Un modèle de mémoire pour l'apprentissage de communication dans un SMA

Shirley Hoet, Nicolas Sabouret

Un modèle de mémoire pour l'apprentissage de communication dans un SMA

Shirley Hoet[a] Nicolas Sabouret[b]
shirley.hoet@lip6.fr nicolas.sabouret@limsi.fr

[a]Laboratoire d'Informatique de Paris 6,
Université Pierre et Marie Curie, France

[b]LIMSI-CNRS,
Université Paris-Sud France

Résumé

*L'apprentissage de comportement dans un contexte multi-agent est un problème difficile, en particulier parce que la prise en compte de la communication avec les autres agents requiert la mémorisation d'informations spécifiques. Dans cet article, nous présentons les modèles de mémoire existants et nous montrons qu'ils ne permettent pas de gérer l'apprentissage de communication. Nous proposons ensuite un modèle de mémoire pour l'apprentissage par renforcement des messages de commande (*request*) et de contrôle (*query*). Ce modèle permet de gérer l'asynchronisme du système et les attentes de réponses aux messages. Enfin, nous présentons une évaluation de ce modèle sur un exemple simple et nous montrons qu'il construit une politique en un temps raisonnable et avec un espace mémoire réduit.*

Mots-clés : *Mémoire, communication, apprentissage*

Abstract

Learning in a multi-agent context is a difficult task, especially since the communication with other agents require additional information to be stored and used in the learning process. In this paper, we propose a model of agent memory for command and control in a multi-agent context. We first show why the existing memory models are not sufficient to support automated learning of command and control in a multi-agent context. Then, we describe our model for storing messages and answers. Finally we show on an example that this model allows faster learning and better convergence.

Keywords: *Memory, Communication, Learning*

1 Introduction

Dans un système multi-agents, il est souvent nécessaire que les agents communiquent pour apprendre à agir [11, 14]. La communication directe (c'est-à-dire l'envoi de message avec intention de communiquer) permet ainsi aux agents de récupérer les informations qui leur manquent ou de se coordonner lorsque cela est nécessaire. Tout particulièrement les messages de type commande (*request*) qui permettent de déléguer des actions et les messages de type contrôle (*query*) qui permettent de demander des informations aux autres agents. Nous nous intéressons particulièrement à ces messages qui sont au cœur de la communication SMA [3].

Dans le cas où les agents sont autonomes, des travaux ont montré que les agents devaient non seulement communiquer pour apprendre à agir mais aussi apprendre à communiquer [8, 5]. En effet, les agents doivent apprendre dans quel contexte il est nécessaire de communiquer et surtout quels messages ils peuvent envoyer pour résoudre leurs problèmes. Toutefois, les travaux du domaine font l'hypothèse que les agents ont des connaissances partagées sur le problème à résoudre (système fortement couplé) ou qu'ils s'exécutent de manière synchrone [8, 5]. Or dans le cas de systèmes réels tel que [1], les agents évoluent généralement dans un système asynchrone, faiblement couplé et partiellement observable. De plus ils peuvent être hétérogènes dans le sens où ils ne partagent pas les mêmes fonctionnalités. Dans ce contexte, apprendre à communiquer devient un problème difficile à résoudre.

Premièrement, les agents n'ont pas d'informations sur les capacités et les connaissances des autres agents : ils ne peuvent prévoir les effets des messages qu'ils envoient sur les agents qui les reçoivent. De ce fait, il n'est pas possible d'utiliser des outils de planification pour déterminer les messages à utiliser et quand les envoyer pour résoudre un problème donné. L'agent doit apprendre par essai-erreur, par exemple en utilisant de l'apprentissage par ren-

forcement pour découvrir quand et comment utiliser ses messages.

Deuxièmement, lors de la délégation de tâche (*request*), chaque commande peut avoir un temps d'exécution différent qui ne peut pas être anticipé par l'agent demandeur. L'agent doit donc apprendre à déterminer quand une commande est terminée et quand il peut entamer les actions qui dépendent de cette terminaison. Les agents doivent donc apprendre à attendre en plus d'apprendre à communiquer.

Troisièmement, l'environnement étant partiellement observable et les agents n'ayant pas de connaissances sur les autres agents, ils ne peuvent pas prédire quelles sont les informations qui leur manquent.

Enfin, l'envoi de messages entre agents peut être une opération coûteuse, par exemple dans les réseaux mobiles ad-hoc. Les agents doivent donc apprendre à utiliser la communication avec parcimonie.

Pour gérer l'ensemble de ces contraintes, nous avons proposé dans [4] d'utiliser des algorithmes d'apprentissage par renforcement spécifiques pour mettre à jour leurs croyances. Nous avons aussi montré qu'il était nécessaire de mémoriser les communications comme un élément de l'état de l'agent pour l'utiliser dans l'algorithme d'apprentissage.

C'est pourquoi, dans cet article, nous présentons un modèle de mémoire pour l'apprentissage de communication autour des messages de commande (*request*) et de contrôle (*query*). La section suivante présente les modèles de mémoire existants et leurs limites. La section 2 présente notre modèle d'agents et notre mécanisme de mémorisation des messages. La section 3 présente une évaluation de notre modèle et montre que nos agents sont capables d'apprendre des politiques plus rapidement que d'autres algorithmes de gestion de la mémoire. Enfin, nous discutons les perspectives de nos travaux dans la section 4.

1.1 Travaux existants

Il existe trois types de modèle de mémoire dans la littérature. Ces modèles se différencient par le type d'informations qu'ils mémorisent ainsi que par leur mécanisme de mise-à-jour.

