Contribution au problème de la sélection de l'action en environnement partiellement observable

Olivier Sigaud * — **Pierre Gérard** *,**

* Dassault Aviation
B.P. 300, 92552 St-Cloud Cedex
olivier.sigaud@dassault-aviation.fr
** AnimatLab - LIP6
8 rue du Capitaine Scott, 75015 Paris
pierre.gerard@lip6.fr

RÉSUMÉ. *La sélection de l'action par les agents dans un environnement partiellement observable est un problème central de l'intelligence artificielle située. Nous montrons dans cet article comment le cadre mathématique des POMDPs permet de traiter le problème. Nous présentons l'intérêt de l'algorithme* U-Tree *et nous identifions la difficulté à introduire dans le bon ordre les distinctions dans l'arbre de décision que l'algorithme engendre. Nous montrons comment un algorithme génétique permet de trouver une solution. Nous nous livrons alors à une discussion de notre travail vis à vis des systèmes de classeurs à états internes à partir de l'exemple de ZCSM, ce qui nous conduit à une comparaison avec* Alecsys *et* HQ-learning*. Nous concluons par une synthèse de ce que nous retenons de ces différents systèmes.*

ABSTRACT. *Action selection in a partially observable environment is a key problem of situated artificial intelligence. We show in this paper how the problem is solved in the POMDP framework. We present the assets of the* U-Tree *algorithm, and we highlight the difficulties to introduce in the right order the distinctions in the generated decision tree. We show how a genetic algorithm provides a solution. Then we discuss our work with respect to classifier systems with internal state, taking ZCSM as an example, which leads to a comparison with* Alecsys *and* HQ-learning*. We conclude with a synthesis of the assets of all these systems.*

MOTS-CLÉS: *POMDP, algorithme génétique, arbre de décision, systèmes de classeurs, apprentissage par renforcement*

KEY WORDS: *POMDP, genetic algorithm, decision tree, classifier system, reinforcement learning*

1. Introduction

1.1. *Les processus de décision markoviens*

Par opposition à l'intelligence artificielle symbolique traditionnelle qui conçoit des programmes informatiques déconnectés de leur environnement, l'intelligence artificielle située s'intéresse à la modélisation d'agents caractérisés par une capacité d'interaction avec celui-ci.

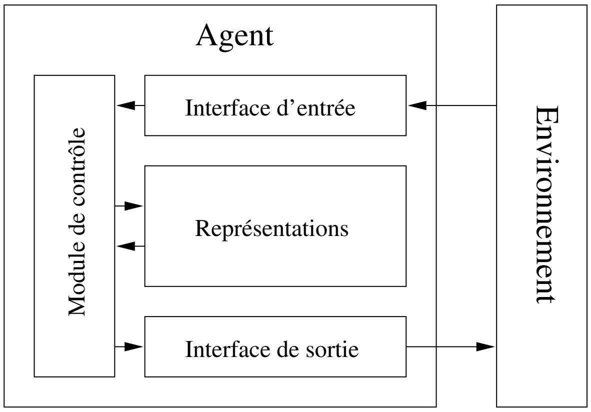

Figure 1. *Architecture d'un agent situé*

Lorsque l'environnement est mal connu et incertain, il est souvent très difficile d'en concevoir un modèle réaliste. Dans ce cas de figure, les mécanismes d'apprentissage et de sélection de l'action permettent de réaliser des systèmes efficaces qui ne nécessitent pas une modélisation préalable de l'environnement par le concepteur. Pour mettre en œuvre des agents situés, nous considérons des agents dont l'architecture est illustrée par la figure 1. Le module de contrôle modifie les représentations en fonction de son expérience et se sert de celles-ci pour décider de l'action sur l'environnement. L'interface d'entrée traduit des informations reçues de l'environnement en informations exploitables par le module de contrôle et l'interface de sortie traduit les actions décidées en actions effectives sur l'environnement.

Dès lors que la dynamique des interactions avec l'environnement peut se représenter par des distributions de probabilité, le cadre mathématique des *processus de décision markoviens* (MDP) s'avère adapté à la réalisation d'un apprentissage par renforcement. Un processus de décision markovien se décrit par :

– un ensemble fini d'états discrets S ;

– un ensemble fini d'actions discrètes A ;

– une fonction de transition $T : S \times A \rightarrow \Pi(S)$ où $\Pi(S)$ est une distribution de probabilité ;

– une fonction de renforcement $R : S \times A \rightarrow \Re$ qui associe une punition ou une récompense scalaire à chaque transition.

Pour un agent donné, l'objectif consiste en général à maximiser globalement R sur un parcours ou un ensemble de parcours. D'après le théorème de la programmation dynamique [BEL 57], maximiser localement R à chaque pas permet de maximiser R globalement, sous réserve que l'hypothèse de Markov soit vérifiée. Cette hypothèse stipule que l'état de l'agent à l'instant $t+1$ dépend uniquement de son état à l'instant t et de son action : $S(t+1) = f(S(t), A(t))$.

Ce théorème est à la base des méthodes d'apprentissage par renforcement [KAE 96]. Ainsi, le Q-LEARNING [WAT 89] consiste à associer une qualité Q à chaque action dans chaque état, cette qualité reflétant l'espérance de renforcement. Un second théorème nous assure que, si l'hypothèse de Markov est vérifiée et si l'on parcourt tous les états un nombre infini de fois, en appliquant l'algorithme *value iteration* (issu de la programmation dynamique) aux qualités, alors les Q convergent de telle façon que choisir à chaque instant l'action de meilleure qualité permet de trouver la commande optimale, c'est-à-dire la commande qui maximise une fonction du renforcement reçu, par exemple la moyenne sur la durée de vie de l'agent.

