

"Les Québécois" :

Bourgeois Homère
Coubier Raphaël
Gautier Corentin
Odorico Thibault

Encadrants :

Suro François
Ferber Jacques
Lafourcarde Mathieu

TER : Evo Agents

Université de Montpellier II

Année universitaire : 2018 - 2019



MESD@LR



Résumé

Ce TER est à l'origine une proposition de monsieur François Suro, dans le cadre de sa thèse. Le but est ici de réussir à faire émerger chez des robots virtuels des comportements complexes dans le cadre d'un jeu de capture de drapeau. Ces robots, doivent apprendre à agir et réagir en fonction des paramètres qu'ils sont capables de capter au sein d'un environnement. L'idée derrière ce projet est de savoir si les robots sont également capables d'apprendre des comportements au sein d'un système multi-agent, d'interagir entre eux, et d'appliquer une stratégie de groupe. Notre rôle a été de chercher quels comportements apprendre, et développer les expériences d'apprentissage pour les robots, en essayant d'y incorporer des comportements de groupe. Nous avons travaillé à partir d'un projet de base en Java fourni par Mr Suro pour la partie développement, et avons eu accès au cluster MUSE du centre MESO@LR pour pouvoir entraîner les robots en grande quantité et à grande vitesse.

Remerciements

Nous souhaitons remercier Monsieur François Suro pour le sujet proposé, mais également pour sa disponibilité et son aide précieuses durant tout le TER. Nous souhaitons également remercier Mr Jacques Ferber d'avoir encadré ce projet. Enfin, nous souhaitons également remercier le centre MESO@LR de nous avoir donné accès au cluster MUSE, qui a joué un rôle crucial dans l'avancement de notre projet.

Table des matières

Résumé	1
Remerciements	2
Glossaire	5
I) Introduction	6
I.1) But du projet	6
I.2) Système multi-agent	6
I.3) Environnement	6
I.4) Motivations	7
I.5) Etat de l'art	7
I.5.1) Première approche	7
I.5.2) CARACaS.....	9
I.5.3) MAS2CAR	11
I.5.4) Synthèse.....	12
II) MIND	13
II.1) Présentation	13
II.1.1) Architecture de base	13
II.1.2) Différents éléments du modèle	13
II.1.3) Apprentissage génétique	17
II.2) Intérêts du modèle	18
II.2.1) Apprentissage génétique	18
II.2.2) MIND en particulier	19
II.2.3) Limitations	19
III) Stratégies de développement.....	20
III.1) Règles du jeu	20
III.1.1) Initiales	20
III.1.2) Modifications effectué (Règles et robot)	20
II.2) Le robot	21
II.2.1) Composants	21
II.2.2) Intégration de MIND pour le robot	22
III.3) Hiérarchie	22
III.3.1) Cheminement	22
III.3.2) Limitations et modifications	23
III.3.3) Hiérarchie finale	23
IV) Présentation des expériences.....	25

IV.1) Général	25
IV.1.1) Environnement de développement.....	25
IV.1.2) Développement d'une expérience	26
IV.1.3) Observation des résultats de l'expérience	27
IV.2) Flee	27
IV.2.1) But du comportement	27
IV.2.2) Environnement de l'expérience	27
IV.2.3) Expériences complémentaires	28
IV.3) KeepSameSpeedAsTarget	28
IV.3.1) But du comportement	28
IV.3.2) Environnement de l'expérience	28
IV.4) KeepSameOrientationAsTarget	29
IV.3.1) But du comportement	29
IV.3.2) Environnement de l'expérience	29
IV.5) Align	30
IV.5.1) But du comportement	30
IV.5.2) Environnement de l'expérience	30
IV.6.3) Expériences complémentaires	31
IV.6) Flock	31
IV.6.1) But du comportement	31
IV.6.2) Environnement de l'expérience	32
IV.6.3) Expériences complémentaires	33
V) Réflexions	34
V.1) Résultat obtenus	34
V.2) Gestion du projet	35
V.2.1) Organisation	35
V.2.2) Révisions de l'objectif	36
VI) Conclusion	37
Bibliographie	38

Glossaire

IA : Raccourci pour intelligence artificielle

Skill ou compétence : comportement spécifique que l'on souhaite apprendre à un bot, il peut être plus ou moins complexes.

Hardcoded skill : Skill développé en java, on définit manuellement la valeur des sorties en fonction des entrées. Par exemple `out [0] = in [1]` : cela signifie que la deuxième valeur d'entrée du module sera affectée à la première valeur de sortie.

Skill appris : Skill neuraux, c'est-à-dire qui fonctionne via un réseau de neurones.

