actions, on pourrait utiliser une solution consistant à découper un problème en sous-problèmes. Cette solution offre même l'avantage de prendre en compte des environnement non-markoviens avec une grande économie d'informations additionnelles, tout en garantissant un bonne stabilité de l'état interne[40]. Nous montrons que les anticipations partielles de MACS font aussi obstacle à la définition d'un partitionnement des états.

Nous avons vu comment l'identification de classeurs oscillants mais complètement spécialisés permet d'identifier les situations ambiguës. Il est alors possible de commencer par aborder un problème quelconque comme s'il était markovien, c'est-à-dire sans état interne d'aucune sorte. Si le problème se montre non-markovien, il est alors possible d'introduire un état interne en découpant le problème. Pour ce faire, on peut ajouter un module au système global qui n'en comptait qu'un seul au début. Chacun de ces modules aura en charge l'anticipation dans un sous-ensemble de l'espace des états. En particulier, ce découpage devra permettre de distinguer la situation identifiée comme problématique. Si ce premier découpage ne suffit pas et, si à l'intérieur de certains modules, il reste des ambiguïtés, il sera toujours possible de redécouper, jusqu'à introduction du nombre de modules adéquats. Par rapport aux systèmes présentés à la section

[40]En outre, si le chaînage nous semble une bonne solution dans le cadre de l'apprentissage par renforcement direct, comme c'est le cas de CXCS, cette solution ne nous paraît pas aussi bonne dans le cadre de l'apprentissage latent. En effet, l'apprentissage d'un modèle complet de l'environnement suppose que toutes les transitions soient correctement modélisées, qu'elles rapprochent ou qu'elles éloignent de l'objectif alors que, dans le cas de l'apprentissage par renforcement, on ne retient que les chaînes qui rapprochent de l'objectif. Ainsi, le nombre de chaînes à constituer pour l'apprentissage latent est beaucoup plus important.

6.2.7, un tel découpage incrémental éviterait d'avoir à spécifier à l'avance le nombre de modules nécessaires.

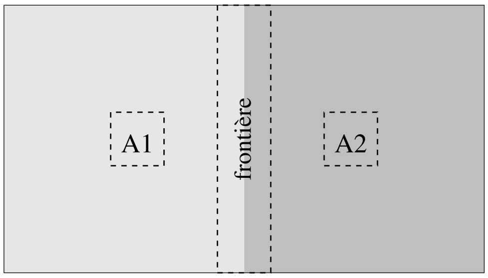

Fig. *6.7: Définition d'une frontière entre deux zones de l'environnement.*

Pour découper un problème non-markovien en sous-problèmes markoviens, il convient de définir un mécanisme permettant de passer d'un module à l'autre. Une première solution envisageable pour définir un tel mécanisme serait d'identifier une frontière constituée de situations, comme dans la figure 6.7. Dès lors, la perception d'une situation frontière entraînerait un changement de module, et donc une modification de l'état interne. Les situations A1 et A2 sont identiques mais doivent être distinguées. Pour ce faire, d'une manière ou d'une autre, on construit une frontière entre les deux situations identiques, de manière à découper le problème non-markovien en deux sous-problèmes markoviens.

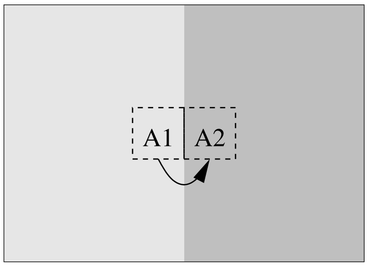

Fig. *6.8: Impossibilité à définir une frontière.*

Malheureusement, cette technique ne peut fonctionner que s'il est possible de construire une telle frontière. Or, ce n'est pas toujours le cas, comme le montre la figure 6.8. Dans cet exemple, les situations ambiguës sont côte à côte et il n'y a aucune situation entre les deux qui puisse faire office de frontière. Le changement de module doit donc être spécifié sur une transition et pas sur une situation. Dans notre exemple, on changerait de module dès que la transition entre A1 et A2 surviendrait. Dans *HQ-learning* et le système de Sun, c'est le choix qui a été fait.

Ici encore, les anticipations partielles de MACS sont un obstacle. En effet, puisqu'aucun classeur ne définit une transition complète, il est impossible d'associer aux classeurs une information concernant le changement de modules.

Dans ce chapitre, nous avons examiné différentes manières d'aborder les problèmes non-markoviens et nous avons souligné comment, même s'il permet de représenter de nouvelles régularités, MACS rend difficile la prise en compte de ces problèmes. En conséquence, il est difficile de l'employer tel quel pour des applications réelles. Surmonter ces difficultés avec MACS est donc un problème difficile qui reste à résoudre.

Pour des applications réelles, un grand nombre de problèmes non-markoviens peuvent être résolus de manière acceptable, pour peu que l'on soit capable de traiter le cas stochastique (Jaakkola *et al.*, 1995). Par ailleurs, la prise en compte d'attributs numériques continus éviterait une discrétisation préalable. Nous proposons dans le chapitre suivant une voie possible pour traiter les attributs continus et bruités avec des systèmes de classeurs, en tirant parti de l'anticipation.

Les propositions suivantes s'appuient sur une volonté d'unification des méthodes utilisées pour l'apprentissage latent et pour l'apprentissage par renforcement dans le cadre des systèmes de classeurs. Dans tous les cas, il s'agit de trouver l'approximation d'une fonction – de récompense ou de transition – et nos travaux en cours visent à apporter une solution efficace à ce problème général.

Chapitre 7

De nouvelles perspectives pour les systèmes de classeurs

7.1 Vers un système de classeurs générique pour l'approximation de fonctions

À la section 4.5, nous avons indiqué comment MACS permet de modéliser plusieurs fonctions de transition partielles $T_i\ : s_1 \times ... \times s_d \times a_1 \times ... \times a_e \to s_i$ pour définir la fonction de transition globale $T\ : s_1 \times ... \times s_d \times a_1 \times ... \times a_e \to s_1 \times ... \times s_d$. Une telle fonction partielle est illustrée par la figure 7.1.

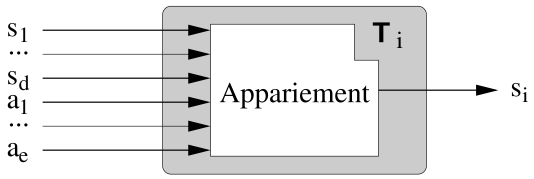

FIG. *7.1: Fonction d'anticipation partielle.*

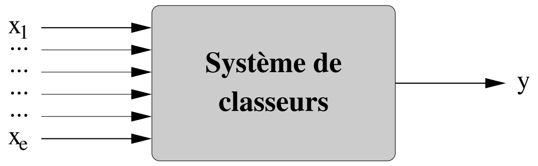

FIG. *7.2: Système de classeur général.*

Dans les systèmes de classeurs traditionnels, la partie `Action` est distincte de la partie `Condition`. En outre, dans ACS, YACS ou MACS, la généralisation n'opère pas sur les parties `Action`. Pourtant, en formulant le problème de l'anticipation comme un problème d'approximation de la fonction de transition $T\ : s_1 \times ... \times s_d \times a_1 \times ... \times a_e \to s_1 \times ... \times s_d$, on remarque que les attributs a_i ont un rôle équivalent aux attributs s_i parmi les attributs d'entrée : ce sont tous des arguments de la fonction dont on doit apprendre une approximation. Dans le processus d'apprentissage, les actions ne devraient donc pas être considérés comme des attributs particuliers. Sans faire cette distinction dans le processus d'apprentissage, il devient aussi naturel d'utiliser la généralisation pour les actions que pour les attributs des situations

Cette constatation vaut pour l'ensemble des systèmes de classeurs. Comme le propose Wilson (2002), la nature des arguments (attributs des situations ou actions) ne doit pas intervenir dans le processus d'apprentissage. Cette distinction sémantique ne doit intervenir que durant la phase d'exploitation des résultats de l'apprentissage. Dans le cas de l'apprentissage latent, on ne distingue les actions et les situations que lorsque l'on cherche à utiliser le modèle de la fonction de transition pour anticiper.