Les mémoires temporelles stockent les N dernières observations et actions de l'agent, avec N la taille maximale de la mémoire. La taille de la mémoire nécessaire pour effectuer l'apprentissage est déterminée en utilisant un algorithme incrémental qui permet d'augmenter de 1 en 1 cette taille durant l'apprentissage. Cette taille pouvant être très grande et donc augmenter de façon conséquente l'espace d'état de l'agent, les travaux de [2, 7] se sont intéressés à déterminer de manière heuristique la meilleure taille de mémoire pour chaque état initial de l'agent. Cependant, ces modèles requièrent que l'agent mémorise les N dernières informations pour pouvoir prendre en compte une action effectuée il y a N pas de temps, même si seule cette action est pertinente. Cela conduit à un espace d'état à explorer k^{N-1} fois plus grand, lorsque k est le nombre d'actions possibles pour chacune des $N-1$ actions non pertinentes.

La mémoire de travail [12, 6] est une mémoire composées d'un nombre de cases prédéfinies à l'avance. Une case peut stocker une action ou une observation passée de l'agent [12] ou un encore un bit [6]. Pour déterminer ce que la mémoire doit contenir, les auteurs proposent d'utiliser l'apprentissage par renforcement. Ils définissent ainsi des actions pour chaque case permettant d'effacer le contenu de la case, de le remplacer par une valeur donnée ou de la maintenir. Durant l'apprentissage de ses actions, l'agent effectuera simultanément autant d'apprentissage pour déterminer comment modifier sa mémoire qu'il existe de case dans sa mémoire. La limite principale de ce modèle est la taille de la mémoire qui est définie *a priori*. Elle ne doit pas être trop petite afin de stocker les informations nécessaires à l'apprentissage de l'agent, mais elle ne doit pas non plus être trop grande afin de limiter le nombre d'états créés durant l'apprentissage.

La mémoire cognitive [9, 15] est un modèle de mémoire complexe basé sur l'utilisation d'une mémoire épisodique couplée à une mémoire de travail. La mémoire épisodique stocke l'ensemble des expériences de l'agent c'est un historique complet des actions et des observations de l'agent. Du fait que la mémoire épisodique est potentiellement de taille infinie, elle ne peut être utilisée telle quelle pour l'apprentissage. L'agent doit extraire de cette mémoire les informations nécessaires et les stocker dans sa mémoire de travail qui est utilisée pour l'apprentissage. La limite de ce modèle est qu'il n'existe pas de mécanisme d'extraction utilisable sur des problèmes réels.

Chaque structure de mémoire présente des avantages et des inconvénients dans le contexte de

l'apprentissage de communication. C'est pourquoi, dans cet article, nous proposons un autre modèle, fondé sur une combinaison de la mémoire de travail et de la mémoire temporelle.

2 Stockage des messages

2.1 Nomenclature

Dans cet article, nous notons $\mathbb{A}$ l'ensemble des agents et $Agt_l \in \mathbb{A}$ l'agent apprenant. Un agent quelconque $Agt \in \mathbb{A}$ est décrit à un instant t par son état $s_t^{Agt} \in S^{(Agt)}$ où $S^{(Agt)}$ est l'ensemble des états possibles de l'agent. L'état s_t^{Agt} d'un agent est constitué :

– de l'ensemble des observations de l'agent $\Omega_t^{(Agt)}$. Une observation est un couple $\langle var, val \rangle$ où var est une variable décrivant un élément de l'environnement et val la valeur observée par l'agent à l'instant t ;
– de l'ensemble des données internes de l'agent $\mathbb{D}_t^{(Agt)}$. Une donnée est un couple $\langle var, val \rangle$ où var est une variable décrivant l'état interne de l'agent et val sa valeur à l' instant t ;
– de la mémoire de l'agent $m_t^{(Agt)}$. Cette structure est décrite en détails dans la section 2.3.

Un agent $Agt \in \mathbb{A}$ possède un ensemble d'action $A^{(Agt)}$ qui est constitué :

– des actions propres de l'agent, c.-à-d. les actions que l'agent peut exécuter pour modifier soit des variables de l'environnement soit ses données propres ;
– de messages que l'agent peut envoyer aux autres agents de l'environnement.

Un message est décrit selon la nomenclature FIPA [1] par un n-uplet $\langle em, p, c, dst \rangle$ où :

– $em, dst \in \mathbb{A}$ sont respectivement l'émetteur et le destinataire du message ;
– p est le performatif du message ;
– c est le contenu du message.

Un message est considéré comme une action par nos agents. Parce que nous nous appuyons sur un algorithme d'apprentissage par renforcement (*cf.* section 2.5), nos agents ont besoin de déterminer la fin de chaque action, y compris les actions de communication. Dans cet article, nous faisons le choix qu'une action de communication est terminée soit à la réception du message réponse, lorsque le message envoyé en nécessite une, soit à la fin de l'envoi du message lui-même s'il ne nécessite pas de réponse. Pour déterminer si un message nécessite ou non une réponse, nous utilisons les protocoles définis par FIPA. Ainsi l'envoi d'un message de performatif *request* nécessite une réponse de performatif *impossible*, *not-understood* ou *agree*. Les messages *impossible*, *not-understood* ou *agree* eux ne nécessitent pas de réponse.

Un agent Agt a un comportement cyclique. À chaque cycle d'exécution, l'agent :

1. observe l'environnement et met à jour son ensemble d'observation $\Omega^{(Agt)}$;
2. traite l'ensemble des messages qu'il a reçu depuis son dernier cycle d'exécution (le traitement consiste à répondre aux messages le nécessitant) ;
3. détermine les actions $A' \in A^{(Agt)}$ qu'il peut effectuer dans l'état s_t ;
4. exécute l'ensemble des actions A'.

Un agent Agt peut donc exécuter plusieurs actions par cycle par conséquent, la durée d'un cycle n'est pas la même pour tous les agents. Cependant du fait que cette durée dépend de l'état des agents (qui détermine les actions que l'agent effectue), nous considérons qu'elle est constante pour un état $s^{(Agt)}$ donné de l'agent.

Dans le cas de l'agent apprenant Agt_l, une seule action est exécutée par cycle d'exécution comme le nécessite les algorithmes d'apprentissage par renforcement. De plus cette action sera choisie à l'aide de l'algorithme d'apprentissage contrairement aux autres agents. Enfin, un cycle d'exécution correspond à un pas de temps d'apprentissage. Ce pas de temps est local à l'agent apprenant est ne correspond pas au pas de temps d'un autre agent du système. En effet, chaque agent a une durée d'éxecution qui lui est propre.