Malheureusement, l'hypothèse de Markov n'est pas toujours vérifiée en pratique. Un agent situé dispose souvent d'une observation de son environnement qui ne lui permet pas de déterminer avec certitude dans quel état il est. L'apprentissage peut encore fonctionner, mais la convergence n'est plus garantie. Les chaînes de Markov cachées (HMM) [RAB 86] constituent un cadre mathématique qui permet de rendre compte des cas où l'observation ne suffit pas à déterminer l'état. Au lieu d'observer les états, l'apprenti observe un ensemble discret de symboles Ω et il doit apprendre l'inverse d'une fonction d'observation $O : S \rightarrow \Pi(\Omega)$ pour déterminer dans quel état il est en fonction de ce qu'il observe. Mais il manque aux HMM la notion d'action, nécessaire à la modélisation des agents qui intéressent l'intelligence artificielle située. Les *processus de décision markoviens partiellement observables* (POMDP) sont la synthèse entre les MDP et les HMM, et constituent à ce titre le modèle adéquat pour la simulation d'agents qui disposent d'une observation imparfaite.

1.2. *Les processus de décision markoviens partiellement observables*

Un processus de décision markovien partiellement observable se compose d'un MDP classique comprenant les éléments $< S, A, T, R >$ décrits plus haut, auquel on ajoute un ensemble discret de symboles 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.

La résolution classique d'un POMDP [KAE 98] consiste à supposer que l'agent ne sait pas dans quel état il est - mais connaît un modèle du MDP sous-jacent à ses observations - puis à introduire l'état dans lequel l'agent croit être (*belief state*). On sait alors, en théorie, trouver la commande optimale parmi toutes les séquences d'actions de longueur inférieure ou égale à n. On sait en effet trouver n pour que la meilleure commande soit optimale à ϵ près au sens indiqué plus haut.

En pratique, pour contourner un problème d'explosion combinatoire, des heuristiques spécifiques, comme l'algorithme WITNESS [LIT 95], évitent d'engendrer les séquences qui ont peu de chance d'être optimales. Plus récemment, Hansen propose dans [HAN 98] un algorithme d'itération sur les lois de commande (*policy iteration*) qui construit directement des automates représentant la commande optimale. Mais ces heuristiques qui reposent trop directement sur le formalisme ne font que retarder l'explosion combinatoire, et restent limitées au traitement de problèmes de très petite dimension. Et surtout, elles présentent l'inconvénient de présupposer la connaissance du MDP sous-jacent. Cette hypothèse suppose une modélisation préalable de l'environnement de l'agent et rend le cadre classique de résolution des POMDPs mal adaptée aux types de problèmes qui nous intéressent ici : ceux pour lesquels le concepteur n'a pas de modèle de l'environnement.

Dans la suite de cet article, nous présentons l'algorithme U-TREE de McCallum, qui propose dans [MCC 95] des solutions heuristiques plus radicales, permettant de s'affranchir de cette hypothèse. Nous nous concentrons ensuite sur le problème du choix de l'ordre d'introduction des distinctions associées à cet algorithme.

2. L'algorithme U-Tree

2.1. *Présentation*

Plutôt que de se donner a priori une représentation des états du monde vers laquelle les états perçus par l'agent doivent converger, l'idée de McCallum consiste à construire un arbre de décision associant une action à chaque feuille. Chacune de ces feuilles représente un état avec autant de perceptions et de mémoire qu'il est nécessaire pour distinguer cet état des autres. Un agent utilisant l'algorithme U-TREE garde une mémoire de toutes ses observations, renforcements reçus et actions. Toutes ces instances sont organisées en une liste représentant la séquence des expériences passées de l'agent, et chacune des instances est placée dans une feuille de l'arbre. L'agent se sert de cette mémoire d'instances pour construire l'arbre de décision et choisir l'action.

L'architecture d'un tel système est présentée sur la figure 2. On peut aussi se référer aux figures 4 et 5 pour mieux comprendre la structure de l'arbre de décision.

Chaque noeud de l'arbre, par les arcs qui en sont issus, introduit une nouvelle distinction qui repose sur une observable présente, sur une action passée ou sur une

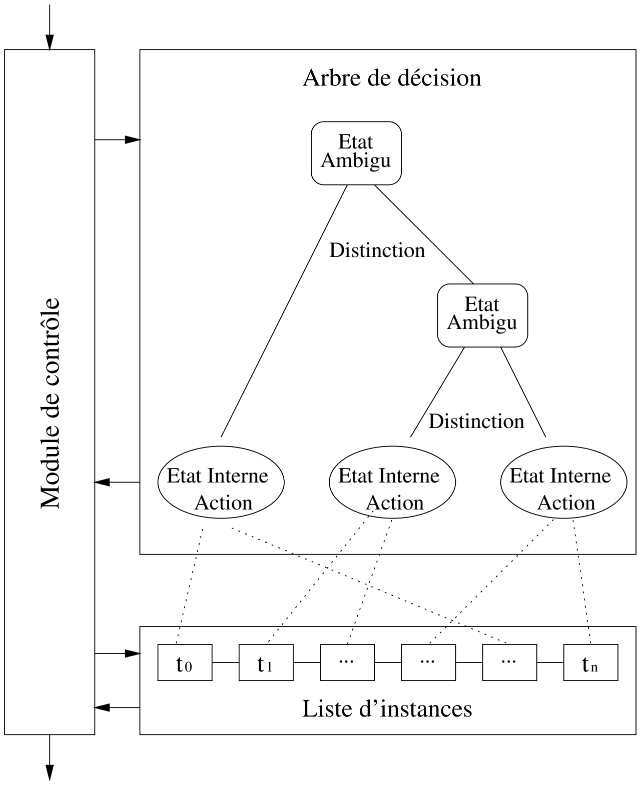

Figure 2. *La forme des représentations d'un agent utilisant l'algorithme* U-TREE

observable passée [1]. A chaque instant, l'agent confronte ses perceptions présentes et sa mémoire d'instances à son arbre de décision pour déterminer son état et décider de l'action la plus adaptée pour maximiser son renforcement.