Sub skill : Skill qui compose un skill plus complexe, c'est à dire supérieur hiérarchiquement. Ainsi un skill complexe C1 est un sub de son skill plus complexe, le skill Master par exemple.

Base skill : skill avec une ou plusieurs sorties vers les actionneurs.

Sensor / Capteur : partie du robot qui lui permet de récolter des informations sur son environnement

Actuator / Actionneur : partie du robot qui lui permettent d'agir dans son environnement.

Bot : abstraction informatique d'un véritable robot, comportant sa propre intelligence artificielle, ainsi que des capteurs et actionneur fonctionnant dans les simulations, similaires dans leur fonctionnement à ceux d'un robot physique.

Tick : unité de temps qui sépare deux états consécutifs d'une simulation

I) Introduction

I.1) But du projet

Dans le cadre de la programmation multi-agent, nous devons élaborer une stratégie pour gagner contre des bots implémentés en code Java dans un environnement simulé (EvoAgent). Le but de la simulation est de gagner le plus de point en ramenant le drapeau de l'adversaire depuis la base ennemie jusqu'à leur propre base, en affrontant les robots de l'équipe adverse. Les agents peuvent attaquer et se défendre en tirant sur les ennemis, et doivent se déplacer sur une carte comportant divers obstacles. Lors de notre TER, il fallait donc entraîner une équipe de robots, leur apprendre des compétences de base, les recombinaison en capacités plus complexes afin qu'ils puissent à terme travailler en équipe et effectuer des manœuvres en fonction de leur environnement. Nous avons donc défini une hiérarchie de comportement et développé les environnements d'apprentissage et les fonctions de récompenses pour apprendre cette stratégie aux robots. Le but ultime étant que ces robots puissent par la suite affronter l'équipe d'agent par défaut du logiciel EvoAgent.

I.2) Système multi-agent

L'idée était, durant ce projet, de se concentrer sur l'apprentissage de robots au sein d'un système multi-agent. Un système multi-agent est un système qui comprend différents agents (ici les robots), chacun avec leur intelligence artificielle propre, mais qui évoluent au sein d'un environnement en groupe. Les systèmes multi-agent voient alors les agents se mettre en relation, agir en commun vers un même but, se répartir des tâches, etc... Cela soulève différentes problématiques vis-à-vis des capacités que les robots doivent être capables de développer. Par exemple, comment les robots vont-ils analyser leur environnement, et quelle réaction vont-ils adopter ? Pourront-ils réussir à s'organiser avec les autres robots de leur équipe via une méthode de communication ? Simplement en fonction de la situation ? Ce sont ce type de préoccupations à propos des systèmes multi-agents qui sont réellement au cœur de ce sujet de TER.

I.3) Environnement

Le projet se réalise à partir d'un environnement programmé par Monsieur François Suro dans le cadre de son doctorat sur la création d'un modèle de hiérarchie de comportement pour l'apprentissage des robots. Le code est en Java, et implémente toute la structure nécessaire au développement de nouveaux environnement et fonctions de récompense indispensables pour le projet. Par ailleurs, nous avons eu accès au cluster MUSE du centre MESO@LR afin de pouvoir effectuer des entraînements plus nombreux et plus poussés pour les robots. L'entraînement à grande échelle des robots est une composante cruciale du développement de leur intelligence artificielle. Ces ressources mises à notre disposition ont joué un rôle essentiel dans l'accomplissement du projet.