L'agent apprenant Agt_l possède aussi une action supplémentaire : l'action *wait*. Cette action est particulière car elle n'a pas d'effets que ce soit sur l'environnement ou sur l'agent apprenant. Lorsque l'agent choisit d'exécuter l'action *wait* durant un cycle d'exécution, il n'effectue ni action propre ni communication. Ainsi cette action permet à l'agent d'apprendre à ne rien faire. Cette action est nécessaire pour que l'agent est la possibilité d'apprendre à attendre les effets des messages qui le nécessitent. En effet, la fin d'un message du point de vue de l'agent est déterminée par l'arrivée d'un message réponse. Or dans le cas des commandes (envoi d'un message *request*), la réception d'un message *agree* spécifiant que l'agent a accepté de recevoir la commande ne signifie pas que l'action demandée a été effectivement exécutée. Or dans certains cas l'agent a besoin que les effets de la pré-

1. http://www.fipa.org/

cédente commande soient avérés pour pouvoir effectuer une autre commande ou exécuter une action. Dans ce cas l'agent a donc besoin d'attendre. De plus l'agent ne connaissant pas les effets de ses commandes, il ne sait pas combien de temps il doit attendre. Il est donc nécessaire que l'agent apprenne à attendre.

2.2 Besoin

Pour apprendre à communiquer et plus exactement apprendre à commander via un message *request* et contrôler via un message *query*, un agent a besoin de mémoriser un certain nombre d'éléments.

Premièrement, l'agent doit pouvoir mémoriser les commandes précédemment envoyées. En effet, le système étant asynchrone les effets d'une commande peuvent être différés du point de vue de l'agent apprenant. Or le système étant faiblement couplé, l'agent ne connaît pas ses effets. L'agent ne peut donc pas tout simplement attendre d'observer les effets de sa précédente commande car il ne les connaît pas avant d'en envoyer une nouvelle. Par conséquent, l'agent peut envoyer une commande avant que la précédente soit terminée et ainsi observer au même instant des effets dus à différentes commandes. C'est pourquoi il est nécessaire que l'agent mémorise les commandes qu'il a précédemment envoyée. Cependant certaines commandes peuvent ne pas aboutir, c'est le cas par exemple lorsque l'agent qui a reçu la commande refuse de l'exécuter. Dans ce cas il n'est pas nécessaire de mémoriser la commande envoyée. D'autre part, un agent peut accepter d'effectuer une commande de différentes manières. Il peut accepter de le faire maintenant ou plus tard ou encore il peut demander à l'agent de lui "rappeler" la commande à effectuer dans un certain nombre de pas de temps. Mémoriser simplement la commande envoyée fera perdre cette information, c'est pourquoi nous pensons qu'il est nécessaire de mémoriser les réponses des commandes plutôt que les commandes elles-même et de ne mémoriser ces réponses seulement quand celles-ci sont du type "acceptation". Pour cela nous faisons l'hypothèse que l'agent possède une fonction $rew(p)$ où p est le performatif de la réponse reçue qui renvoie une valeur positive si le message de performatif p est considéré comme positif pour l'agent et une valeur négative dans le cas contraire.

Cependant mémoriser les réponses n'est pas suffisant. En effet que ce soit des réponses relatives à des commandes ou à de la demande d'information, l'agent peut avoir parfois besoin de savoir quand il a reçu cette réponse. Ainsi, si un agent cuisinier délègue à un autre une action comme *allumer la plaque*, l'agent a besoin de mémoriser quand cette commande a eu lieu. Ceci lui permettra en autre de savoir quand son diner est prêt et aussi de ne pas le brûler. De même la date est nécessaire pour les informations stockées par l'agent. Un agent a besoin généralement de mémoriser depuis combien de temps il détient une information afin de déterminer si celle-ci est toujours vrai et par conséquent s'il a besoin de renvoyer un message permettant de la mettre à jour.

D'autre part, il peut-être plus compacte de mémoriser qu'un agent n' a pas encore envoyé un message m. Imaginons que dans un état s l'agent doit envoyer le message m s'il ne l'a pas encore fait et sinon il doit envoyer un message m'. si l'agent ne mémorise que les messages qu'il a envoyé dans le passé alors il va avoir besoin de beaucoup de mémoire pour agir correctement dans l'état s. En effet avec x crans de mémoire, il sait s'il a ou non effectué m il y a x pas de temps, mais il ne sait pas s'il a envoyé le message m il y a plus de x pas de temps. Il serait donc plus compact de mémoriser les messages que l'agent n'a pas envoyé.

Enfin, il est important qu'un agent puisse mémoriser les phases où il n'envoie pas de commande et est en train d'attendre en utilisant l'action *wait*. Plus exactement, il doit mémoriser les périodes d'attentes afin d'apprendre s'il doit encore attendre ou s'il peut envoyer une commande. Cependant la mémorisation de cette période d'attente n'est pertinente que dans les cas où l'agent est effectivement en train d'attendre. Dans les autres cas, c'est-à-dire mémoriser que l'agent a attendu puis envoyer la commande B puis la commande C, la période d'attente n'est pas utile pour l'apprentissage de l'agent.

Pour résumer, l'agent a besoin de mémoriser les réponses "positives" des messages qu'il a envoyé et la date à laquelle il les a envoyé, les messages qu'il n'a pas encore envoyés ainsi que ses périodes d'attentes.