Dans ce chapitre, nous développons l'idée d'un système de classeurs général, qui serait défini comme un approximateur de fonctions quelconques à base de règles (voir figure 7.2), sans distinction entre actions et situations dans le processus d'apprentissage. En redéfinissant le problème dévolu aux systèmes de classeurs comme un problème d'approximation de fonctions, on peut réaliser de manière similaire un apprentissage latent et un apprentissage par renforcement. En effet, que l'on cherche à approximer une fonction de récompense ou une fonction de transition partielle, on cherche toujours à approximer une fonction.

On peut donc utiliser plusieurs systèmes de classeurs pour réaliser un apprentissage latent du modèle de l'environnement. Chaque système apprend l'approximation d'une fonction de transition partielle permettant d'anticiper la valeur d'un attribut particulier.

Un autre système de classeurs permet d'approximer la fonction :

$$Q \ : s_1 \times ... \times s_d \times a_1 \times ... \times a_e \rightarrow \Re$$

Cette dernière approximation peut être accélérée par l'utilisation d'un apprentissage latent d'un modèle de l'environnement, en utilisant une architecture *Dyna* pour intégrer tous les systèmes de classeurs, comme cela a été fait avec MACS.

7.1.1 Qu'est-ce qu'un classeur ?

En mettant de côté les différences entre actions et attributs perçus, un classeur devient une règle composée de deux éléments seulement :

- une partie condition qui doit être appariée avec les entrées du système. Cet élément du classeur définit une restriction du domaine de validité du classeur ;
- une partie décision qui propose une valeur de sortie en fonction des valeurs d'entrée.

La partie `Condition` doit pouvoir apparier des entrées symboliques, comme c'est le cas dans MACS, mais aussi numériques. Dans le cas d'attributs symboliques, la généralisation opère grâce à l'utilisation des symboles `#`. Dans le cas d'attributs prenant des valeurs numériques et donc ordonnées, il est possible d'utiliser des intervalles pour définir le domaine de validité d'un classeur, comme cela a été fait dans le cadre de XCSI (Wilson, 2001b) pour des problèmes de fouille de données. Avec autant d'intervalles que d'attributs, une partie `Condition` restreint le domaine de validité du classeur selon toutes les dimensions. En conséquence, l'ensemble des parties conditions des différents classeurs découpe l'espace des situations possibles en autant d'hyper-cubes qu'il y a de classeurs. En effet, les restrictions apportées par les parties `Condition` utilisent des constantes, et le découpage de l'espace des états se fait parallèlement aux axes des différentes dimensions correspondant aux attributs (voir figure 7.3).

La partie décision du classeur propose une valeur numérique ou symbolique, quand la condition est satisfaite. Elle calcule une valeur de sortie en fonction des entrées. Si la valeur que doit

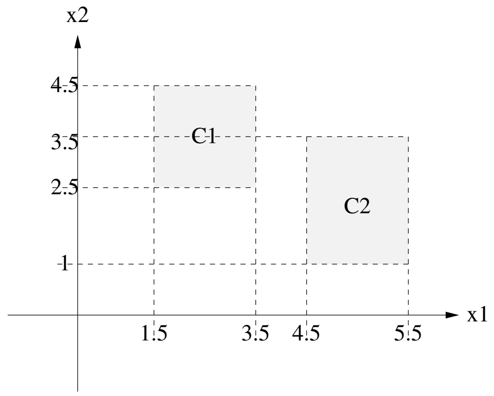

FIG. *7.3: Utilisation d'intervalles pour définir des domaines de validité numériques.* Dans cet exemple, il y a deux attributs numériques x_1 et x_2. La condition du classeur C1 est $[1.5 \rightarrow 3.5][2.5 \rightarrow 3.5]$, et la condition de C2 est $[2.5 \rightarrow 4.5][1 \rightarrow 3.5]$.

proposer le classeur est symbolique, la partie décision est une fonction constante renvoyant toujours le même symbole. Si le classeur doit proposer une valeur numérique, la partie décision est utilisée pour calculer la sortie du classeur en fonction des valeurs des attributs numériques. Les attributs symboliques ne servent plus alors qu'à la restriction du domaine de validité du classeur.

Une partie décision caractérisée par un seul paramètre ne permet de spécifier qu'une fonction constante, qui renvoie le paramètre en question. C'est le cas de XCS, où la fonction de décision de chaque classeur est paramétrée par une constante numérique. Cette constante est la prédiction du renforcement attendu. Chaque classeur de XCS définit donc un sous-ensemble de l'espace des situations. À l'intérieur de chaque sous-espace, la valeur proposée est constante. En conséquence, XCS propose une approximation constante par morceaux de la fonction de récompense attendue.

7.2 Approximation de fonctions numériques

Dans cette section, nous faisons des propositions pour la conception d'un système de classeurs permettant d'apprendre l'approximation d'une fonction numérique quelconque, c'est-à-dire d'en apprendre un modèle.

7.2.1 Approximation linéaire par morceaux

XCSI (Wilson, 2001b) permet de trouver une approximation pour une fonction. Par exemple, utilisé pour approximer une fonction avec un seul argument, ce système peut permettre de modéliser la fonction $f\ : x \rightarrow x^2$ entre 0 et 2 grâce aux trois classeurs suivants :

- $[0.0 \rightarrow 1.0], v = 0.3$
- $[1.0 \rightarrow 1.5], v = 1.6$
- $[1.5 \rightarrow 2.0], v = 3.1$

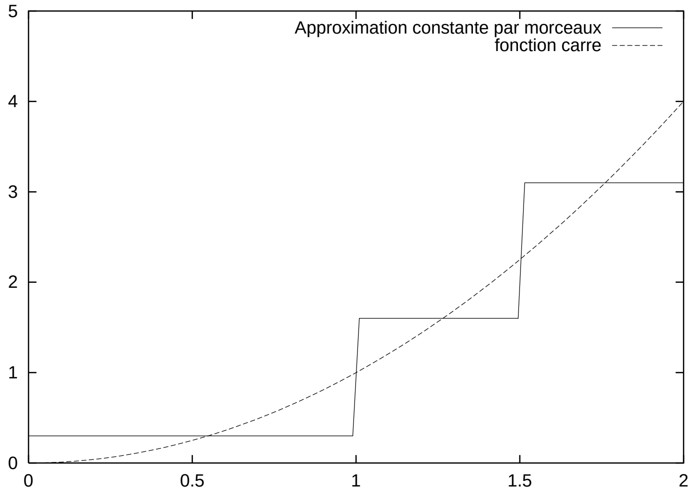

FIG. 7.4: *Approximation constante par morceaux de* $f\ : x \to x^2$.

La partie entre crochets indique, pour chaque classeur, son domaine de validité, et la seconde partie indique la valeur proposée par le classeur dans cet intervalle. Cette approximation est illustrée par la figure 7.4. Une approximation de cette nature est souvent très imprécise et on peut lui préférer une approximation linéaire par morceaux comme celle de la figure 7.5.