1.4) Motivations

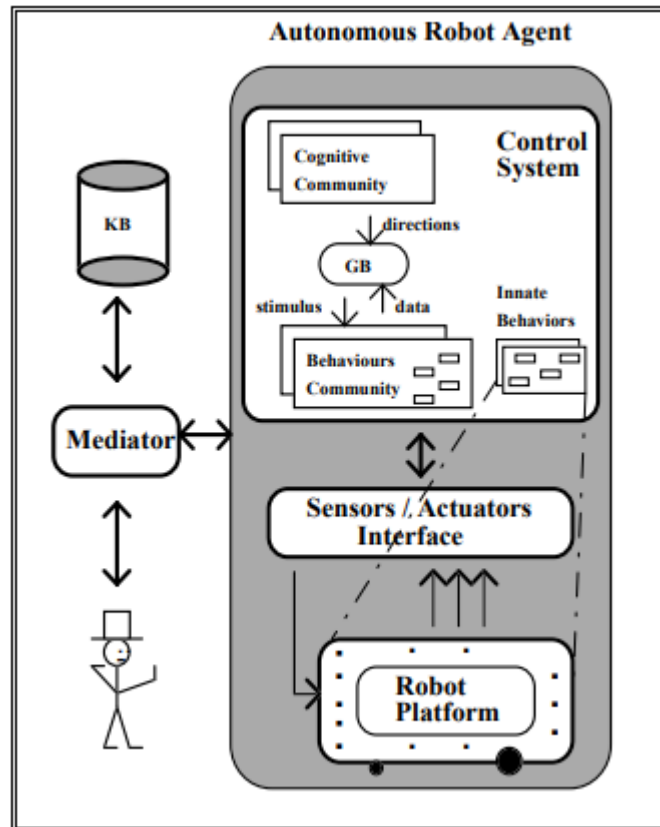
Le projet nous a intéressé dès l'apparition de la proposition. L'idée d'entraîner des robots dans un environnement ludique (un jeu de capture de drapeaux) nous a séduit, et nous avons cherché à nous impliquer très tôt dans le projet. Le domaine de l'intelligence artificielle, et particulièrement les systèmes à base de réseaux de neurones, qui sont au cœur de ce projet, sont un sujet aujourd'hui à la pointe de l'innovation informatique. L'idée de pouvoir s'investir dans un sujet abordant un domaine pointu de notre secteur d'études a fini de nous convaincre d'essayer de participer à ce projet. Nous avons déjà eu l'occasion de développer et d'entraîner une intelligence artificielle basée sur un réseau de neurones durant notre scolarité, et souhaitons réitérer l'expérience dans le cadre d'un projet plus ambitieux.

1.5) Etat de l'art

Le développement d'une structure de hiérarchie comportementale est un domaine de recherche très actif depuis les années 90, avec la démultiplication des applications possibles pour les robots autonomes. Plusieurs modèles ont émergé, chacun avec leurs spécificités propres, dans des laboratoires du monde entier, dans le but de remplir différentes missions. Les hiérarchies doivent être adaptées aux conditions dans lesquels ces robots évolueront, que ce soit en simulation ou sur de vrais robots, seul ou en groupe, etc...

1.5.1) Première approche

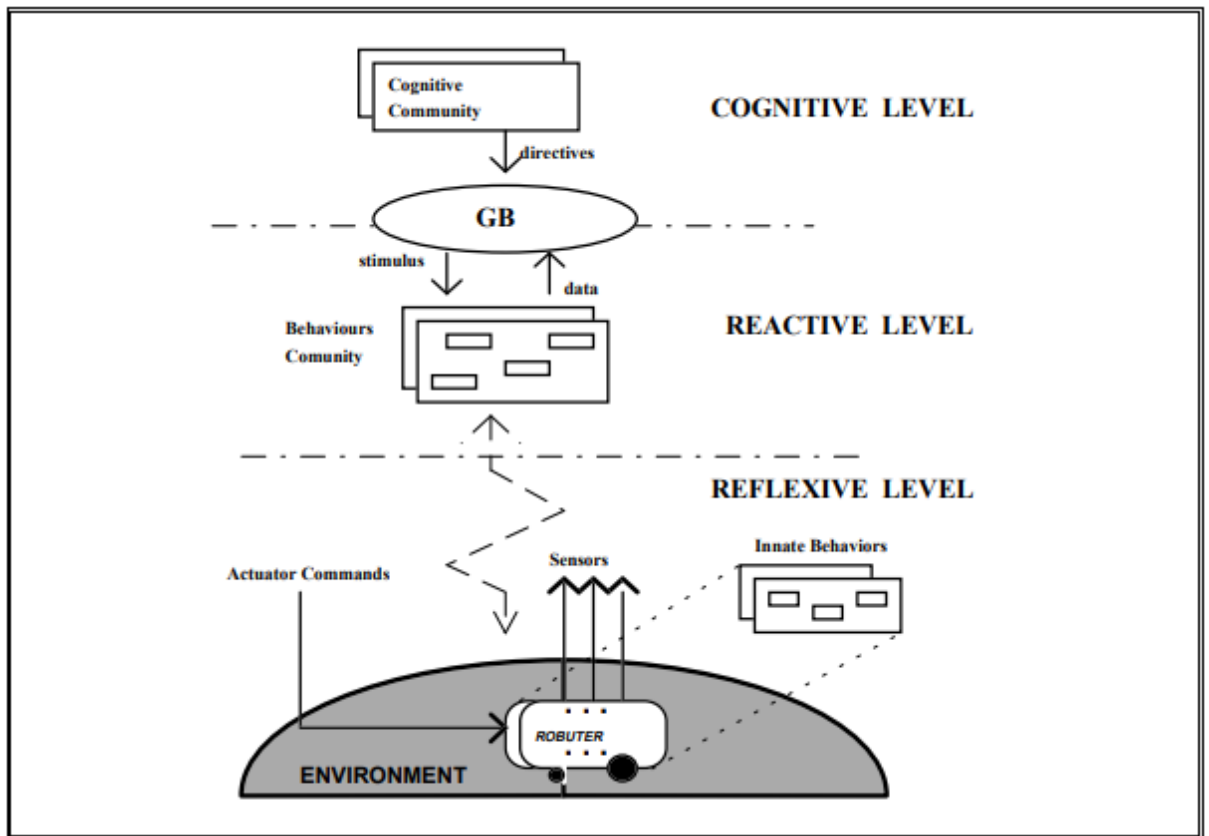
Une approche proposée par Maria C. Neves et Eugénio Oliveira en 1997 consiste en une architecture liant étroitement perception et actions du robot. Leur modèle intègre à la fois une architecture réactive basée sur les perceptions immédiates du robot, mais également une structure de réflexion basée sur des connaissances préexistantes. Le modèle repose sur un système de comportements qui sont amenés à se combiner de différentes manières pour permettre au robot de réagir au mieux selon les variations de son environnement, voir même à apprendre de nouveaux comportements de manière autonome. Il est également possible au robot de se référer à une base de connaissance qui ne soit pas directement implantée dans sa base de comportements innés, ou de faire appel à une intervention humaine en cas de besoin via un composant "médiateur".