Pour construire cet arbre, on commence avec une seule feuille. Par la suite, les feuilles pour lesquelles un état caché est détecté sont partitionnées et deviennent des noeuds. A chaque découpage, des arcs issus du nouveau noeud sont créés, arcs qui expriment la distinction introduite. Chaque arc mène à une nouvelle feuille et chacune représente un nouvel état plus spécifique que celui exprimé par la feuille de départ. En outre, les instances de la feuille initiale sont réparties entre les deux nouvelles feuilles. Un état caché est révélé par le fait que l'agent reçoit des renforcements différents dans ce qu'il considère comme le même état. Ainsi, la dimension de l'espace d'état ne croît qu'en fonction des nécessités et continue de croître en s'appuyant sur la mémoire tant

[1] Nous appelons *observable* une dimension de la perception. Dans l'exemple qui nous servira d'illustration, un agent situé dans un labyrinthe ne perçoit que la présence ou l'absence de murs vers chacune des quatre directions cardinales. Chacune de ces quatre informations est une observable. Le vecteur formé de ces quatre éléments est une observation.

que des états cachés empêchent l'agent de maximiser son renforcement.

En conséquence, si l'agent dispose d'observables lui donnant plus d'informations que nécessaire pour mener à bien sa tâche, sa représentation peut converger vers une représentation dans laquelle l'hypothèse de Markov est vérifiée sans qu'il utilise à chaque instant la totalité des observables dont il dispose. L'algorithme dote en quelque sorte l'agent d'une capacité d'attention sélective : il ne s'intéresse qu'aux informations pertinentes pour accomplir sa tâche. Ainsi, l'agent est capable de faire face au manque comme à la surabondance d'informations.

2.2. *Description de l'algorithme*

A chaque pas de temps, l'agent reçoit de son environnement une observation o_t, un renforcement r_t, et effectue une action a_t sur son environnement. Une observation est un vecteur d'observables. Ces éléments sont regroupés dans des instances de la forme $T_t =< T_{t-1}, a_{t-1}, o_t, r_t >$ placées dans l'arbre de décision.

Chaque feuille de l'arbre représente un état S. Dès qu'une instance est créée, on doit la placer dans une feuille de l'arbre. Pour déterminer dans quelle feuille de l'arbre placer l'instance, on part de la racine et on descend en choisissant l'arc dont le label exprime les conditions vérifiées par l'instance selon la distinction exprimée par les arcs issus du premier noeud. Ces distinctions portent sur la valeur des observables ou des actions à l'instant présent ou à des instants antérieurs.

On note $\tau(s, a)$ l'ensemble des instances de la feuille s qui contiennent l'action a. A chaque fois qu'on place une instance dans une feuille, on met à jour [2] :

- $|\tau(s, a)|$, le nombre d'instances de la feuille s qui contiennent l'action a ;
- $R(s, a) = \frac{|\sum_i T_i \in \tau(s,a) r_i|}{|\tau(s,a)|}$, le renforcement moyen reçu pour l'action a dans la feuille s ;
- $NbTrans(s|s', a)$, le nombre de transitions mettant en œuvre l'action a entre la feuille s' qui contient l'instance précédente et la feuille s ;
- $Pr(s'|s, a) = \frac{NbTrans(s'|s,a)}{|\tau(s,a)|}$, la probabilité d'aboutir en s' quand on effectue l'action a en s.

Ces quantités permettent de calculer la qualité $Q(s, a)$ de l'action a dans la feuille s :

$$Q(s, a) = R(s, a) + \gamma \sum_{s'} Pr(s'|s, a)U(s')$$

où $U(s') = max_a(Q(s', a))$ est l'utilité de la feuille s'.

La croissance de l'arbre est gouvernée par la nécessité de lever les ambiguïtés au sein des feuilles. A intervalles réguliers de k pas, on tente de découper chaque feuille

[2] Notre présentation est légèrement modifiée par rapport à [MCC 95], compte tenu des améliorations que McCallum suggère d'apporter à son algorithme.

en introduisant de nouvelles distinctions, puis on applique le test de Kolmogorov-Smirnov sur les distributions de renforcements reçus et attendus entre la feuille initiale et chacune des nouvelles feuilles ainsi engendrées. Si le test est positif, la distinction est pertinente et les nouvelles feuilles sont conservées dans l'arbre. Sinon, la distinction est ignorée jusqu'à une tentative ultérieure.

Introduire une nouvelle distinction revient à choisir une observable ou une action et un pas de temps selon lesquels découper une feuille. Ce choix peut se faire de façon aléatoire. Mais nos premières expérimentations montrent qu'une variation dans l'ordre d'introduction de nouvelles distinctions entraîne de grandes disparités dans la qualité du comportement et la taille de l'arbre obtenu. Ce point n'est pas traité de façon satisfaisante par McCallum [MCC 95], qui laisse entendre qu'il fixe cet ordre de façon *ad hoc* en fonction de l'application. Nous proposons ici des solutions à ce problème.

3. L'ordre d'introduction des distinctions

3.1. *Mise en évidence des états cachés*

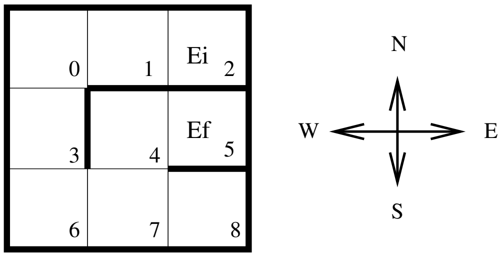

Figure 3. *Un problème jouet*

Considérons le problème jouet de la figure 3. L'agent doit aller de E_i à E_f. Il ne peut observer que la présence ou l'absence de murs autour de la case dans laquelle il se situe. Il dispose ainsi de quatre observables, une pour chaque mur. Chacune d'elles vaut V quand il y a un mur, F sinon. Les quadruplets formés par les quatre observables prises dans l'ordre NESW constituent l'ensemble des observations possibles. Ainsi, dans la case 0 de la figure 3, l'agent observe $VFFV$.