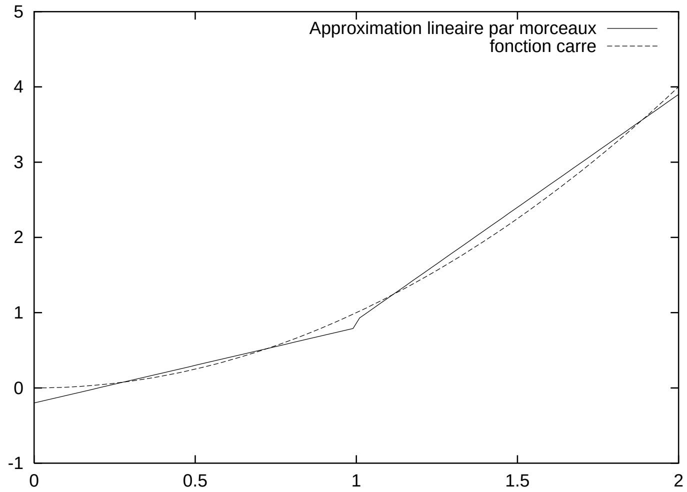

FIG. 7.5: *Approximation linéaire par morceaux de* $f\ : x \to x^2$.

Dans ce cas, comme le propose Wilson avec XCSF (Wilson, 2001a; Wilson, 2002), la partie décision du classeur définit non plus une constante, mais une fonction linéaire des attributs numériques. Pour ce faire, la fonction de décision de chaque classeur est paramétrée par

– une valeur par attribut numérique, qui représente la pente de l'hyper-plan par rapport à l'axe représentant cet attribut ;

– une valeur représentant la valeur à l'origine, c'est-à-dire lorsque tous les attributs sont nuls.

Par exemple, l'approximation linéaire par morceaux de la figure 7.5 s'obtient avec les deux classeurs suivants :

– $[0.0 \rightarrow 1.0], a = 1, b = -0.2$
– $[1.0 \rightarrow 2.0], a = 3, b = -2.1$

Dans ce cas, les valeurs a et b associées à chaque classeur caractérisent une fonction linéaire de décision rendant $y = ax + b$ en fonction d'une valeur d'entrée x.

À condition de multiplier les intervalles et les paramètres de la fonction linéaire associée à chaque classeur, ces exemples sont généralisables à un nombre quelconque de dimensions. Si l'on utilise de tels classeurs pour calculer la valeur y que propose le modèle pour les valeurs d'attribut $(x_1, \cdots, x_d)$, on sélectionne d'abord le classeur dont chaque intervalle de la partie `Condition` contient la valeur de l'attribut correspondant. Les paramètres de la fonction linéaire sont ensuite utilisés pour calculer la valeur de sortie en fonction du vecteur d'entrée $(x_1, \cdots, x_d)$.

7.2.2 Utilisation de statistiques descriptives

Estimations statistiques

Pour définir un tel partitionnement, XCSF utilise des algorithmes génétiques pour trouver les intervalles constituant la partie `Condition` des classeurs. Si ces algorithmes sont très utiles pour permettre d'atteindre un optimum global, en permettant d'explorer efficacement un très large spectre de l'espace des solutions, on peut en revanche craindre que ce ne soit au détriment de leur rapidité à trouver une solution. En effet, localement, ces algorithmes ne proposent que des modifications aléatoires, qui ne seront évaluées qu'*a posteriori*. L'utilisation d'estimations et d'heuristiques pour guider la recherche de classeurs fiables doit permettre d'accélérer ce processus. Pour ne pas souffrir des mêmes problèmes que les heuristiques de YACS et MACS, il convient d'en définir de plus souples pour permettre de prendre en compte des environnements stochastiques ou éviter des *optima* locaux. Ainsi, il nous paraît souhaitable d'utiliser des heuristiques dès que cela est possible, tout en palliant les problèmes d'optimalité locale qui pourraient en résulter grâce à l'utilisation d'algorithmes génétiques.

Dans la suite, nous présentons brièvement quelques travaux préliminaires pour la définition d'un nouveau système de classeurs dévolu à l'approximation linéaire par morceaux de fonctions numériques. Ce nouveau système utilise des estimations dérivées de méthodes statistiques de régression linéaire multivariée (Rao, 1970). Ces méthodes permettent de trouver des approximations linéaires de fonctions, et sont robustes au bruit. L'utilisation de méthodes statistiques doit permettre de gérer des informations présentant un caractère stochastique. Dans la suite, nous identifions certains problèmes difficiles pour les méthodes statistiques, et nous proposons une méthode utilisant des algorithmes génétiques pour les résoudre.

Pour l'approximation linéaire par morceaux au moyen de systèmes de classeurs, deux problèmes se posent :

– partitionner l'espace des états en sous-espaces dans chacun desquels une approximation

linéaire est une bonne approximation. Ce problème est celui de la découverte de parties `Condition` pour un ensemble de classeurs bien adapté à la résolution de problème d'approximation ;
- étant donné une telle partition, trouver les poids des fonctions linéaires et être capable d'estimer une erreur, de manière à pouvoir corriger la partition. Ce problème est celui de la définition de la fonction de décision d'un classeur, et de la mise à disposition d'estimations d'erreur sur cette fonction.

Nous allons montrer comment traiter le second problème avant d'évoquer la solution que nous envisageons pour traiter le premier.

Notations

Pour le problème de l'approximation linéaire à l'intérieur d'un certain domaine de validité, il convient d'associer, à chaque classeur, un vecteur de poids caractérisant une fonction linéaire. Notons le W,

$$W = (w_0, w_1, \cdots, w_d)$$

Nous notons d le nombre de dimensions de la fonction à approximer. C'est aussi le nombre d'attributs. Les poids $w_{i \in [0,d]}$ sont utilisés pour calculer une valeur de sortie y en fonction des valeurs des attributs d'entrée $x_{i \in [1,d]}$.

$$y = w_0 + w_1 x_1 + \cdots + w_d x_d$$

En prenant une notation matricielle et en considérant que les attributs d'entrée forment un vecteur $\mathbf{x}$ et que ce vecteur est toujours augmenté d'une constante 1 pour l'utilisation du poids w_0, on a :

$$\mathbf{x} = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{pmatrix}$$

ou t désigne la transposée. Dès lors, on a :

$$y = \mathbf{x}W$$

Considérons maintenant la matrice X_k rassemblant tous les vecteurs d'entrée jusqu'au pas de temps k, et Y_k le vecteur rassemblant les valeurs de sortie observées $y_0 \cdots y_k$. X_k est une matrice de $k+1$ lignes et de $d+1$ colonnes et Y_k est un vecteur colonne de dimension $d+1$.

$$X_k = \begin{pmatrix} \mathbf{x}^t{}_0 \\ \vdots \\ \mathbf{x}^t{}_k \end{pmatrix} \qquad\qquad Y_k = \begin{pmatrix} y_0 \\ \vdots \\ y_k \end{pmatrix}$$

Régression linéaire multivariée, exacte et incrémentale

La régression linéaire multivariée (Rao, 1970) pour trouver le vecteur de poids W_k s'effectue en utilisant la formule :

$$W_k = X_k^+ Y_k$$

où $^+$ désigne une pseudo-inverse. En développant cette formule, on a :

$$W_k = (X_k^t X_k)^{-1}(X_k^t Y_k)$$

Les deux termes $X_k^t X_k$ et $X_k^t Y_k$ sont des matrices de covariance de taille constante, quel que soit k. $X_k^t X_k$ est de dimension $(d+1)^2$ et $X_k^t Y_k$ est un vecteur de dimension (d+1). En outre, il est possible d'écrire une formule de récurrence propre à exprimer chacun de ces termes au rang $k+1$ en fonction de leur valeur au rang k et du nouvel exemple $(\mathbf{x_{k+1}}, y_k)$:

$$X_{k+1}^t X_{k+1} = X_k^t X_k + \mathbf{x_{k+1}} \mathbf{x_{k+1}}^t$$

et

$$X_{k+1}^t Y_{k+1} = X_k^t Y_k + \mathbf{x_{k+1}} \mathbf{y_{k+1}}^t$$

Ainsi, en utilisant ces formules et en associant deux matrices de covariance à chaque classeur, il est possible d'obtenir une estimation exacte des poids de la fonction linéaire rendant mieux compte des données observées. Pour calculer des poids semblables, dans XCSF, Wilson utilise l'algorithme de Widrow-Hoff, qui est plus lent, moins robuste au bruit et, surtout, qui ne donne pas de solution exacte.