Structure de l'Agent Robot Autonome

Pour ce qui est de l'architecture de contrôle du robot à proprement parler, elle se décompose en trois parties qui interagissent ensemble. La plus haute couche de cette hiérarchie, appelée "cognitive level", est chargée de donner les indications sur la marche à suivre au robot à partir d'informations remontées par les autres couches du système, ou de plans d'actions intrinsèques qui vont guider les opérations du robot. L'échange d'informations se fait via le "Green Board Agent" (GB) qui est chargé de faire l'intermédiaire entre les couches réactives et cognitives, en stockant des informations de la couche inférieure, et en envoyant des stimulus vers cette dernière pour orienter les modules du niveau réactif sur la marche à suivre.

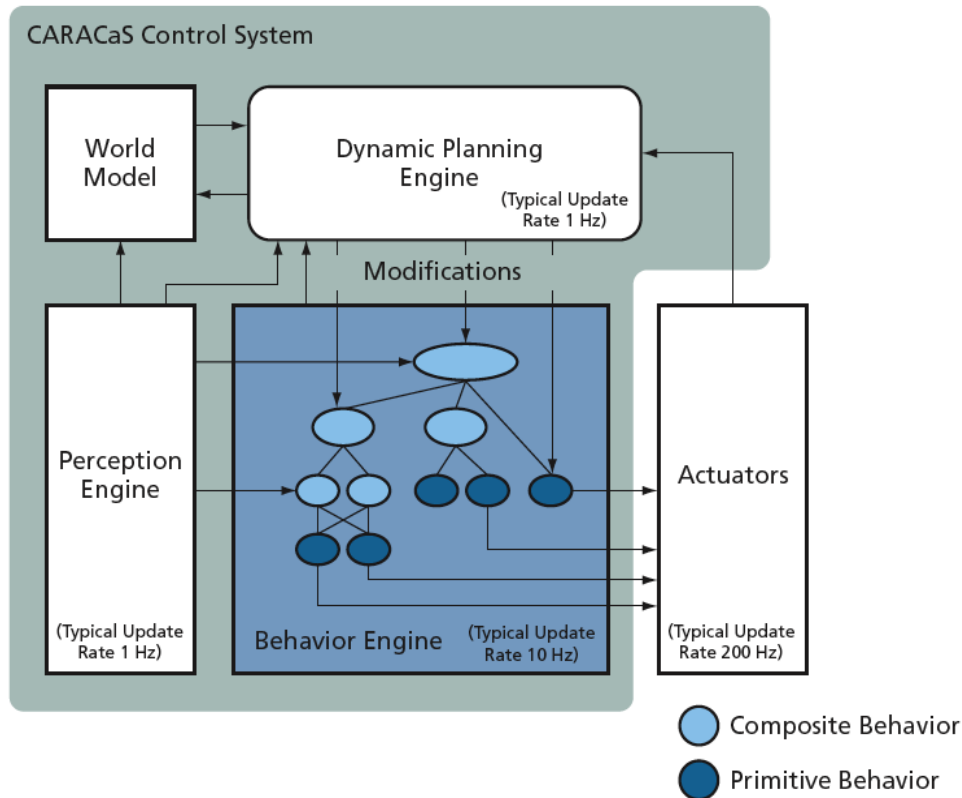
Ces modules qui appartiennent à la "Behavior Community" se rapprochent des travaux de Dorigo et Colombetti qui ont servi de modèle au fonctionnement de MINDS. Il s'agit de faire interagir des modules de comportements afin d'obtenir un résultat adéquat en sortie. Ces comportements sont de différents niveaux, et fonctionnent en hiérarchie. A la base de cette hiérarchie se trouvent des skills simples, corrects et indépendants des autres skills simples. Chacun doit pouvoir effectuer sa tâche de manière autonome. Ils se combinent grâce à des opérateurs de combinaison, d'exclusion, de condition... afin d'obtenir des skills de niveaux supérieurs qui eux-mêmes peuvent se recombinaient grâce aux mêmes opérateurs... Les skills prennent en entrée des informations de capteurs ou des stimulus venu du GB pour déterminer l'information en sortie. On retrouve encore un système similaire à MINDS dans son principe.