A chaque instant, l'agent a le choix entre quatre actions : aller au nord (N), à l'est (E), au sud (S) ou à l'ouest (W). Il peut très bien entreprendre de se diriger vers un mur et s'y cogner, auquel cas il reste sur place et reçoit un renforcement négatif. Empêcher a priori de telles actions, qu'un observateur extérieur jugerait absurdes, suppose l'introduction préalable de connaissances liées au domaine. Si de telles solutions heuristiques s'avèrent efficaces en pratique, c'est une option que nous n'avons pas choisie, afin de conserver la généricité de la solution.

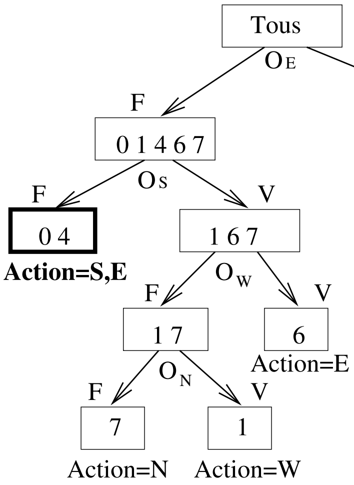

Figure 4. *Décomposition sans mémoire du labyrinthe*

La figure 4 représente l'arbre distinguant au mieux les cases du labyrinthe à partir de l'observation présente, toute distinction supplémentaire n'engendrant que des feuilles vides. On y a indiqué les cases correspondant à chaque description et l'action optimale en fonction de la meilleure estimation du renforcement attendu. Il apparaît que l'agent ne peut pas distinguer les cases 0 et 4 d'une part, ni les cases 2, 5 et 8 d'autre part. Les feuilles correspondantes sont donc ambiguës.

3.2. *Distinction à l'aide de la mémoire*

Aussi longtemps que les distinctions introduites dans l'arbre portent sur l'instant présent, l'agent est réactif et considère le problème comme markovien. Il semble donc naturel de commencer par introduire les observables de l'instant présent aussi longtemps que ces distinctions sont pertinentes, avant d'introduire des distinctions portant sur les instants précédents ou suivants.

Pour distinguer davantage, l'agent peut se reposer sur sa mémoire. Il faut alors qu'il introduise l'action par laquelle il a abouti dans la feuille ambiguë et les observables qui lui permettent de distinguer la case dans laquelle il était avant d'accomplir cette action. C'est ce qui apparaît sur la figure 5, qui se lit de la façon suivante : si l'agent a effectué l'action N et a abouti en 0 ou 4, c'est qu'il était en 0, 3, 4 ou 7. S'il a observé $O_S = V$, il était en 7 donc il est en 4. Sinon, il était dans 0, 3 ou 4 et il faut distinguer davantage. On note que certaines actions n'ont pas pu mener dans la feuille ambiguë, donc la feuille correspondante reste vide et il n'est pas nécessaire de partitionner. On observe par ailleurs que, si l'agent était en 0 ou 4 au pas précédent parce que sa dernière action l'aurait mené à se cogner dans un mur, il n'est toujours pas

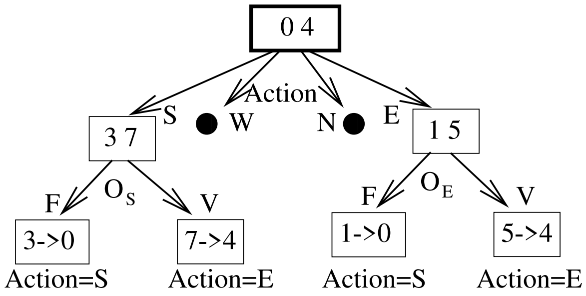

Figure 5. *Levée des ambiguïtés à l'aide de la mémoire*

possible d'introduire une distinction discriminante à ce pas de temps, et il faut partitionner dans le passé jusqu'au moment où l'agent est arrivé en 0 ou 4 à partir d'un état déjà identifié comme différent. Dans un cas extrême, la profondeur de l'arbre peut être aussi grande que le nombre d'actions entreprises par l'agent au cours de sa vie, s'il a passé son temps à se cogner dans le même mur. On voit donc la difficulté spécifique qu'introduit le fait de pouvoir se cogner dans les murs.

3.3. *Distinction par l'action à venir*

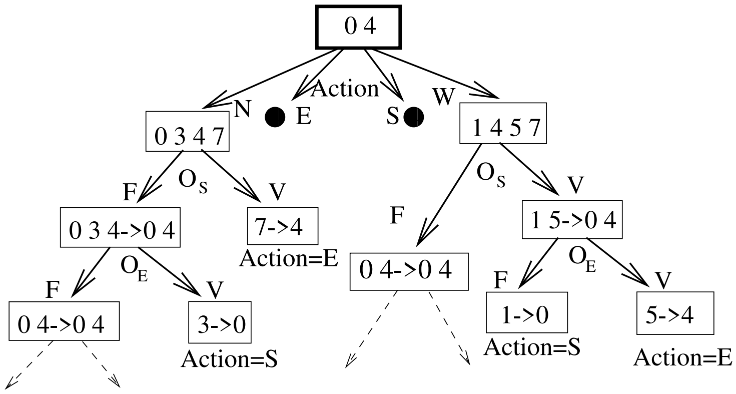

Figure 6. *Levée des ambiguïtés à l'aide du futur*

Pour distinguer davantage, l'agent peut aussi se tourner vers le futur, comme cela apparaît sur la figure 6. Il spécifie alors ce qu'il faut faire pour déterminer dans quel état il était. Avec un schéma du même type que précédemment, il peut dire que si, étant en 0 ou 4, il effectue l'action E, et s'il observe $O_E = F$, alors c'est qu'il sera en 1 et qu'il est actuellement en 0. S'il observe $O_E = V$ pour la même action, c'est qu'il sera en 5 et qu'il est actuellement en 4.

Cette méthode permet de choisir une action destinée à lever les ambiguïtés. Elle semble pouvoir donner lieu à un usage plus efficace des distinctions. En revanche, elle peut conduire l'agent à choisir des actions qui l'écartent de son objectif et donc à adopter un comportement sous-optimal. Des expérimentations restent à mener pour évaluer la qualité de cette stratégie.