Détermination de l'erreur

En ce qui concerne le partitionnement de l'espace des états en sous-espaces permettant une approximation linéaire par morceaux aussi compacte que possible, comme dans YACS et dans MACS, nous utilisons des généralisations et des spécialisations explicites, guidées par des estimations statistiques. La première estimation nécessaire est celle de l'erreur commise sur la régression linéaire. Une erreur trop grande nécessite de découper plus avant le sous-espace d'états spécifié par la partie `Condition`. Cette estimation peut aussi être utilisée dans les systèmes de classeurs n'utilisant que des algorithmes génétiques, comme valeur sélective. Ainsi, nous pensons que XCSF gagnerait à utiliser une estimation donnant l'erreur exacte plutôt que d'utiliser des méthodes de régression comme Widrow-Hoff

En statistiques descriptives, une manière habituelle de calculer une erreur normalisée consiste à calculer un coefficient $r^2 \in [0,1]$ rendant compte de la part des données qui est correctement expliquée par le modèle linéaire. Ce coefficient est calculé de la manière suivante :

$$r^2 = \frac{\sigma_t - \sigma_r}{\sigma_t}$$

La valeur σ_t est la somme des carrés des écarts à la moyenne $\overline{y}$ des valeurs de sortie observées $y_{i \in [0,k]}$. Elle représente la variation totale des valeurs de sortie. La valeur σ_t est la variation des valeurs de sortie qui n'est pas expliquée par le modèle linéaire. En conséquence, $\sigma_t - \sigma_r$ est la variation expliquée, et r^2 est la part des variations expliquées par le modèle par rapport à la variation totale. Si $r^2 = 1$ alors le modèle explique parfaitement les données et si $r^2 = 0$ alors le modèle ne rend pas du tout compte des données.

La valeur σ_t est une simple somme de carrés de différences :

$$\sigma_t = \sum_{i=0}^{k} (\overline{y} - y_i)^2$$

Elle peut être calculée de manière incrémentale pour peu que l'on associe à chaque classeur sa valeur de k et celle de $\overline{y}$. La valeur σ_r est :

$$\sigma_r = |Y_k - X_k W_k|^2$$

Ici encore, les matrices Y_k et X_k ne sont pas de dimensions fixes et leur taille croît à mesure que le nombre de dimensions k devient plus grand, mais σ_r reste calculable de manière incrémentale. En effet, développer l'équation précédente donne :

$$\sigma_r = Y_k^t Y_k - 2W_k^t X_k^t Y_k + W_k^t X_k^t X_k W_k$$

Les valeurs $X_k^t Y_k$ et $X_k^t X_k$ sont déjà connues, et $Y_k^t Y_k$ est une simple somme de carrés, qu'il est facile de calculer incrémentalement.

Donc, tout ce qui est nécessaire pour calculer une estimation exacte de l'erreur normalisée r^2 est calculable incrémentalement. Nous croyons que ce type d'estimations statistiques devrait retenir l'attention de la communauté des chercheurs sur les systèmes de classeurs. Dans notre cadre, ils devraient se révéler précieux pour décider des spécialisations et des généralisations. Ce point est discuté dans la section suivante.

Un compromis entre la taille du modèle et sa précision

Se poser le problème de la généralisation, c'est chercher un bon compromis entre le nombre de paramètres pour modéliser une fonction, et la précision de l'approximation. Augmenter le nombre de modèles linéaires, c'est-à-dire augmenter le nombre de classeurs, c'est augmenter la taille du modèle général, linéaire par morceaux, mais c'est aussi offrir la possibilité de faire diminuer l'erreur commise par le modèle. Ainsi, généraliser, c'est prendre le risque d'augmenter l'erreur, mais c'est diminuer la taille du modèle. Spécialiser, c'est diminuer l'erreur mais augmenter la taille du modèle.

Nous faisons ici des propositions pour des mécanismes de généralisation et de spécialisation, c'est-à-dire pour trouver incrémentalement des classeurs dont les parties `Condition` sont adaptées à une modélisation à la fois compacte et fiable.

Dans le problème de l'approximation linéaire par morceaux, l'estimation de l'erreur r^2 est précieuse pour décider si un classeur doit être spécialisé ou si deux classeurs devraient être agrégés pour n'en former plus qu'un seul.

Chacun des classeurs est un modèle linéaire. En conséquence, il est spécifiable par $d+1$ valeurs, ou d est le nombre de dimensions de la fonction à approximer, ou encore le nombre d'attributs. Spécialiser, c'est introduire un point de rupture. Par exemple, spécialiser un classeur dont la partie `Condition` est `[0,2][1,4]` pour créer deux classeurs de parties `Condition [0,1][1,4]` et `[1,2][1,4]`, c'est introduire le point de rupture 1 selon le premier attribut. Si le premier modèle était paramétrable par $d+1$ valeurs réelles, l'agrégation des deux modèles résultants nécessite $2(d+1)+1$ paramètres (deux modèles linéaires et un point de rupture). Cette complexité des modèles peut servir à définir un coût de la spécialisation, qu'il convient de mettre en balance avec le gain en précision attendu. Pour ce faire, nous envisageons d'utiliser les mesures de MDL (*Minimal Description Length*) issues de la théorie de l'information et utilisées dans C4.5 (Quinlan, 1993) par exemple.

Généralisation et spécialisation avec des algorithmes génétiques

Étant donné deux hypothèses suffisamment évaluées, l'une faisant intervenir un classeur et son erreur associée, et l'autre faisant intervenir deux classeurs et leurs erreurs respectives, les mesures de MDL permettent de décider de la pertinence d'une spécialisation ou d'une généralisation. Dans le cas de la généralisation, on peut, à partir de deux classeurs et de leurs matrices de covariance respectives, formuler une hypothèse en unissant les parties `Condition`, et en sommant les matrices de covariances, de manière à estimer l'erreur qui résulterait de l'agrégation des deux classeurs initiaux. Si la perte de précision n'est pas très importante en regard du gain en complexité du modèle résultant, on pourra procéder à la généralisation.

Dans le cas de la spécialisation, le problème est plus épineux. En effet, puisque nous nous imposons la contrainte de l'incrémentalité, nous ne pouvons pas stocker tous les couples $(\mathbf{x_i}, y_i)_{i\in[0,k]}$, nous ne pouvons stocker que les matrices de covariances associées. Ainsi, il est impossible d'utiliser l'ensemble des couples $(\mathbf{x_i}, y_i)_{i\in[0,k]}$ pour calculer le meilleur point de rupture. Une solution consiste à associer un certain nombre d'hypothèses à chaque classeur, et d'évaluer ces hypothèses en même temps que le classeur. Ces hypothèses forment une population sur laquelle il est possible d'utiliser un algorithme génétique de manière à trouver les meilleurs points de rupture, pour effectuer des spécialisations efficaces. Comme dans YACS et dans MACS, l'objectif reste de ne modifier le modèle (la liste de classeurs) que lorsque suffisamment d'informations auront été collectées pour valider la modification *a priori*, au lieu de modifier le modèle au hasard et de n'évaluer les modifications qu'*a posteriori*. Par conséquent, si les statistiques descriptives offrent de bons outils pour l'évaluation d'un modèle, elles restent limitées à l'évaluation et à la définition d'estimations. Seuls, elles sont insuffisants pour permettre de décider d'une spécialisation. Les estimations que nous proposons doivent donc être utilisées pour biaiser un mécanisme de recherche globale comme les algorithmes génétiques.