Architecture du système de contrôle

1.5.2) CARACaS

Le modèle Control Architecture for Robotic Agent Command and Sensing (CARACaS) est un modèle mis au point par la NASA en 2008. Celui-ci a pour but principal de permettre à de nombreux systèmes robotiques de pouvoir évoluer de manière autonome, qu'ils soient seuls ou au sein d'un groupe. Pour ce modèle, l'accent est mis sur l'adaptabilité du robot, et sa capacité à prendre des décisions au sein de son environnement. Le modèle présenté vise à être implémenté sur un véritable robot, et non pas dans une simulation informatique comme c'est le cas pour nous. Ainsi, la fréquence de mise à jour est un point d'attention crucial pour l'équipe de développement.



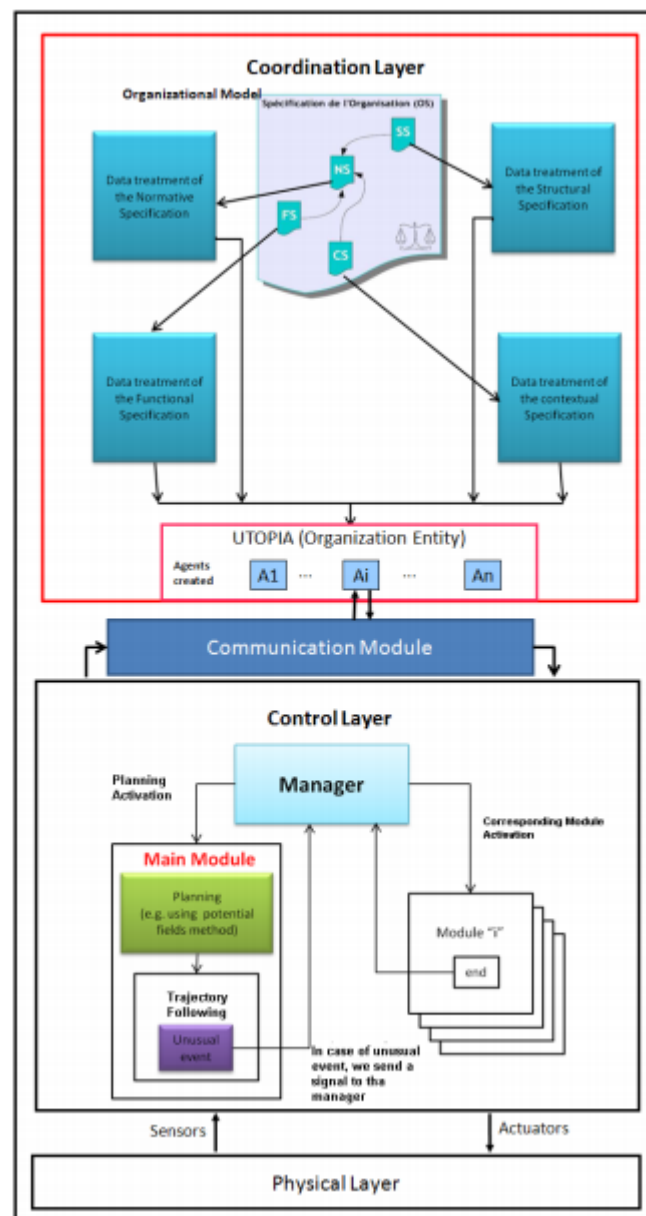
Structure du modèle CARACaS

La structure hiérarchique du robot dispose d'une grande partie dédiée à l'analyse de l'environnement et à la prise de décision. Le "World Model" sert de mémoire au robot, dans laquelle il peut stocker ou récupérer des informations sur son environnement. Le "Perception Engine" gère les informations des capteurs, et influe sur la planification des comportements futurs, décidée par le "Dynamic Planning Engine". Ces deux parties modèlent les informations à placer dans la mémoire, et vont paramétrer le "Behavior Engine". Ce dernier va déterminer l'action que doit effectuer le robot dans l'immédiat, et transmettre actionner les commandes motrices du robot.