3.4. *Recherche du bon ordre par un algorithme génétique*

Nous avons indiqué comment tirer parti de la mémoire ou des actions futures pour lever les ambiguïtés dans l'arbre de décision. Mais il reste encore à fixer l'ordre dans lequel on introduit les distinctions d'un instant donné. Introduire ces distinctions dans un ordre aléatoire peut mener à des performances très variables.

Pour atteindre de bonnes performances, nous utilisons un algorithme génétique [HOL 75, GOL 89]. Un chromosome est associé à chaque agent. Celui-ci en extrait l'ordre d'introduction des distinctions selon lequel il fait croître son arbre de décision. Le chromosome s'exprime comme une liste circulaire de critères de distinction sur laquelle l'agent maintient un marqueur. A chaque fois qu'il doit choisir une observable pour introduire une distinction, l'algorithme lit l'observable pointée par le marqueur, qui avance d'un cran. Si la distinction ainsi introduite est valide, on l'applique à l'arbre, sinon elle est ignorée. La liste étant circulaire, le marqueur revient au début quand il atteint la fin du chromosome.

On fait apparaître sur la figure 7 un exemple de chromosome et on montre comment on l'utilise pour engendrer l'arbre de la figure 4.

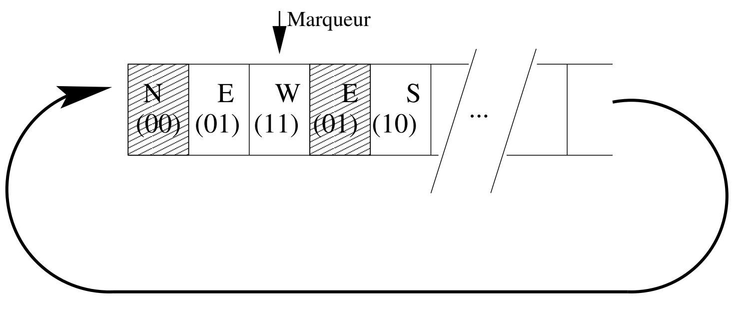

Figure 7. *Un exemple de chromosome*

– Lors de la première tentative de découpage, l'algorithme essaie l'observable du mur Nord. Comme il n'a pas encore assez d'instances à la racine, le découpage échoue, le marqueur avance.
– Lors de la seconde tentative de découpage, il essaie l'observable du mur Est. Cette fois le découpage réussit, la racine est coupée en deux conformément à ce qui apparaît sur la figure 4, le marqueur avance.

– La troisième tentative de découpage, selon Ouest (W), est appliquée à la branche de droite de la figure 4. Elle réussit, le marqueur avance.
– La quatrième, selon Est, appliquée à la branche de gauche, échoue car on a déjà découpé selon Est. C'est la cinquième, selon Sud, qui réussit. Et ainsi de suite.

Avec ce codage, chaque agent dispose de sa propre dynamique d'expansion de l'arbre. Nous utilisons un algorithme génétique très classique :
– les chromosomes de première génération sont choisis aléatoirement ;
– on classe les agents d'une génération par ordre de qualité décroissante (l'agent est d'autant meilleur qu'il trouve plus vite l'état final) ;
– les deux meilleurs sont les parents de la génération suivante ;
– on produit les descendants en croisant les chromosomes des deux parents.

Dans des expérimentations préliminaires, nous avons utilisé une population de moins de dix agents, chacun doté d'un chromosome de longueur 20. Ces expérimentations nous ont permis de constater que cette procédure très simple suffit à sélectionner un agent optimal pour le problème de la figure 4 : partant de la case 2, il apprend à rejoindre la cas 5 en passant successivement par les cases 2, 1, 3, 6, 7 et 4. Il reste toutefois à la valider sur des applications plus difficiles.

4. Comparaison avec les systèmes de classeurs

4.1. *Généralités sur les systèmes de classeurs*

Un système de classeurs [HOL 75, BOO 89, GOL 89] simple est composé d'une population de règles appelées classeurs. Ces règles sont composées d'une partie condition et d'une partie action. Si la situation de l'agent correspond à la partie condition d'un classeur, celui-ci devient candidat pour que le système exécute la partie action de la règle. Parmi ces candidats, les classeurs activés sont choisis en fonction de la force des différents classeurs. Ces forces sont mises à jour par apprentissage par renforcement. L'évolution de la population des classeurs est dévolue à un algorithme génétique dans lequel la force des règles est utilisée comme mesure de l'adaptation *(fitness)* des classeurs.

Comme les systèmes de classeurs, notre version modifiée de l'algorithme U-TREE fait intervenir simultanément un algorithme génétique et une capacité d'apprentissage par renforcement. C'est pourquoi nous nous livrons dans la suite à une comparaison informelle entre notre algorithme et les systèmes de classeurs.

Nous présentons brièvement les systèmes de classeurs ZCS[3] [WIL 94], puis ZCSM[4] [CLI 94] : une version de ZCS enrichie de la gestion d'états internes. Ces systèmes

[3]Zeroth-level Classifier System
[4]Zeroth-level Classifier System with Memory

sont moins performants mais plus simples respectivement que XCS [WIL 95] et XCSM [LAN 98].

4.2. *Présentation de ZCS : un exemple de systèmes de classeurs sans état interne*

La population des classeurs de ZCS [WIL 94] est de taille fixe. Ces classeurs sont tous de la forme :

$$[Observation] \rightarrow [Action\, externe](Force)$$

Chacune des parties $[Observation]$ et $[Action\, externe]$ d'un classeur est de la forme $\{V, F, \#\}^d$ pour l'observation et $\{V, F, \#\}^a$ pour l'action externe, où le symbole $\#$ est équivalent à $''V\ ou\ F''$. Le nombre de symboles $\#$ donne une mesure de la généricité du classeur. La partie $[Action\, externe]$ doit être de longueur suffisante pour coder toutes les actions possibles et chaque symbole ternaire de la partie $[Observation]$ correspond à une observable. $(Force)$ est la force du classeur. Elle est utilisée pour estimer la qualité du classeur en terme de renforcement attendu.