2.3 Notre modèle de mémoire

La structure de notre mémoire est décrite à un instant t par un (n+1)-uplet $m_t = \langle m_t^1, ..., m_t^n, w \rangle$ où :
– n est le nombre total de messages que peut envoyer l'agent ;

– $\forall k \in [1,n], m_t^k = \langle delai, answer \rangle$ tel que $answer = p(c)$ contienne la dernière réponse positive ($rew(p) > 0$) au k-ème message de l'agent apprenant. $delai \in [0, Tm] \cup \infty$ correspond au nombre de pas de temps écoulés depuis que la réponse a été reçue, où ∞ s'il est supérieur à la limite Tm. $delai = 0$ signifie que l'agent n'a encore jamais envoyé de k-ème message ;
– $w \in \mathbb{N}^+$ stockant les durées d'attente de l'agent.

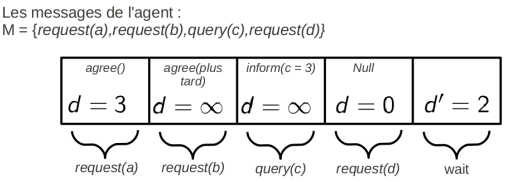

FIGURE 1 – Un exemple de la mémoire d'un agent Agt_l tel que A^{Agt_l} = {*request(a)*, *request(b)*, *query(c)*, *request(d)*, *wait*} et utilisant une valeur de $Tm = 3$. Dans cet exemple, l'agent est en train d'attendre depuis 2 pas de temps. Il a reçu un message *agree()* il y a trois pas de temps suite à l'envoi d'un *request(a)*. Il a reçu deux messages *agree(with_delai)* et *query(c=3)* il y a plus de trois pas de temps. Et il n'a jamais envoyé de messages *request(d)*.

Notre modèle de mémoire permet donc à la fois de stocker les réponses de notre agent et les dates via la variable $delai$ comme l'illustre la figure 1. Lorsque l'agent n'a pas effectué une action alors sa variable $delai$ est à 0. Ainsi, l'agent peut vérifier rapidement qu'il n'a pas encore effectué l'action.

Pour limiter le nombre de configurations possibles pour la mémoire, et donc faciliter la convergence de l'apprentissage, nous avons choisi de borner le $delai$ par une valeur Tm. La valeur ∞ permet alors à l'agent de mémoriser qu'il a reçu une réponse, même si la date exacte n'est pas précisée.

2.4 Mise-à-jour de la mémoire

La mémoire de l'agent est mise à jour à chaque cycle d'apprentissage. Cette mise à jour est constituée de trois étapes.

La première va consister à incrémenter de 1 la valeur de la variable $delai$ de chaque case m_t^k. Cette action permet donc à l'agent de mémoriser le délai exact écoulé depuis le dernier envoi de chaque message m^k. Dans le cas où la valeur de délai dépasse Tm, elle est remplacé par le symbole ∞ signifiant « j'ai reçu cette réponse il y a longtemps ».

La deuxième consiste à modifier la case w. Si l'agent vient d'effectuer une action $wait$ alors la valeur contenue par w est augmentée de 1. Sinon, l'agent n'est pas dans une période d'attente par conséquent la case w est remise à 0.

La troisième étape consiste à mémoriser la dernière réponse reçue par l'agent, lorsque la dernière action effectuée par l'agent est l'envoi d'un message m et que la réponse reçue $r = \langle p, c \rangle$ est telle que $rew(p) > 0$. Alors le contenu de la case m_t^k correspondant au message m prendra la valeur $\langle r, 1 \rangle$.

Si n est le nombre de messages que l'agent peut envoyer, la mise-à-jour consistera en $\theta(n)$ opérations. Cependant comme nous le verrons, l'algorithme d'apprentissage nécessite de stocker et d'apprendre $\theta(n*(Tm+1)^n)$ valeurs correspondant à l'espérance de récompense obtenue par l'agent pour chaque couple (s, a) où $s \in S^{(Agt_l)}$ est un état et $a \in A^{(Agt^l)}$ une action. Par conséquent, le stockage de la variable délai et l'opération de mise-à-jour sont négligeables en terme de complexité dans ce processus.

2.5 Algorithme d'apprentissage

Notre algorithme d'apprentissage a pour but de déterminer une politique d'action et de communication en utilisant le modèle de mémoire que nous avons détaillé à la section 2.3. Dans cette section nous avons montré que le nombre de configurations possibles de notre mémoire dépendait de la valeur de la variable Tm qui borne la valeur maximale que peut prendre la variable $delai$. La difficulté dans ce cas est de déterminer la meilleure valeur de la variable Tm permettent à l'agent d'apprendre à agir et à communiquer. C'est pourquoi nous proposons un algorithme d'apprentissage incrémental qui va progressivement augmenter la valeur de Tm et effectuer pour chaque valeur de Tm un cycle d'apprentissage par renforcement. Notre algorithme est décrit brièvement par la figure 1. Dans la suite de cette section, nous expliquons plus en détails les quatre points importants de notre algorithme :
– l'utilisation de l'algorithme du Q-Learning pour apprendre à communiquer (représenté par la fonction Ql dans l'algorithme 1) ;
– l'utilisation d'un ensemble d'états restreint de l'agent sur lequel appliqué le Q-Learning(noté S_l) ;
– l'utilisation d'une heuristique pour construire l'ensemble d'états S_l (représenté par la fonction $detection$) ;
– l'arrêt de notre algorithme.

Algorithme 1 Algorithme incrémental utilisant notre modèle de mémoire pour l'apprentissage de la communication.

Entrées: $\gamma, \epsilon, N_e, T_e, E, T, T_d, N$
$S_l \leftarrow \emptyset$
$Tm \leftarrow 1$
$R_1, Q, \Delta Q, \nu, S_l \leftarrow Ql(Tm, \gamma, N_e, T_e, E, T, T_d, S_l)$
$S_l^* \leftarrow detection(S_l, Q, \Delta Q, \nu, N)$
$R_0 \leftarrow 0$
tant que $|R_1 - R_0| < \epsilon$ **faire**
$R_1 \leftarrow R_0$
$R_1, Q, \Delta Q, \nu, S_l \leftarrow Ql(Tm, \gamma, N_e, T_e, E, T, T_d, S_l)$
$S_l^* \leftarrow detection(S_l, Q, \Delta Q, \nu, N)$
$Tm \leftarrow Tm + 1$
fin tant que
pour $s \in S_l$ **faire**
$\pi(s) \leftarrow \arg\max_{a \in A} Q(s, a)$
fin pour
Sorties: π

Q-Learning. Le Q-Learning [13] est un algorithme d'apprentissage par renforcement qui repose sur l'utilisation d'une fonction de valeur $Q : E * A \longrightarrow \mathbf{R}$ où $Q(e, a)$ correspond à la récompense attendue lorsque l'agent exécute l'action a lorsqu'il se trouve dans l'état e. La meilleure action pour chaque état est alors définie par : $a^* = \arg\max_{a \in A} Q(s, a)$.