Dans ce modèle, l'accent est mis sur la prise de décision en adéquation avec l'environnement. Deux moteurs sont pleinement dédiés à la récupération, au traitement et au stockage de l'information, au sein d'une stratégie globale de comportement. Le "Dynamic Planning Engine", au cœur de la structure, s'occupe de cette analyse en fonction des informations données et celle mises en mémoire, mais également des rétroactions du Behavior Engine et des actionneurs. Bien que conçue pour des robots autonomes, on peut imaginer que cette hiérarchie puisse également gérer des informations transmises entre les robots dans son "Perception Engine", ajuster son comportement en conséquence, et transmettre d'autres informations en retour de manière réactive.

1.5.3) MAS2CAR

En 2011, une équipe de chercheurs propose un modèle dédié entièrement tourné vers le multi-agents nommé MultiAgents System to Control and Coordinate teAmworking Robots (MAS2CAR). Ce système, décomposé en trois couches, permet de gérer la communication entre robots grâce à sa couche de coordination, qui s'occupe de la communication entre les robots, et de l'organisation de tâches de groupe impliquant plusieurs robots. Ces différents robots peuvent être de types différents puisque la couche de coordination est abstraite, et indépendante du hardware des robots. Les deux autres couches, qui sont cette fois-ci propre à chaque robot, s'occupent du contrôle individuel et de la gestion des composants physiques du robot.



Structure du modèle MAS2CAR

Ce qui fait la spécificité de ce modèle, c'est bien la couche de coordination, qui agit comme un agent décisionnel au sein d'un groupe de robot. Elle peut être perçue comme un intermédiaire entre les agents du système, chargé à la fois de les coordonner dans l'accomplissement de tâches communes, et de traiter leurs requêtes individuelles envers d'autres robots. Les robots peuvent procéder à l'envoi de message vers cette couche, qui s'occupe ensuite de traiter le message et d'envoyer aux robots concernés un ordre / une information en conséquence. Cette couche de supervision s'occupe également de gérer la détection des robots les uns par rapport aux autres, et les potentiels collisions entre les agents, en parallèle de la gestion de message.

Les deux autres couches de l'architecture sont quant à elle plus classiques. La couche de contrôle s'occupe du calcul des actions nécessaires à l'accomplissement de la tâche en cours, et fait remonter des informations à la couche de coordination pour déterminer l'action à adopter par la suite. La présence d'un obstacle imprévu va par exemple être communiquée à la couche de supervision du robot, qui va lui demander d'effectuer en réaction une manœuvre d'esquive, en dépit de l'action planifiée qu'il exécutait jusque-là. Cet ordre sera transmis à la couche de contrôle via le module de communication, et le robot adaptera sa trajectoire. La couche de contrôle se chargera de faire les nouveaux calculs de trajectoire, et la dernière couche, la couche physique, transmettra les ordres aux différents moteurs du robots.

1.5.4) Synthèse

Ces trois modèles constituent des exemples de système capables d'entraîner des robots à des tâches complexes. Chacun permet à la fois au robot de prendre une décision et d'agir en conséquence, et agissant sur une hiérarchie plus ou moins complexe. Elles essaient au mieux de répondre à un problème donné, que ce soit l'intégration au sein d'un système multi-agent, ou l'implémentation sur un véritable robot. Chacune de ces hiérarchies a pris le parti de créer une couche spécifique de modules dédiés à l'activation de capacités physique du robot, qui dépend d'une hiérarchie de prise de décision beaucoup plus complexe. L'apprentissage génétique semble également être une constante, puisque tous ces modèles visent à faire évoluer des robots dans un environnement qu'ils ne connaissent à priori pas, ou très peu, et de leurs faire apprendre un ou plusieurs comportements leur permettant la réalisation de différentes tâches complexes. Ainsi, le modèle MIND présenté par la suite, qui est une alternative à ces modèles que nous allons utiliser tout au long de ce projet, reprend également les grands principes de ces modèles proposés auparavant.