A chaque pas de temps, l'interface d'entrée de l'agent convertit les informations sensorielles en un message d'entrée de la forme $\{V, F\}^d$. Ce message est directement comparé aux parties $[Observation]$ de tous les classeurs. Les classeurs appariés sont ceux dont la partie $[Observation]$ est identique au message d'entrée, aux symboles $\#$ près. Les classeurs appariés sont regroupés par ensembles dont la partie $[Action\, externe]$ est semblable. Les forces des classeurs de chaque ensemble sont sommées et l'un d'eux est sélectionné aléatoirement, avec une pondération par la somme des forces. La partie $[Action\, externe]$ correspondante est envoyée à l'interface de sortie qui transforme ce message en action effective sur l'environnement.

L'apprentissage par renforcement permet d'ajuster la force des règles grâce à l'algorithme de la Bucket Brigade [HOL 75]. Cet algorithme est similaire au Q-LEARNING [WAT 89] et permet de propager un renforcement reçu à tous les classeurs qui ont contribué à le provoquer, en ajustant leurs forces. Ainsi, les classeurs les plus adéquats voient leur force augmenter alors que celle des moins adéquats diminue.

La population initiale des classeurs est générée aléatoirement. Son évolution est dévolue à deux mécanismes :

- le recouvrement qui crée un nouveau classeur chaque fois qu'aucun n'est adapté aux perceptions reçues, ou que la somme des forces des classeurs appariés et proposant la même action est très faible ;

- un algorithme génétique classique avec croisements et mutations qui fait évoluer la population des classeurs en prenant pour critère d'évaluation la force des classeurs.

Dès qu'un nouveau classeur est créé, celui de moindre force est supprimé.

4.3. *Présentation de ZCSM : un exemple de systèmes de classeurs avec état interne*

ZCS ne dispose pas d'états internes, et se montre inadapté à la résolution de problèmes non markoviens. Cliff et Ross [CLI 94] proposent le système ZCSM, qui est un système ZCS augmenté de la gestion d'un registre de mémoire de b bits représentant l'état interne de l'agent et permettant d'aborder des problèmes non markoviens. La figure 8 illustre la forme des représentations d'un agent utilisant ZCSM.

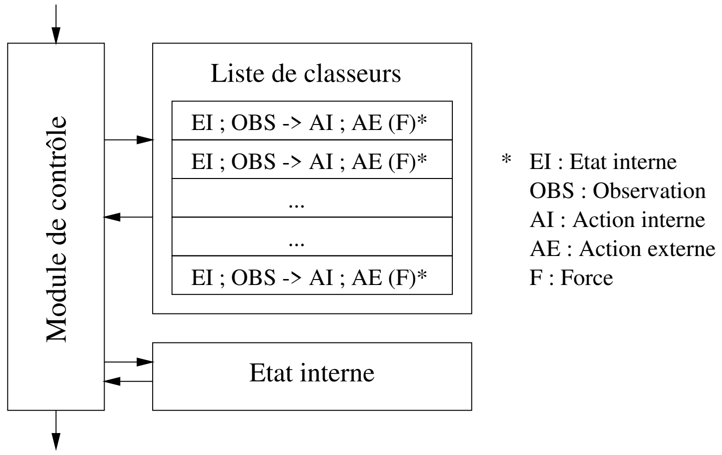

Figure 8. *La forme des représentations d'un agent utilisant ZCSM*

Pour prendre en compte la modification de l'état interne de l'agent au cours de son expérience, les classeurs prennent la forme :

$$[Etat\,interne][Observation] \rightarrow [Action\,interne][Action\,externe](Force)$$

Les parties $[Etat\,interne]$ et $[Action\,interne]$ sont de la forme $\{V, F, \#\}^b$. A la liste des actions possibles sont ajoutées une action nulle permettant à l'agent de modifier son état interne sans agir sur l'environnement. L'ajout de la partie $[Etat\,interne]$ dans la condition du classeur modifie le processus d'appariement : pour qu'un classeur produise son action, sa partie $[Observation]$ doit s'apparier avec le message d'entrée et sa partie $[Etat\,interne]$ doit s'apparier avec le registre de mémoire, aux symboles $\#$ près. L'action est décomposée en deux parties :

– l'action externe produit un message qui sera transformé en action effective sur l'environnement par l'interface de sortie ;

– l'action interne modifie l'état interne ; les bits du registre mémoire correspondant aux symboles $\#$ ne sont pas modifiés.

Les mécanismes d'apprentissage et d'évolution permettent de sélectionner les classeurs les plus pertinents, c'est-à-dire ceux qui produisent à la fois la bonne action selon le contexte, mais aussi qui modifient l'état interne de l'agent de manière adéquate.

4.4. *Discussion*

U-TREE augmenté d'un algorithme génétique et les systèmes de classeurs, que nous avons illustrés par ZCSM, utilisent tous les deux des mécanismes d'apprentissage par renforcement combinés à des algorithmes génétiques. Cette ressemblance nous conduit à examiner les différences entre ces deux méthodes. Si l'environnement est markovien, on peut réécrire un arbre de décision obtenu par U-TREE sous la forme d'un ensemble de classeurs qui ne modifient pas l'état interne de l'agent. On obtient alors un nombre de classeurs égal au nombre de feuilles. La partie $[Action\ externe]$ de chacun de ces classeurs code l'action optimale calculée dans la feuille. Les observables qui ne sont pas utilisées sur le chemin dans l'arbre menant à la feuille s'expriment par des symboles $\#$ dans la partie $[Observation]$ du classeur. Une première comparaison dans le cas markovien nous permet de dégager deux avantages de U-TREE sur les systèmes de classeurs.

D'une part, la configuration arborescente revient à se doter d'un ensemble de classeurs dont les conditions sont exclusives les unes des autres. Cela dispense de gérer la force des classeurs pour sélectionner celui qui sera déclenché parmi ceux dont les conditions sont vérifiées. Mais l'exclusivité des chemins de l'arbre, par opposition à la nécessité de gérer une concurrence entre des règles, peut devenir un inconvénient. En effet, alors que l'élimination des classeurs les plus faibles offre la possibilité de défaire ce qui a été fait, les choix faits par U-TREE pour ajouter une distinction dans l'arbre sont définitifs.