Malgré toute la prudence dont nous pourrions faire preuve, la contrainte de l'incrémentalité implique que certaines spécialisations seront décidées alors que les données collectées ne couvriront pas l'espace des états de manière homogène. C'est typiquement ce qui se produit dans le cas d'une exploration locale. Nous aurons donc toujours besoin d'un mécanisme de généralisation qui permette de reconsidérer les spécialisations sous-optimales. Un premier mécanisme de généralisation, comme celui évoqué plus haut, permet d'utiliser la généralisation de manière raisonnée, sans trop perdre en précision du modèle. Pourtant, ce type de généralisation ne permet pas de sortir d'éventuels optima locaux en termes de taille du modèle. En effet, les bornes des intervalles initiaux auront été décidées auparavant par la spécialisation, et des généralisations incrémentales et locales risquent de provoquer des inter-blocages empêchant de reconsidérer en profondeur des choix faits au début de l'apprentissage. Il est alors nécessaire d'introduire de la redondance, et d'accepter qu'un point de l'espace soit couvert par plusieurs classeurs.

Pour réaliser un mécanisme de généralisation, l'utilisation de règles plutôt que d'arbres de décision apporte plus de souplesse puisqu'il n'est jamais nécessaire d'appréhender une structure arborescente et contraignante. Le système gère alors une simple collection de règles indépendantes[41]. Que plusieurs règles couvrent le même sous-espace d'états ne pose pas de difficulté, c'est d'ailleurs ce que font les systèmes de classeurs fondés sur des algorithmes génétiques. Cette redondance permet de considérer en parallèle différentes solutions au problème du partitionnement de l'espace des états, et d'éviter des problèmes de sous-optimalité. Pour assurer cette redondance salutaire, il suffit de permettre sporadiquement, et à une fréquence maîtrisée, une généralisation ne satisfaisant pas aux critères de qualité du compromis entre précision et complexité, sans supprimer les classeurs originaux. Une telle généralisation sous-optimale offrira de nouvelles possibilités de spécialisation puis de re-généralisation, et permettre d'extraire le système entier d'un optimum local quant à la compacité du modèle. Mais de telles modifications du modèle doivent rester marginales et mesurées, de manière à ne pas nuire à l'efficacité générale du système, ni à la découverte rapide d'un partitionnement adéquat assurant la précision du modèle. La compacité et l'intelligibilité du modèle restent importantes et ont tout à gagner à l'utilisation d'algorithmes génétiques, mais il ne faut pas que cela se fasse au détriment de la rapidité de convergence vers une approximation fiable.

Dans ce chapitre, nous avons considéré le problème assigné aux systèmes de classeurs comme celui, général, de l'approximation de fonctions numériques ou symboliques. L'apprentissage latent et l'apprentissage par renforcement sont alors des aspects particuliers de ce problème général. Ces considérations étendent le champ d'application des systèmes de classeurs hors de l'apprentissage par renforcement.

Un système de classeurs général comme celui que nous envisageons serait capable de traiter des problèmes d'apprentissage tels que, selon la classification de Mitchell (1997) :

- à chaque exemple est associée la valeur que le système doit prédire (l'apprentissage est dit « supervisé ») ;

[41]Les arbres de décision, comme les règles, garantissent une indépendance des modèles puisque leurs domaines de validités sont clairement indépendants. Les solutions de partitionnement utilisant des méthodes comme les plus proches voisins rendent les domaines de validité des différents modèles dépendants les uns des autres, et empêchent l'utilisation d'estimations statistiques telles que celles développées ici.

- les exemples sont présentés les uns après les autres, leur nombre est indéterminé et chaque exemple permet de réviser ce qui a déjà été appris (l'apprentissage est dit « en ligne »)
- la fonction à modéliser peut éventuellement changer au cours de l'apprentissage (l'apprentissage est dit « non stationnaire »)

Intégré dans une architecture logicielle plus complète, ce type d'approximateur de fonctions permettrait d'aborder d'autres classes de problèmes. Un premier exemple illustrant cette démarche est donné par Wilson (2001b), qui utilise XCSI pour des problèmes de fouille de données. À partir d'un grand nombre d'exemples et de classes qui leur sont associées, il faut que le système apprenne à classer n'importe quel exemple qui serait proposé ultérieurement. Si les attributs d'entrée représentent les symptômes d'un patient, par exemple, on cherchera à déterminer s'il est malade ou non. La fonction dont on cherche une approximation est ici celle qui, en fonction de symptômes, permet de décider si un patient appartient à la classe « malade » ou bien « sain ». L'apprentissage opère ici à partir d'une base de données d'exemples préétablis. Wilson (2001b) utilise un protocole très simple de présentation des exemples qui permet à XCSI, un système d'apprentissage en ligne, de réaliser un apprentissage de type *« batch »*[42] pour résoudre des problèmes de fouille de données. Un second exemple concerne XCS. Les équations de Bellman sont ici utilisées pour calculer, en fonction de la prédiction de certains classeurs, une qualité qui est utilisée pour superviser la mise à jour d'autres prédictions. Le processus calculant ces qualités vient en supplément du noyau de XCS, dont le rôle strict est la mise à jour des prédictions, le calcul des valeurs adaptatives et l'application de l'algorithme génétique. Ce processus additionnel permet à XCS d'aborder des problèmes nécessitant plusieurs actions successives pour être résolus.

Nous considérons donc un système de classeurs comme un module élémentaire qui permet d'apprendre en ligne l'approximation d'une fonction quelconque. De tels modules peuvent être utilisés dans des architectures logicielles conçues pour résoudre des problèmes particuliers. Par exemple, pour l'apprentissage par renforcement et l'apprentissage latent, on peut utiliser une architecture inspirée de celle de MACS.

[42] L'apprentissage en mode *batch* construit une classification à partir d'un nombre prédéterminé d'exemples, par opposition à l'apprentissage en ligne qui opère sur un nombre indéterminé d'exemple

Conclusion

Dans cette thèse, nous avons décrit deux systèmes, YACS et MACS, pour aborder la question de l'anticipation dans le cadre des systèmes de classeurs.

Dans ce cadre, nous avons principalement abordé deux questions :

- celle de l'utilisation d'estimations pour guider la recherche de classeurs permettant d'anticiper les conséquences des actions ;
- celle des régularités à utiliser pour la généralisation dans l'apprentissage latent.

YACS aborde la première question en utilisant le même formalisme qu'ACS, qui nous a servi de référence dans cette thèse, mais en proposant de nouvelles heuristiques, et surtout des estimations pour guider des spécialisations et des généralisations explicites.

MACS utilise un nouveau formalisme pour pouvoir exprimer de nouvelles régularités. Ce formalisme est tel que chaque classeur n'anticipe qu'un seul attribut. Il conduit à poser la question de l'utilité des parties `Effet` dans les systèmes de classeurs à anticipation, et à redéfinir un système de classeurs comme un système général d'approximation de fonctions, qui peut être utilisé indifféremment pour l'apprentissage latent et pour l'apprentissage par renforcement.

YACS et l'utilisation d'estimations

Comme ACS, YACS utilise des classeurs qui sont tous caractérisés par trois parties :

- une partie `Condition` qui spécifie le domaine de validité du classeur ;
- une partie `Action` qui définit l'action envisagée ;
- une partie `Effet` qui rend compte des changements observés dans l'environnement, quand l'action considérée est entreprise dans une des situations spécifiées par la condition.