II) MIND

II.1) Présentation

II.1.1) Architecture de base

Comme évoqué précédemment, l'intelligence artificielle des robots se base sur un modèle spécifique nommé MIND. Ce modèle cherche à simuler le cheminement de l'apprentissage de tâches complexes, agencant en arbre des compétences de plus en plus complexes à mesure que l'on remonte les branches. Le but de cette organisation est d'obtenir des modules d'actions génériques, capables de fonctionner indépendamment des modules plus complexes qui peuvent l'utiliser. Ces modules pourront à tout moment être ré-entraînés individuellement sans que cela n'impacte les modules fils qui en dépendent, voir remplacer par des substituts incorporant les comportements souhaités codés (et non plus entraînés et appris) à tout moment. Enfin, l'ajout de nouvelles capacités (nouvelles compétences ou capteurs) au robot n'impacte pas la hiérarchie existante, mais peut se faire en ajoutant des branches de modules, pouvant se baser sur des skills de base, sans avoir à modifier les skills plus complexes préexistants.

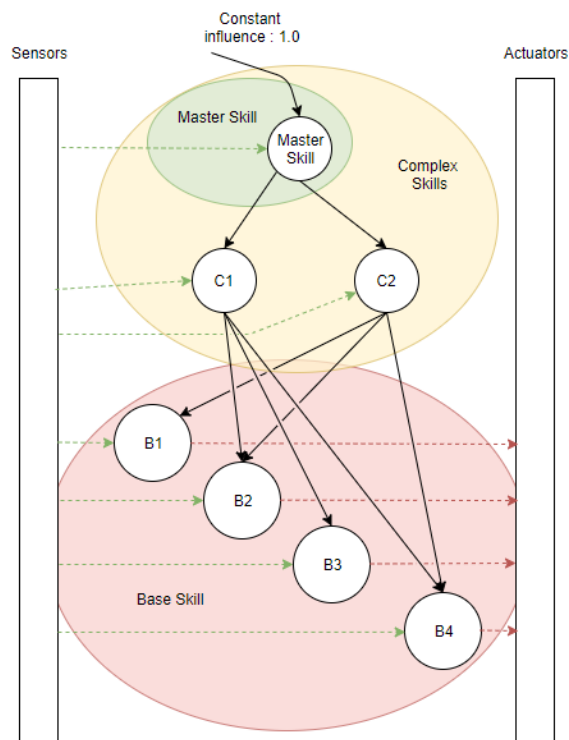
II.1.2) Différents éléments du modèle

Le modèle s'articule autour de différents éléments qui doivent donc interagir entre eux afin de produire une intelligence artificielle avec un comportement complexe et cohérent par rapport à son environnement. Un skill est un module dédié à une tâche spécifique. Il encapsule une fonction prenant en entrée un certain nombre d'informations provenant soit de variables alimentées par un autre skill ou d'un capteur du robot par exemple. Cette fonction est basée sur un modèle vectoriel : elle associe un vecteur V_e d'informations d'entrée à un vecteur V_s d'informations de sortie.

$$V_s = f(V_e)$$

Chaque skill, lors de son entraînement, va ajuster la fonction interne pour affiner son comportement et obtenir un meilleur résultat. A terme, on a donc une fonction qui est la somme de tous les ajustements opérés lors des entraînements sans que l'on sache exactement ce qui a été appris.

Si un module actionne directement les moteurs du robot, on le nomme "base skill". Ce sont des compétences souvent simples que l'on retrouve en général dans le bas de la hiérarchie. Les skills qui ont en sortie d'autres skills sont nommés des "complex skills", et on appelle "sub skills" les skills de sortie. Le complex skill envoie donc en sortie une influence, qui va définir l'importance accordée à tel ou tel sub skill, qui se servira de la valeur reçue pour le calcul de sa fonction et donc de ses propres valeurs de sortie. On appelle vecteur d'influence V_{infl} l'ensemble des valeurs de sortie. On peut alors voir apparaître des hiérarchie complexes de skills, avec au sommet de cette hiérarchie un unique "master skill" qui reçoit en entrée une influence de 1.0.



Catégorisation des modules d'une hiérarchie

A partir du master skill, chaque complex skill calcule son vecteur de sortie V_s et multiplie chaque élément par la somme des influences reçues, formant le vecteur d'influence V_{infl} . Le skill envoie alors chaque élément $Infl_x$ de V_{infl} au subskill correspondant (Fig.3) :

$$V_{infl} = V_s * \sum_{x=1}^n Infl_x$$

(Eq.2)

Avec V_{infl} le vecteur d'influence vers les subskills, $V_s = f(V_E)$ le vecteur de sortie de la fonction interne du skill, et $\sum_{x=1}^n Infl_x$ la somme de toutes les influences reçues par le skill (également noté $\Sigma Infl$).