D'autre part, là où les systèmes de classeurs reposent sur le hasard pour engendrer de nouvelles conditions, U-TREE introduit des distinctions pertinentes en fonction des besoins et s'arrête au bon degré de généralité des règles. La différence provient de ce que U-TREE réalise un apprentissage fondé sur son expérience pour faire évoluer sa représentation, là où les systèmes de classeurs reposent sur un algorithme génétique. Garder une mémoire de toutes les instances et s'en servir pour guider la levée des ambiguïtés entre les états est très efficace, puisque des distinctions ne sont ajoutées que si elles sont pertinentes en regard de l'expérience de l'agent. Cette mémoire d'instances permet de converger plus rapidement vers une solution optimale que lorsqu'on laisse à des mécanismes évolutifs le soin de trouver les états internes qui permettront à l'agent de maximiser son renforcement.

Globalement, l'obtention d'une bonne représentation dans le cas markovien semble donc nettement plus efficace avec U-TREE qu'avec les systèmes de classeurs.

Lorsqu'on se tourne vers le cas non markovien, en revanche, la comparaison devient plus nuancée. Dès que l'agent se trouve dans une situation ambiguë, U-TREE utilise sa mémoire du passé à cet instant pour lever les ambiguïtés. De son côté, ZCSM positionne un état interne avant de rencontrer l'ambiguïté et assure la persistance de cet état interne jusqu'à ce qu'un autre classeur vienne le modifier. Le mécanisme de U-TREE semble économique et évite d'avoir à spécifier à l'avance le nombre d'états internes pour résoudre le problème posé. Mais le mécanisme de ZCSM s'avère avantageux dans la mesure où il permet d'associer le positionnement de l'état interne à l'observation de *situations clefs* dont il faut se souvenir, puis assure la propagation des conséquences de ces observations sur un nombre de pas de temps potentiellement infini. ZCSM permet donc de s'affranchir de la spécification d'un intervalle de temps entre l'observation d'une situation clef et l'action qui doit en découler, ce que ne permet pas U-TREE puisque les nœuds portent une condition temporelle explicite.

Si l'information permettant d'identifier un état caché se trouve loin dans le passé [5], U-TREE devra réaliser une très grande et inutile expansion de l'arbre en profondeur. Le problème est encore accentué par le fait que U-TREE est incapable de revenir sur une mauvaise construction de l'arbre de décision. En cela, les systèmes de classeurs sont plus souples et doivent permettre la construction de hiérarchies de comportements.

Enfin, U-TREE augmenté de mécanismes évolutionnistes et ZCSM font tous les deux appel à un algorithme génétique, mais une différence fondamentale suffit à les distinguer nettement sur l'usage qu'ils en font. En effet, les systèmes de classeurs font évoluer la population des classeurs, mais restent dans le cadre de l'apprentissage de la politique optimale par un seul agent. Par contre, notre algorithme fait évoluer une population d'agents qui sont évalués sur leur performance globale. La sélection ne s'opère donc pas au même niveau.

Ce premier examen informel fait donc apparaître, d'une part, l'intérêt d'une représentation de la mémoire faisant intervenir des *situations clés* indépendantes de la succession rigide des instants et, d'autre part, l'efficacité d'algorithmes fondés sur l'expérience plutôt que sur des opérateurs génétiques aléatoires. L'étude d'un système de classeurs à états internes et dont l'évolution des classeurs serait fondée sur l'expérience et inspirée de U-TREE nous semble donc prometteuse.

5. Travaux apparentés

5.1. *Les opérateurs non-génétiques dans* ALECSYS

Cette idée d'utiliser le formalisme des systèmes de classeurs et de guider la formation de nouveaux classeurs par l'expérience a déjà été exploitée par Dorigo [DOR 94] pour le système ALECSYS.

Dans ALECSYS, chaque classeur garde une mémoire des renforcements reçus. Cette mémoire est utilisée pour calculer la variance de ces renforcements. Une forte variance indique que le classeur oscille en raison d'une trop faible spécialisation. Dorigo introduit alors l'opérateur *Mutspec* qui crée deux classeurs spécialisés à partir

[5] Dans notre exemple du labyrinthe, c'est le cas, par exemple, quand l'agent se cogne longtemps dans les murs d'une case ambiguë

d'un classeur général. Cet opérateur est utilisé par l'algorithme génétique en complément de l'opérateur de mutation. Il utilise l'expérience de l'agent pour engendrer de nouveaux classeurs. Il permet donc de ne pas attendre que le hasard fasse bien les choses et que l'opérateur de mutation engendre les classeurs spécialisés pertinents. L'observable à spécialiser est choisie aléatoirement. Si le choix ne s'avère pas pertinent, les classeurs spécialisés seront éliminés par le mécanisme de sélection et l'opérateur *Mutspec* sera appliqué une nouvelle fois.

Ce mécanisme est assez semblable à celui utilisé dans U-TREE pour l'introduction de distinctions dans le cas markovien, à ceci près que l'observable à introduire est choisie de manière raisonnée dans U-TREE, ce qui rend l'algorithme un peu moins dépendant du hasard.

5.2. HQ-LEARNING

Dans [WIE 97], Wiering et Schmidhuber proposent l'algorithme HQ-LEARNING qui aborde les problèmes non markoviens en activant séquentiellement un nombre déterminé d'agents, chacun capable de résoudre un problème markovien. La désactivation d'un agent et l'activation du suivant ont lieu quand le premier atteint un sous-but défini par une observation.

Chaque agent dispose :

- d'une *Q-table* faisant correspondre à chaque paire $(Observation, Action)$ possible un renforcement attendu calculé par un algorithme de Q-LEARNING ;
- d'une *HQ-table* faisant correspondre à chaque observation une valeur exprimant l'intérêt de l'observation en tant que sous-but, et calculée par un autre algorithme de Q-LEARNING.