YACS diffère d'ACS dans les mécanismes de construction des classeurs. Là où ACS lie fortement les attributs de la partie `Condition` aux attributs de la partie `Effet` pour effectuer des spécialisations, YACS les décorrèle. En effet, YACS ne postule aucun lien particulier entre les valeurs des attributs de même rang dans les parties `Condition` et `Effet`. Les parties `Condition` servent uniquement à spécifier dans quelles situations le classeur peut anticiper, et les parties `Effet` représentent les changements perçus, indépendamment des parties `Condition`. Contrairement à ACS, dans YACS, ces parties sont modifiées séparément par des mécanismes indépendants.

La différence entre YACS et ACS réside donc dans leurs mécanismes de généralisation et de spécialisation. Toutefois, YACS hérite de l'approche originale d'ACS, qui consiste à utiliser des heuristiques pour créer de nouveaux classeurs, au lieu de recourir à des algorithmes génétiques, comme c'est souvent le cas dans le cadre des systèmes de classeurs traditionnels. La plupart des systèmes de classeurs emploient en effet des opérateurs de mutation, qui généralisent ou spécialisent les classeurs indifféremment et de manière aléatoire, pour les évaluer *a posteriori*. De leur côté, ACS et YACS utilisent des spécialisations explicites et guidées par l'expérience, même si, à la différence de YACS, ACS a recours à des algorithmes génétiques pour effectuer des généralisations. YACS fonde ses heuristiques sur des estimations, qui indiquent quels attributs devraient être spécialisés ou généralisés en priorité pour améliorer le modèle. L'utilisation de ces estimations a permis de définir des mécanismes permettant à YACS d'apprendre environ trois fois plus vite qu'ACS dans les environnements testés.

Si les heuristiques utilisées pour la spécialisation et la généralisation sont efficaces, c'est parce que les estimations sur lesquelles elles reposent sont fiables. Ces estimations permettent d'évaluer quels attributs devraient être spécialisés ou généralisés en priorité. Puisque leur calcul est indépendant des heuristiques utilisées dans YACS, ces estimations pourraient être utilisées autrement, pour introduire des biais dans les mutations aléatoires d'algorithmes génétiques, par exemple. Ce faisant, selon l'importance des biais, il serait possible de gagner en vitesse d'apprentissage, sans perdre les bonnes propriétés des algorithmes génétiques. Des biais ont déjà été introduits dans des systèmes de classeurs qui utilisent des algorithmes génétiques pour accélérer l'apprentissage. Les estimations de YACS permettraient d'améliorer les biais utilisés pour la spécialisation, et les biais utilisés pour la généralisation pourraient aussi gagner à utiliser de telles estimations.

Nous n'avons utilisé que des mondes de cases pour les expérimentations, comme c'est souvent le cas dans la littérature sur les systèmes de classeurs. Toutefois, les procédures décrites dans ces travaux ne font aucune hypothèse spécifique à ce type d'environnement. Comme pour d'autres systèmes de classeurs, les mécanismes qui ont été décrits ici peuvent donc être utilisés pour d'autres types de problèmes, pour peu qu'ils soient représentables par un automate à états finis.

MACS et les parties `Effet` dans les systèmes de classeurs à anticipation

Les estimations utilisées dans YACS l'ont aussi été dans MACS, avec succès. MACS apprend un modèle de l'environnement à peu près 2.5 fois plus vite que YACS. Toutefois, au delà des performances, l'originalité de MACS réside essentiellement dans le formalisme qu'il utilise. MACS propose que chaque classeur n'anticipe plus une situation entière, mais un seul attribut. En regroupant les classeurs anticipant le même attribut, on obtient des modules définissant chacun une fonction d'anticipation partielle. Ces anticipations partielles conduisent à reconsidérer l'apprentissage latent tel qu'il est conçu dans ACS et dans YACS, c'est-à-dire en exploitant une partie `Effet` pour chaque classeur.

En effet, une troisième partie n'est utile que dans la mesure où chaque classeur définit une fonction partielle rendant plusieurs valeurs, au lieu d'une seule. Lorsqu'il n'y a qu'une seule valeur à anticiper, il n'y a plus vraiment lieu d'utiliser une partie `Effet`. Un système de classeurs de type XCS, qui cherche l'approximation d'une fonction, et donne la prédiction d'une valeur unique, pourrait tout à fait remplacer un module anticipant un attribut particulier. Un système de classeurs à anticipation comme MACS pouvant être vu comme une ensemble de modules indépendants, chacun anticipant un seul attribut, on pourrait aussi, pour apprendre un modèle de l'environnement, employer un ensemble de modules de type XCS, un pour chaque attribut.

Pour profiter des avantages de l'anticipation dans les systèmes de classeurs pour l'apprentissage par renforcement, il n'est donc pas nécessaire d'utiliser des systèmes de classeurs spécifiques. Une organisation modulaire permettrait d'avoir recours à des systèmes de classeurs classiques pour apprendre une approximation de la fonction de transition.

En outre, comme nous l'avons montré avec MACS, cette solution permet d'utiliser pour la généralisation, sans effort particulier, une classe plus large de régularités.

Une architecture *Dyna* utilisant des approximateurs de fonction

Un système qui peut être utilisé pour anticiper la valeur d'un attribut, peut aussi l'être pour prédire une valeur de récompense. Dans tous les cas, le système cherche à approximer une fonction. Dans le cas de l'anticipation, il approxime la fonction de transition et, dans le cas de l'apprentissage par renforcement, il approxime la fonction des qualités. Le même système peut donc être employé pour approximer n'importe quelle fonction.

La nature de la fonction modélisée par un système n'est pas importante pour le processus d'apprentissage. Cette distinction sémantique n'intervient que dans l'utilisation que l'on fait des fonctions modélisées, et de leur organisation dans un système modulaire. Pour l'apprentissage par renforcement, on peut par exemple utiliser plusieurs approximateurs de fonctions semblables, pour construire une architecture *Dyna*, et utiliser des techniques de recherche opérationnelle pour l'apprentissage.

Dès lors, il devient possible de n'utiliser que des systèmes de classeurs habituels comme XCS, et de profiter quand même des avantages de l'anticipation dans le cadre de l'apprentissage par renforcement. En particulier, on pourra reproduire des mécanismes d'exploration et d'exploitation tels que ceux décrits dans ce mémoire, de manière à apprendre rapidement un modèle et d'accélérer l'apprentissage d'une politique.

Un système de classeurs pour l'approximation de fonctions numériques

Pour la suite de nos recherches, nous nous proposons de réaliser un tel système modulaire pour l'apprentissage par renforcement. Pour ce faire, nous avons besoin d'un système permettant d'approximer n'importe quelle fonction.

Pour aborder des problèmes réels, comme des problèmes de robotique par exemple, il est souvent nécessaire de traiter des données numériques et continues. Or la plupart des systèmes de classeurs traitent de données symboliques et discrètes. Dans ce cas, la prise en compte d'attributs numériques nécessite une discrétisation préalable de la part du concepteur. En pratique, ce type de discrétisation n'est pas aisée, et on préfère pouvoir traiter les données numériques brutes.

En conséquence, notre premier objectif est maintenant d'aborder la question générale de l'approximation incrémentale de fonctions numériques à l'aide de systèmes de classeurs. Dans ce mémoire, nous avons évoqué d'autres voies de recherches pour faire suite à MACS, comme la prise en compte de l'incertain. Mais la direction que nous choisissons nous semble la plus prometteuse. Outre la prise en compte de situations continues, si on peut traiter des perceptions bruitées et stochastiques, on peut être en mesure de traiter de manière acceptable un grand nombre de problèmes faisant intervenir des ambiguïtés.