Le principe du Q-Learning est de construire itérativement la fonction Q pour chaque couple (e, a) en expérimentant les actions dans l'environnement. A chaque prise de décision (c'est à dire choix d'une action à effectuer) l'agent choisit une action a dans A_e, exécute a, reçoit une récompense $r(e, a)$ et observe son nouvel état e'. Il met alors à jour la valeur Q(e,a) en fonction de e' et de $r(e, a)$ selon la formule suivante :

$$Q(e, a) = (1-\alpha_t)Q(e, a)+\alpha_t(r+\gamma \max_{a \in A} Q(e', a'))$$

où $\alpha_t \in [0, 1]$ est le taux d'apprentissage et γ est le facteur d'actualisation (il permet de modéliser les préférences de l'agent en matière de récompense. Si l'agent préfère une récompense immédiate γ sera proche de 0, si au contraire toutes les récompenses sont importantes pour lui γ tendra vers 1).

La stratégie de sélection utilisée pour l'action à exécuter est la température de Boltzman où la probabilité de sélectionner l'action a dans l'état e est égale à :

$$P(a_t = a|e_t = e) = \frac{e^{Q(e,a)/T_t}}{\sum_{b \in A} e^{Q(e,b)/T_t}}$$

Avec T_t le paramètre de température qui décroît lentement en fonction du temps t. Lorsque la température au départ est très élevée alors la probabilité de choisir une action a ne dépend pas de $Q(e, a)$ et est uniforme. Au contraire lorsque la température est plus faible et proche de 0, la probabilité de choisir l'action a dépend de la valeur de $Q(e, a)$, ainsi une action avec un Q élevé a alors plus de chance d'être choisie.

Un ensemble d'états restreints. L'algorithme du Q-Learning nécessite de construire la fonction de Q-Value pour l'ensemble des couples (état,action) de l'agent. Dans, la cas où l'ensemble d'actions soit constitué de n messages et l'état de l'agent soit décrit uniquement par une mémoire m alors le Q-Learning requiera de stocker et d'apprendre $\theta(n * (Tm + 1)^n)$ valeurs pour la fonction Q(s,a) ce qui peut s'avérer très grand. Comme nous ne pouvons pas limiter le nombre de messages que l'agent peut envoyer il est impératif de limiter la valeur de Tm tout en essayant d'obtenir une valeur de Tm suffisament grande pour permettre à l'agent d'apprendre à communiquer. Pour cela nous proposons d'utiliser un algorithme incrémental qui va augmenter de manière progressive la valeur de Tm durant l'apprentissage. Néanmoins, si l'agent doit mémoriser qu'il a fait A il y a 4 pas de temps lorsqu'il observe o_1, alors Tm doit être égale au moins à 4. Ainsi, pour l'observation o_2, l'agent mémorisera aussi tout ce qu'il a fait depuis 4 pas de temps même si cela n'est pas nécessaire. C'est pourquoi, nous proposons de n'augmenter Tm que dans les états qui le nécessitent. Pour ce faire, nous n'appliquons pas l'algorithme d'apprentissage sur l'ensemble d'états $S^{(Agt_l)}$ de l'agent Agt_l mais sur un ensemble plus restreint que nous noterons S_l.

Pour construire S_l, nous avons besoin de définir la relation de généralisation entre deux configurations de mémoire. Soit deux configurations de mémoire d'un agent $m_1 = \langle m_1^1, ..., m_1^n, w_1 \rangle$ et $m_2 = \langle m_2^1, ..., m_2^n, w_2 \rangle$. Nous considérons que m_1 généralise m_2 si et seulement si :
– $w_1 = w_2$;
– $\forall i \in [1, n], m_1^i.answer = m_2^i.answer$;
– $\forall i \in [1, n], m_1^i.delai \neq \infty, m_1^i.delai = m_2^i.delai$

Par extension, nous définissons la relation de généralisation entre deux états. Soit deux états $s_1 = \langle \Omega_1, \mathbb{D}_1, m_1 \rangle$ et $s_2 = \langle \Omega_2, \mathbb{D}_2, m_2 \rangle$, s_1 généralise s_2 si et seulement si :
– $\Omega_1 = \Omega_2$
– $\mathbb{D}_1 = \mathbb{D}_2$

– m_1 généralise m_2

La construction de S_l se fait de manière incrémentale :

à T = 0, $Tm = 1$ **et** $S_l = \emptyset$ L'algorithme effectue un cycle d'apprentissage. Durant l'apprentissage, chaque nouvel état $s_t \notin S_l$ est automatiquement ajouté à l'ensemble S_l et l'agent effectue donc son apprentissage sur l'ensemble $S^{(Agt_l)}$. Une fois l'apprentissage terminé, l'algorithme détecte l'ensemble des états $S^*_{app} \subset S_l$ qui ont besoin d'un Tm plus grand. Pour cela l'algorithme utilise la notion d'ambiguïté telle que définie par [2] que nous présentons ci-dessous. Si un état est considéré comme ambigu alors il est ajouté à l'ensemble S^*_{app} et il est supprimé de l'ensemble S_l ;

à T = 1, $Tm = Tm + 1$. L'algorithme effectue un nouvel apprentissage. À chaque nouvel état rencontré $s_t \notin S_l$, l'algorithme détermine l'état $s_l \in S_l$ qui généralise s_t. Si s_l existe alors l'agent apprend à agir en fonction de s_l. Sinon, l'état s_t est ajouté à l'ensemble S_l. L'agent effectue donc son apprentissage à la fois sur des états où $Tm = 1$ et des états où Tm = 2. Une fois l'algorithme d'apprentissage terminé, l'ensemble des états $S^*_{app} \subset S_l$ est de nouveau détecté et supprimé de l'ensemble S_l ;

à T=N, $Tm = N + 1$. L'algorithme effectue de nouveau un cycle d'apprentissage et continue à augmenter l'ensemble d'états S_l en augmentant la valeur de Tm dans les états qui le nécessitent. Cet ensemble d'états contient alors des états s_l ayant une variable $delai = Tm$ mais aussi des états où la variable $delai$ la plus grande < Tm.