A chaque fois qu'un agent devient actif, il choisit un sous-but de manière stochastique en fonction des qualités des différents sous-buts estimées dans la *Q-table*. Les valeurs de la *Q-table* sont utilisées pour décider de l'action et sont modifiées quand l'agent atteint son sous-but. Les valeurs de la *HQ-table* sont ajustées en fonction des renforcements reçus de l'environnement.

L'apprentissage simultané des *HQ-tables* et des *Q-tables* des agents induit une dynamique qui permet au système d'identifier des situations clés (les sous-buts) et de découper un comportement en plusieurs comportements séquentiels. La séquence des sous-buts peut s'interpréter comme le découpage d'un environnement non markovien en plusieurs zones dans lesquelles l'hypothèse de Markov est vérifiée.

Dans le cas de HQ-LEARNING comme dans ZCSM, la résolution de problèmes non markoviens passe par l'identification de situations clés dans lesquelles il faut changer de comportement. Mais ZCSM effectue ce découpage en modifiant l'état interne, là où HQ-LEARNING procède à un changement d'agent. Chacun des deux algorithmes oblige le concepteur à spécifier le nombre maximal d'états internes. Dans

HQ-LEARNING, celui-ci est égal au nombre d'agents et dans ZCSM il dépend de la taille du registre mémoire.

Wiering et Schmidhuber [WIE 97] notent que durant une expérience, HQ-LEARNING change peu souvent d'agent actif et donc d'état interne. Par contre, ZCSM laisse aux soins d'opérateurs génétiques la création de règles faisant évoluer les registres de mémoire. Ceci entraîne une instabilité importante de l'état interne et une mauvaise résistance à l'augmentation du nombre d'états cachés.

6. Conclusion

En combinant un algorithme génétique classique avec l'algorithme U-TREE de [MCC 95], nous avons mis au point un algorithme qui combine les processus de sélection, de développement et d'apprentissage. En effet, le développement des agents (l'expansion de leur arbre de décision) est contraint à la fois par l'expression des chromosomes sélectionnés au fil des générations et par l'expérience acquise par les agents en interaction avec leur environnement.

Notre algorithme a fait la preuve de son efficacité sur un problème jouet, et se montre adapté à la résolution de problèmes non markoviens.

La comparaison que nous avons menée n'est pas exhaustive. Par exemple, nous n'avons pas examiné MONALYSA [DON 98] dont les objectifs s'apparentent aux nôtres. Nous ne nous sommes intéressés qu'à quelques aspects isolés de systèmes choisis pour nous permettre de dégager les éléments suivants, que nous retenons pour les intégrer dans un système futur :

– l'utilisation raisonnée de l'expérience pour guider la construction du système, comme c'est le cas dans HQ-LEARNING et ALECSYS et, plus encore, dans U-TREE qui utilise une mémoire d'instances ;
– la mise en place de mécanismes de sélection pour remettre en question les choix qui s'avèrent peu pertinents, comme c'est le cas dans HQ-LEARNING et ALECSYS ;
– la gestion des états cachés par identification de situations clés, comme c'est le cas dans HQ-LEARNING et ZCSM ;
– la gestion de la généralisation et la spécialisation, comme c'est le cas dans ZCSM, U-TREE et ALECSYS.

Une comparaison plus formelle ainsi que des évaluations de performance restent toutefois à approfondir vis-à-vis de toutes les techniques que nous avons évoquées.

Bibliographie

[BEL 57] R. E. BELLMAN. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[BOO 89] L. BOOKER, D. E. GOLDBERG ET J. H. HOLLAND. Classifier systems and genetic algorithms. *Artificial Intelligence*, 40(1-3) :235–282, 1989.

[CLI 94] D. CLIFF ET S. ROSS. Adding memory to ZCS. *Adaptive Behaviour*, 3(2) :101–150, 1994.

[DON 98] J. Y. DONNART. *Architecture Cognitive et Propriétés Adaptatives d'un Animat Motivationnellement Autonome.* PhD thesis, Université Pierre et Marie Curie, Paris, France, 1998.

[DOR 94] M. DORIGO. Genetic and non-genetic operators in ALECSYS. *Evolutionary Computation*, 1(2) :151–164, 1994.

[GOL 89] D. E. GOLDBERG. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, 1989.

[HAN 98] E. A. HANSEN. *Finite Memory Control of Partially Observable Systems*. PhD thesis, University of Massachusetts, Amherst, MA, 1998.

[HOL 75] J. H. HOLLAND. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[KAE 96] L. P. KAELBING, M. L. LITTMAN ET A. W. MOORE. Reinforcement learning : A survey. *Journal of Artificial Intelligence Research*, 4 :237–285, 1996.

[KAE 98] L. P. KAELBING, M. L. LITTMAN ET A. R. CASSANDRA. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101, 1998.

[LAN 98] P. L. LANZI. Adding memory to XCS. In *Proceedings of the IEEE Conference on Evolutionary Computation (ICEC98)*. IEEE Press, 1998.

[LIT 95] M. L. LITTMAN, A. R. CASSANDRA ET L. P. KAELBING. Efficient dynamic-programming updates in partially observable markov decision processes. Technical report CS-95-19, Brown University, 1995.

[MCC 95] R. A. MCCALLUM. *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, Rochester, NY, 1995.

[RAB 86] L. R. RABINER ET B. H. JUANG. An introduction to hidden markov models. *IEEE ASSP Magazine*, Jan. :4–16, 1986.

[WAT 89] C. J. WATKINS. *Learning with delayed rewards*. PhD thesis, Psychology Department, University of Cambridge, England, 1989.

[WIE 97] M. WIERING ET J. SCHMIDHUBER. HQ-learning. *Adaptive Behavior*, 6(2) :219–246, 1997.

[WIL 94] S. W. WILSON. ZCS, a zeroth level classifier system. *Evolutionary Computation*, 2(1) :1–18, 1994.

[WIL 95] S. W. WILSON. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2) :149–175, 1995.