Wilson a déjà proposé un système pour modéliser des fonctions numériques, et en proposer des approximations linéaires par morceaux. Mais ce système emploie presque exclusivement des algorithmes génétiques, et il reste limité à des problèmes de petite dimension. Pour aborder des problèmes plus grands, nous comptons employer des estimations statistiques, exactes et robustes au bruit, pour guider la recherche des classeurs. Cependant, pour éviter les problèmes d'optimalité locale inhérents à ce type de méthodes, nous envisageons une utilisation mesurée d'algorithmes génétiques.

En intégrant l'approximation de fonctions numériques et symboliques comme suggéré dans le dernier chapitre de cette thèse, il devient possible de concevoir un système de classeurs pour traiter directement n'importe quel problème d'apprentissage supervisé, en ligne et non stationnaire[43]. L'intégration à une architecture logicielle, même simple, permettra de résoudre des problèmes sortant de ce cadre strict. Ces perspectives ne concernent donc pas uniquement le domaine de l'apprentissage par renforcement avec ou sans apprentissage latent, mais l'apprentissage dans les systèmes informatiques en général.

[43]selon la terminologie de Mitchell (1997)

Bibliographie

[Bellman, 1957] R. E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton NJ, 1957.

[Bertsekas, 1987] D. P. Bertsekas. *Dynamic Programming : Deterministic and Stochastic Models.* Prentice-Hall, 1987.

[Butz *et al.*, 2000a] M. V. Butz, D. E. Goldberg, et W. Stolzmann. Introducing a Genetic Generalization Pressure to the Anticipatory Classifier System Part I : Theoretical Approach. In Whitely et al. (2000), pages 34–41.

[Butz *et al.*, 2000b] M. V. Butz, D. E. Goldberg, et W. Stolzmann. Introducing a Genetic Generalization Pressure to the Anticipatory Classifier System Part II : Experimental Results. In Whitely et al. (2000).

[Butz *et al.*, 2001] M. V. Butz, D. E. Goldberg, et W. Stolzmann. Probability–Enhanced Predictions in the Anticipatory Classifier System. In Lanzi et al. (2001).

[Butz et Pelikan, 2001] M. V. Butz et M. Pelikan. Analyzing the Evolutionary Pressures in XCS. In Spector et al. (2001), pages 935–942.

[Butz et Wilson, 2002] M. V. Butz et S. W. Wilson. An Algorithmic Description of XCS. *Journal of Soft Computing*, 6(3–4) :144–153, 2002.

[Butz, 2002a] M. V. Butz. An Algorithmic Description of ACS2. In Lanzi et al. (2002), pages 211–229.

[Butz, 2002b] M. V. Butz. Biasing Exploration in an Anticipatory Learning Classifier System. In Lanzi et al. (2002), pages 3–22.

[Cliff *et al.*, 1994] éditeurs D. Cliff, P. Husbands, J. A. Meyer, et S. W. Wilson. *From Animals to Animats 3. Proceedings of the Third International Conference on Simulation of Adaptive Behavior (SAB94).* MIT Press, 1994.

[Donnart et Meyer, 1996] J. Y. Donnart et J. A. Meyer. Learning Reactive and Planning Rules in a Motivationally Autonomous Animat. *IEEE Transactions on Systems, Man, and Cybernetics – Part B : Cybernetics*, 3(26) :381–395, 1996.

[Dorigo et Bersini, 1994] M. Dorigo et H. Bersini. A Comparison of Q-Learning and Classifier Systems. In Cliff et al. (1994), pages 248–255.

[Dorigo, 1994] M. Dorigo. Genetic and Non–Genetic Operators in ALECSYS. *Evolutionary Computation*, 1(2) :151–164, 1994.

[Drogoul et Meyer, 1999] éditeurs A. Drogoul et J. A. Meyer. *Actes de Intelligence Artificielle Située*, Paris, France, 1999. Hermès.

[Gérard *et al.*, 2002] P. Gérard, W. Stolzmann, et O. Sigaud. YACS : a new Learning Classifier System using Anticipation. *Journal of Soft Computing : Special Issue on Learning Classifier Systems*, 2002.

[Gérard *et al.*, 2003] P. Gérard, O. Sigaud, et J.-A. Meyer. Combining Latent Learning and Dynamic Programming in MACS. *European Journal of Operational Research*, 2003. (to appear).

[Gérard et Sigaud, 2001] P. Gérard et O. Sigaud. Adding a generalization mechanism to YACS. In Spector et al. (2001), pages 951–957.

[Gérard, 2001] P. Gérard. Generalisation and Latent Learning in Learning Classifier Systems. In *Proceedings of the XIX EURO Summer Institute*, Toulouse, France, 2001.

[Goldberg, 1989] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, 1989.

[Grefenstette, 1985] éditeur John J. Grefenstette. *Proceedings of the 1st International Conference on Genetic Algorithms and their Applications (ICGA85)*. Lawrence Erlbaum Associates : Pittsburgh, PA, 1985.

[Hannebauer *et al.*, 2001] éditeurs M. Hannebauer, J. Wendler, et E. Pagello. *Balancing reactivity and Social Deliberation in Multiagent Systems*, volume 2103 de *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, 2001.

[Hansen, 1998] E. Hansen. *Finite Memory Control of Partially Observable Systems*. PhD thesis, University of Massachusetts, 1998.

[Hoffmann, 1993] J. Hoffmann. *Vorhersage und Erkenntnis [Anticipation and Cognition]*. Hogrefe, 1993.

[Holland et Reitman, 1978] J. H. Holland et J. S. Reitman. Cognitive Systems based on adaptive algorithms. *Pattern Directed Inference Systems*, 7(2) :125–149, 1978.

[Holland, 1976] J. H. Holland. Adaptation. *Progress in theorical biology*, 1976.

[Holland, 1990] J. H. Holland. Concerning the emergence of tag mediated lookahead in Classifier Systems. *Special Issue of Physica D*, 42 :188–201, 1990.

[Jaakkola *et al.*, 1995] T. Jaakkola, S. P. Singh, et M. I. Jordan. Monte-carlo reinforcement learning in non-markovian decision problems. In *Advances in Neural Information Processing Systems 7 : Proceedings of the 1994 Conference*, Denver, 1995. MIT Press.

[Josefowiez, 2001] J. Josefowiez. *Conditionnement opérant et problèmes decisionnels de Markov*. PhD thesis, Université de Lille 3, 2001.

[Kaelbing *et al.*, 1996] L. P. Kaelbing, M. L. Littman, et A. W. Moore. Reinforcement Learning : A Survey. *Journal of Artificial Intelligence Research*, number :237–285, 1996.

[Kamin, 1969] L. J. Kamin. Predictability, surprise, attention and conditioning. *Classical Conditioning*, pages 279–296, 1969.

[Kovacs, 1997] Tim Kovacs. XCS Classifier System Reliably Evolves Accurate, Complete, and Minimal Representations for Boolean Functions. In *Soft Computing in Engineering Design and Manufacturing*, éditeurs Roy, Chawdhry, et Pant, pages 59–68. Springer-Verlag, 1997.

[Koza *et al.*, 1998] éditeurs J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, Goldberg ; D. E., H. Iba, et R. Riolo. *Genetic Programming 1998 : Proceedings of the Third Annual Conference*. Morgan Kaufmann, 1998.

[Kremer, 1978] E. F. Kremer. The Rescorla-Wagner model : losses in associative strength in compound conditioning stimuli. *Journal of experimental psychology : animal behavior processes*, 4 :22–36, 1978.

[Landau *et al.*, 2002] S. Landau, S. Picault, O. Sigaud, et P. Gérard. A Comparison between ATNoSFERES and XCSM. In Langdon et al. (2002).