Detection des états ambigus. La détection des états ambigus reposent sur les travaux de [2]. Dans ces travaux les auteurs proposent une heuristique, pour déterminer si un état s_l a besoin de mémoire, fondée sur trois critères :

une convergence plus lente, i.e. la valeur de la fonction de Q-Value $Q(s_l, a)$ converge plus lentement pour toutes les actions a ;

le nombre de mises à jour, i.e. un élément a plus de chance d'être ambigu si son nombre de mises à jour est grand ;

l'ambiguïté dans les actions, i.e. l'action optimale d'un état n'est pas clairement définie car les valeurs de Q des deux meilleures actions sont très proches.

L'ambiguïté d'un état s se définit donc de la manière suivante : $amb(s) = \frac{1}{3}(rg_s[up(s)] + rg_s[\frac{1}{Z}\sum_{a \in A_s} \Delta q_a] + rgD_s[q_{a1} - q_{a2}])$ où :

– a_1 et a_2 représentent les deux meilleures actions dans l'état s (elles maximisent la valeur Q(s,a)) ;
– q_a = Q(s,a) et Δq_a est la dernière modification apportée à q_a ;
– Z est le nombre de valeur Q(s,a) de l'état s ;
– $rg_s(x)$ (respectivement $rgD_s(x)$) renvoie la position de l'état s si on ordonne tous les états dans l'ordre croissant(respectivement décroissant) en fonction de x.

Cette fonction établit donc le taux d'ambiguïté d'un état s_l. Une fois ce taux calculé pour chaque élément de l'ensemble S_l, il est alors possible de déterminer les N [2] plus ambigus.

Arrêt de l'algorithme. Comme nous l'avons expliquer précédemment, pour déterminer la meilleure taille de Tm nous utilisons un algorithme incrémental qui augmente la valeur de Tm de 1 en 1. À chaque incrémentation de la valeur de Tm, un cycle d'apprentissage est effectué par l'agent. Ce cycle consiste à effectuer l'algorithme du Q-Learning durant N_e expériences. Comme Tm peut-être insuffisante pour apprendre au début, nous limitons à T_e le nombre de cycles d'exécution que peut effectuer l'agent durant une expérience. Lorsque l'agent a effectué T_e cycles d'exécution, l'expérience est arrêtée et l'agent reçoit une récompense négative. Si ce cas se reproduit E fois, alors nous considérons que la valeur de Tm est insuffisante et qu'il n'est donc pas nécessaire de continuer l'apprentissage. Nous stoppons l'apprentissage de l'agent et augmentons Tm. Ainsi nous limitons la durée des cycles d'apprentissage où l'agent n'arrive pas à apprendre dans le but d'augmenter la vitesse de convergence de notre algorithme vers une solution.

3 Évaluation

Nous avons testé notre modèle de mémoire sur un problème d'apprentissage mono-agent dans un SMA asynchrone, faiblement couplé et partiellement observable. Nous présentons dans une première partie les caractéristiques du problème étudiés. Puis nous présentons nos résultats.

2. où N est déterminé expérimentalement

3.1 Problème

Dans cet exemple, le système est composé de trois agents : l'agent apprenant L, l'agent A et l'agent B. L'agent A possède deux actions : l'action *creerX* et l'action *charger_batterie*. L'action *charger_batterie* est un processus que l'agent peut exécuter s'il n'est pas en train d'effectuer une autre action. L'agent A a une probabilité $p = 1/2$ de choisir cette action. L'action *creerX* est une commande par conséquent l'agent A l'exécute seulement lorsqu'il reçoit un message *request(creerX)* et à condition qu'il n'y a pas de ressource X déjà présente dans l'environnement. Lorsque l'agent reçoit un message *request(creerX)*, il peut y répondre de trois manières différentes en fonction de son état et de l'action qu'il est potentiellement en train d'exécuter :

– via un message *agree(creerX)* lorsqu'il n'y a pas de ressources X dans l'environnement et qu'il n'est pas en train d'exécuter une autre action ;
– via un message *agree(creerX ;plus tard)* lorsqu'il n'y a pas de ressources X dans l'environnement et qu'il est en train d'exécuter l'action *charger_batterie* ;
– via un message *impossible(creerX)* lorsqu'il est déjà en train d'effectuer l'action *creerX* ou que la ressource X est déjà présente dans l'environnement.

L'agent B possède une seule action : l'action *creerY* qui permet de transformer une ressource X en ressource Y. L'agent B peut exécuter l'action *creerY* si et seulement si il reçoit une commande *request(creerY)* et qu'il y a une ressource X présent dans l'environnement. L'agent apprenant possède trois actions : les commandes *request(creerX)*, *request(creerY)* et *wait*, dont il ne connait pas les effets. Il évolue dans un environnement partiellement observable car il ne peut observer la présence de la ressource X dans l'environnement. Enfin, le système est asynchrone car chaque action peut avoir une durée d'exécution différente. Pour notre évaluation nous avons utilisé les durées suivantes :

– $d_{request(creerX)} = d_{request(creerY)} = d_{wait}$;
– $d_{charger\_batterie} = 2 * d_{wait}$;
– $d_{creerX} = 4 * d_{wait}$;
– $d_{creerY} = 3 * d_{wait}$.

Par conséquent du point de vue de l'agent apprenant, la commande *request(creerX)* peut se terminer 3 à 5 pas de temps après l'avoir envoyé et traitée la réponse. De même la commande *request(creerY)* se terminera 2 pas de temps après avoir l'avoir exécutée. La difficulté pour l'agent apprenant est donc d'apprendre quand envoyer la commande *request(creerY)* après l'envoi d'une commande *request(creerX)* en fonction des réponses obtenues et de ses temps d'attentes.

3.2 Résultats

Nous avons implémenté notre modèle sur la plateforme multi-agent VDL [10]. Nous avons aussi implémenté le modèle de mémoire temporelle proposé par [2]. Le but est de comparer les résultats obtenus sur l'exemple des ressources d'un agent apprenant utilisant notre modèle de mémoire et ceux obtenus par un agent utilisant une mémoire temporelle. Nous n'avons pas utilisé la mémoire de travail comme outil de comparaison car comme nous l'avons montré la taille de la mémoire doit être choisi arbitrairement et de plus nécessite d'effectuer plusieurs algorithme d'apprentissage en parallèle rendant la comparaison innapropriée. De même, la mémoire cognitive n'est pas exploitable car il n'existe pas de mécanisme permettant de déterminer ce qu'il faut récupérer dans l'historique de l'agent.

Notre modèle de mémoire utilise les paramètres suivants : $Tm = 1$ et nous avons défini la fonction $rew(p)$ de la manière suivante :

– rew(*agree*) = 0 ;
– rew(*impossible*) = -5 ;

Pour la mémoire temporelle, nous stockons les actions, c'est-à-dire les commandes ou l'action *wait*, effectuée par l'agent. La taille de la mémoire de l'agent est initialisée à 1.

Dans les deux cas, nous utilisons notre algorithme incrémental pour augmenter la taille du paramètre Tm de notre modèle ou la taille de la mémoire temporelle. Les paramètres d'initialisation de l'algorithme d'apprentissage utilisés pour obtenir nos résultats sont les suivants : $\gamma = 0.9$, $T = 5$, $T_d = 0.99$, $\epsilon = 0.001$, $N_e = 1000$, $T_e = 1000$, $E = 10, N = 4$.

La fonction de récompense R ($R : S \rightarrow \mathbb{R}$) de l'agent est définie comme la somme de deux fonctions r_1 et r_2 où

– r_1 vaut 10 si l'agent obtient la ressource Y et 0 sinon ;
– r_2 dépend de la réponse obtenue à l'envoi d'une commande. L'agent obtient la valeur 0 si c'est un message *agree* et -5 sinon ;

Nous considérons la somme totale des récompense obtenues par l'agent durant une expérience en fonction du nombre d'itération de l'algorithme d'apprentissage. La figure 2 représente la moyenne de la récompense obtenue

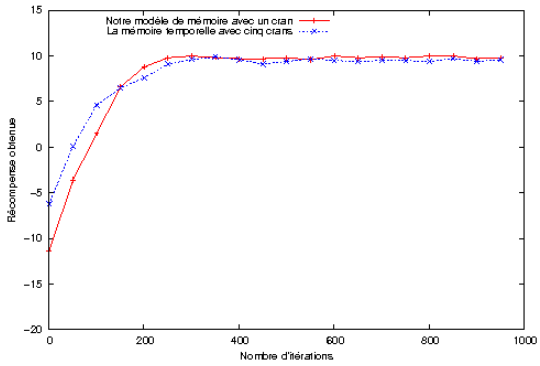

FIGURE 2 – Récompenses moyennes obtenues par l'agent durant un cycle d'apprentissage en utilisant soit notre modèle et 1 cran de mémoire (+) soit la mémoire temporelle et 5 crans de mémoire (x)

par l'agent durant 50 expériences. Comme le montre ce graphique, l'agent en utilisant notre modèle et un cran de mémoire récupère à la fin de l'apprentissage une récompense moyenne de 10. La mémoire temporelle nécessite par contre cinq crans de mémoire pour que l'agent obtienne une récompense moyenne de 10. Avec moins de cinq crans de mémoire, l'algorithme ne converge pas vers une solution. En effet, l'agent apprend dans ces cas là à attendre au lieu d'envoyer des messages, ce qui bloque l'agent dans une phase d'attente et nous oblige donc à arrêter l'expérience. Par conséquent, notre modèle permet donc d'apprendre en utilisant moins de mémoire et en effectuant donc moins de cycles d'apprentissage.

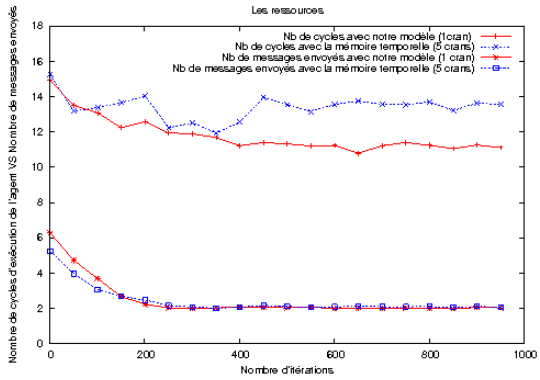

FIGURE 3 – Nombres d'action (ou de messages envoyés) durant un cycle d'apprentissage.