[Langdon *et al.*, 2002] éditeurs W. B. Langdon, E. Cantu-Paz, R. Poli, K. Mathias, G. Rudolph, D. Davis, et R. Roy. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO02)*. Morgan Kaufmann, 2002.

[Lanzi *et al.*, 2000] éditeurs P. L. Lanzi, W. Stolzmann, et S. W. Wilson. *Learning Classifier Systems. From Foundations to Applications*, volume 1813 de *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, 2000.

[Lanzi *et al.*, 2001] éditeurs P. L. Lanzi, W. Stolzmann, et S.W. Wilson. *Advances in Learning Classifier Systems*, volume 1996 de *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, 2001.

[Lanzi *et al.*, 2002] éditeurs P. L. Lanzi, W. Stolzmann, et S.W. Wilson. *Advances in Learning Classifier Systems*, volume 2321 de *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, 2002.

[Lanzi, 1998a] P. L. Lanzi. Adding Memory to XCS. In *proceedings of the IEEE Conference on Evolutionary Computation (ICEC98)*. IEEE Press, 1998.

[Lanzi, 1998b] P. L. Lanzi. An Analysis of the Memory Mechanism of XCSM. In Koza et al. (1998).

[Lanzi, 2000] P. L. Lanzi. Learning Classifier Systems from a Reinforcement Learning Perspective. Rapport technique, Dip. di Elettronica e Informazione, Politecnico di Milano, 2000.

[Littman *et al.*, 1995] M. L. Littman, A. R. Cassandra, et L. P. Kaelbling. Efficient Dynamic Programming Updates in Partially Observable Markov Decision Processes. Rapport Technique CS-95-19, Brown University, 1995.

[Maes *et al.*, 1996] éditeurs P. Maes, Mataric M. J., J. A. Meyer, J. Pollack, et Wilson S. W. *From Animals to Animats 4. Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior (SAB96)*. MIT Press, 1996.

[McCallum, 1995] R. . McCallum. *Reinforcement Learning with Selective Perception and Hidden State.* PhD thesis, University of Rochester, Rochester, NY, 1995.

[McCallum, 1996] R. A. McCallum. Learning to Use Selective Attention and Short–Term Memory. In Maes et al. (1996).

[Meyer *et al.*, 1992] éditeurs J. A. Meyer, H. L. Roitblat, et S. W. Wilson. *From Animals to Animats 2. Proceedings of the Second International Conference on Simulation of Adaptive Behavior (SAB92).* MIT Press, 1992.

[Meyer *et al.*, 2000] éditeurs J. A Meyer, A. Berthoz, D. Floreano, H. L. Roitblat, et S. W. Wilson. *From Animals to Animats 6 : Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior (SAB2000)*, 2000.

[Meyer et Wilson, 1990] éditeurs J. A. Meyer et S. W. Wilson. *From Animals to Animats 1. Proceedings of the First International Conference on Simulation of Adaptive Behavior (SAB90).* MIT Press, 1990.

[Mitchell, 1997] T. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[Pavlov, 1927] I. P. Pavlov. *Conditionned reflexes.* Oxford University Press, New York, 1927.

[Quinlan, 1993] R. Quinlan. *C4.5 : Programs for Machine Learning.* Morgan Kaufmann Publisher, 1993.

[Rao, 1970] R. Rao. *Advanced statistical methods in biometric research.* Hafner, 1970.

[Rescorla et Wagner, 1972] S. A. Rescorla et A. R. Wagner. A theory of pavlovian conditioning : variations in the effectiveness of reinforcement and nonreinforcement. *Classical Conditioning*, II :64–99, 1972.

[Riolo, 1988] Rick L. Riolo. CFS-C : A Package of Domain-Independent Subroutines for Implementing Classifier Systems in Arbitrary User-Defined Environments. Rapport technique, University of Michigan, 1988.

[Riolo, 1990] R. L. Riolo. Lookahead planning and latent learning in a Classifier System. In Meyer et Wilson (1990).

[Seward, 1949] J. P. Seward. An Experimental Analysis of Latent Learning. *Journal of Experimental Psychology*, 1949.

[Sigaud et Gérard, 1999] O. Sigaud et P. Gérard. Contribution au problème de la sélection de l'action en environnement partiellement observable. In Drogoul et Meyer (1999), pages 129–146.

[Sigaud et Gérard, 2001] O. Sigaud et P. Gérard. Being Reactive by Exchanging Roles : an Empirical Study. In Hannebauer et al. (2001).

[Skinner, 1938] B. F. Skinner. *The Behavior of organisms.* Appleon Century Croft, New York, 1938.

[Smith, 1980] S. F Smith. *A Learning System Based on Genetic Adaptive Algorithms.* PhD thesis, Department of Computer Science, University of Pittsburgh, 1980.

[Smith, 1994] R. E. Smith. Memory Exploitation in Learning Classifier Systems. *Evolutionary Computation*, 2(3) :199–220, 1994.

[Spector *et al.*, 2001] éditeurs L. Spector, Goodman E. D., A. Wu, W. B. Langdon, H. M. Voigt, et M. Gen. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO01)*. Morgan Kaufmann, 2001.

[Stolzmann, 1998] W. Stolzmann. Anticipatory classifier systems. In Koza et al. (1998).

[Stolzmann, 1999] W. Stolzmann. Latent learning in khepera robots with anticipatory classifier systems. In *Proceedings of the 1999 Genetic and Evolutionary Computation Conference Workshop Program*, 1999.

[Sun et Peterson, 2000] R. Sun et T. Peterson. Automatic partitioning for multi-agent reinforcement learning. In Meyer et al. (2000).

[Sutton et Barto, 1998] R. S. Sutton et A. G. Barto. *Reinforcement Learning : An Introduction*. MIT Press, 1998.

[Sutton, 1988] R. S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3 :9–44, 1988.

[Sutton, 1992] R.S. Sutton. Reinforcement Learning Architectures for Animats. In Meyer et al. (1992).

[Tolman et Honzik, 1930] E. C. Tolman et C. H. Honzik. Insight in Rats. *University of California Publications in Psychology*, 4 :215–232, 1930.

[Tolman, 1959] E. C. Tolman. Principles of purposive behavior. In *Psychology : A study of science*, éditeur S. Koch, pages 92–157, New York, 1959. Mc Graw-Hill.

[Tomlinson et Bull, 2000] A. Tomlinson et L. Bull. CXCS. In Lanzi et al. (2000), pages 194–208.

[Watkins, 1989] C. J. Watkins. *Learning with delayed rewards*. PhD thesis, Psychology Department, University of Cambridge, England, 1989.

[Whitely *et al.*, 2000] éditeurs D. Whitely, D. E. Goldberg, E. Cantú-Paz, L. Spector, Y. Parmee, et H. S. Beyer. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO2000)*. Morgan Kaufmann, 2000.

[Wiering et Schmidhuber, 1997] M. Wiering et J. Schmidhuber. Hq-learning. *Adaptive Behavior*, 6(2) :219–246, 1997.

[Wilson, 1985] Stewart W. Wilson. Knowledge Growth in an Artificial Animal. In Grefenstette (1985), pages 16–23.

[Wilson, 1994] S. W. Wilson. ZCS, a Zeroth level Classifier System. *Evolutionary Computation*, 2(1) :1–18, 1994.

[Wilson, 1995] S. W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2) :149–175, 1995.

[Wilson, 2001a] S. W. Wilson. Function approximation with a classifier system. In Spector et al. (2001), pages 974–981.

[Wilson, 2001b] S. W. Wilson. Mining oblique data with XCS. In Lanzi et al. (2001).

[Wilson, 2002] S. W. Wilson. Classifiers that approximate functions. *Natural Computing*, 2(1), 2002.