La figure 3 représente le nombre d'actions effectuées par l'agent durant une expérience ainsi que le nombre de messages envoyés. De même, que sur la courbe précédente, nous représentons la moyenne obtenue sur 50 expériences consécutives. Les courbes $(+)$ et $(*)$ représentent les résultats obtenues avec notre modèle de mémoire alors que les courbes $(\times)$ et $(\square)$ représentent les résultats obtenues avec une mémoire temporelle. Dans les deux cas, l'agent apprend rapidement à n'envoyer que les deux seuls messages nécessaires à l'obtention de sa ressource Y. Cependant nous pouvons noter que l'agent utilisant notre modèle de mémoire effectue cependant moins d'actions que l'agent utilisant une mémoire temporelle. Ceci peut s'expliquer par le fait que la mémoire temporelle ne stocke pas les réponses des commandes. Ainsi l'agent ne pouvant faire la différence entre les réponses *agree(creerX)* et *agree(creerX ;plustard)* apprend qu'il vaut mieux attendre plus longtemps et être donc sur que l'agent A ait bien effectuer *creerX* plutôt que de recevoir un message négatif de la part de l'agent B. Ce problème ne se pose pas avec notre modèle de mémoire où l'agent est donc capable d'adapter son comportement en fonction des réponses qu'il reçoit.

4 Conclusion et perspectives

Dans cet article, nous avons présenté un modèle de mémoire pour l'apprentissage de la communication et tout particulièrement pour les messages de type commande et contrôle. Notre solution repose sur la mémorisation des réponses et des dates couplée à un algorithme itératif permettant d'augmenter progressivement les valeurs que peuvent prendre la mémoire.

Nous avons évalué notre modèle en le comparant à un modèle de mémoire temporelle dont la taille est elle aussi définie en utilisant notre algorithme incrémental. Nous avons montré que sur un exemple simple, notre modèle était capable d'apprendre une politique pour la communication en effectuant moins de cycles d'apprentissage que la mémoire temporelle. De plus, notre modèle de mémoire permet à l'agent d'atteindre son but en effectuant moins de cycles d'exécution. En effet, en stockant plus d'information comme les réponses, l'agent utilisant notre modèle de mémoire est capable d'adapter au mieux son comportement alors qu'un agent utilisant une mémoire temporelle doit opter pour une politique moins risquée (plus de temps d'attente après l'envoi d'un message) mais donc moins optimale.

Ces résultats sont donc encourageants et nous aimerions les confirmer en testant notre modèle de mémoire sur des problèmes plus complexe en augmentant par exemple le nombre d'agents et de messages que l'agent apprenant peut envoyer. Cependant, nous pouvons présager des

problèmes de convergence avec un nombre de messages très grands dû à l'utilisation du Q-Learning.

Une autre perspective de ce travail serait d'étendre notre modèle de communication à d'autres messages que la commande et le contrôle. Nous aimerions pouvoir apprendre à communiquer des réponses mais aussi pouvoir effectuer des engagements ou des promesses. Dans ce cas il sera probablement nécessaire d'étoffer notre modèle de mémoire pour pouvoir prendre en compte ces nouveaux messages.

Enfin nous aimerions à plus long terme déterminer de nouveaux mécanismes permettant de réduire la taille de la mémoire utilisée. Pour cela, nous nous intéressons à la généralisation des éléments mémorisés sous forme de concepts plus facile à réutiliser dans de nouveaux contextes et plus simple à stocker. Ceci nous permettrait d'adapter notre algorithme à l'apprentissage de la communication pour les problèmes de grandes tailles.

Références

[1] Arnaud Canu and Abdel-Illah Mouaddib. Collective decision- theoretic planning for planet exploration. In *Proceedings of the 2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, ICTAI '11, pages 289–296, 2011.

[2] Alain Dutech and Manuel Samuelides. Un algorithme d'apprentissage par renforcement pour les processus décisionnels de Markov partiellement observés : apprendre une extension sélective du passé. *Revue d'Intelligence Artificielle*, 17(4), 2003.

[3] J. Ferber. *Les systèmes multi-agents. Vers une Intelligence Collective*. InterEditions, 1995.

[4] Shirley Hoet and Nicolas Sabouret. Apprentissage par renforcement d'actes de communication dans un contexte multi-agent. *Revue d'Intelligence Artificielle*, 24(2) :159–188, 2010.

[5] T. Kasai, H. Tenmoto, and A. Kamiya. Learning of communication codes in multi-agent reinforcement learning problem. In *Proceedings of the 2008 IEEE Conference on Soft Computing in Industrial Applications (SMCia/08)*, pages 1–6, 2008.

[6] Pier Luca Lanzi. Adaptative agents with Reinforcement Learning and Internal Memory. In *Proceedings of the 6th Inter. Conf. on the Simulation of Adaptive Behavior (SAB2000)*, pages 333–342, 2000.

[7] Andrew Kachites Mccallum. *Reinforcement learning with selective perception and hidden state*. PhD thesis, The University of Rochester, 1996. Supervisor-Dana Ballard.

[8] Francisco S. Melo and Manuela Veloso. Learning of coordination : exploiting sparse interactions in multiagent systems. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS '09)*, pages 773–780. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[9] Andrew Nuxoll and John E. Laird. Extending cognitive architecture with episodic memory. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1560–1564. AAAI Press, 2007.

[10] Nicolas Sabouret. A Model of Requests about actions for active components in the semantic web. In *Proceedings of the STarting AI Researchers Symposium (STAIRS 2002)*, 2002.

[11] Ming Tan. Multi-agent reinforcement learning : Independent vs. cooperative agents. In *Proceedings of the 10th International Conference on Machine Learning*, pages 330–337. Morgan Kaufmann, 1993.

[12] Michael T. Todd, Yael Niv, and Jonathan D. Cohen. Learning to use working memory in partially observable environments through dopaminergic reinforcement. In *Advances in Neural Information Processing Systems 21*, pages 1689–1696, 2009.

[13] Christopher J. C. H. Watkins. *Learning from delayed rewards*. PhD thesis, Cambridge University, Cambridge, United Kingdom, 1989.

[14] Ping Xuan, Victor Lesser, and Shlomo Zilberstein. Communication decisions in multi-agent cooperation : Model and experiments. In *Proceedings of the 5th International Conference on Autonomous Agents*, pages 616–623. ACM Press, 2001.

[15] E. A. Zilli and M. E. Hasselmo. Modeling the role of working memory and episodic memory. In *Behavioral Tasks*, volume 18, pages 193–209. Hippocampus, 2008.