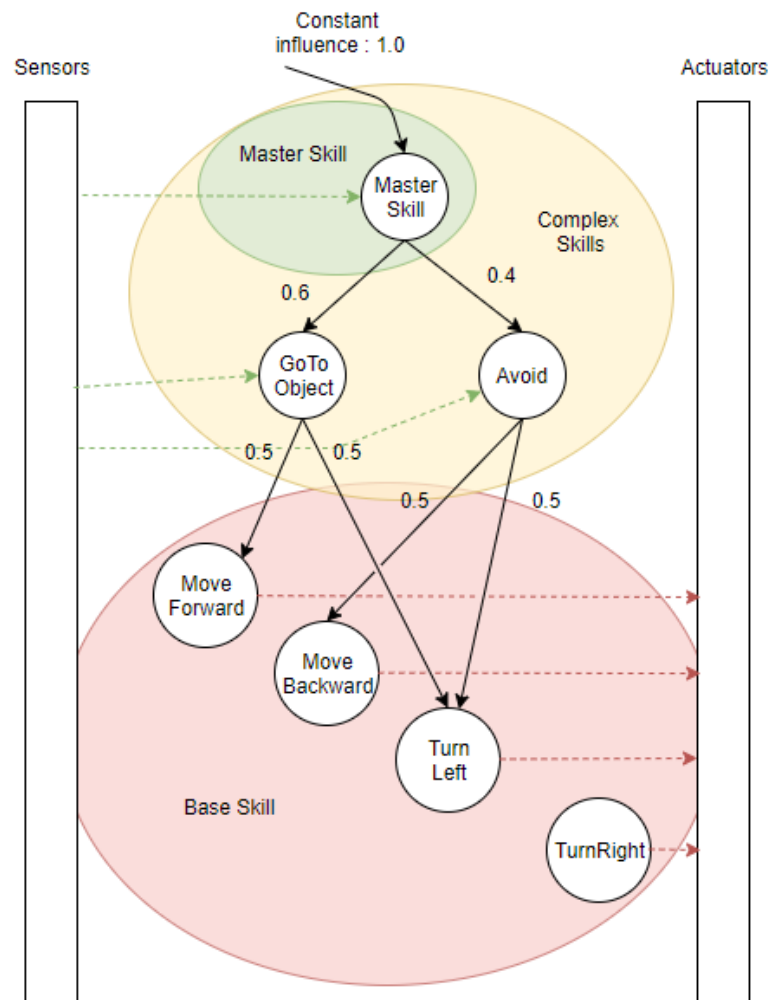
Un base skill, comme un complex skill, calcule son vecteur de sortie et multiplie chaque élément par la somme des influences qu'il a reçues (Eq.2), formant le vecteur de commandes moteur $V_{com} = [Com_1, Com_2, \dots, Com_m]$. Le base skill envoie alors chaque élément Com_x du vecteur de commandes moteur au module moteur correspondant, ainsi que la somme des influences reçues par lui-même $\Sigma Infl$. Chaque module moteur calcule ensuite la commande correspondant à son actionneur sous la forme d'une somme pondérée normalisée :

$$M = \frac{\sum_{x=1}^n Com_x}{\sum_{x=1}^n \Sigma Infl_x}$$

(Eq.3)

Avec M la commande de moteur résultante, x l'index de la compétence de base qui envoie une commande de moteur, Com_x la commande de moteur pondérée pour ce module moteur depuis le base skill x , $\Sigma Infl_x$ la somme des influences du base skill x .

Par exemple, sur la hiérarchie ci-après, on a un master skill qui distribue une influence de 0.6 et 0.4 à ses deux complex skills. Ceux-ci envoient leurs commandes aux skills qui activent les moteurs pour faire se déplacer le robot dans les quatre directions (avant arrière gauche et droite). Chacun de ces skills détermine lui aussi la valeur qu'il va transmettre en sortie à partir de l'influence reçue.



Exemple de hiérarchie simple

- GTO : $V_{infl} = 0.6 * [0.5, 0, 0.5, 0] = [0.3, 0, 0.3, 0]$
- Avoid : $V_{infl} = 0.4 * [0, 0.5, 0.5, 0] = [0, 0.2, 0.2, 0]$

On a les vecteurs de commande V_{com} des base skills suivants :

- moveForward : $V_{com} = [1, 1]$
- moveBackWard : $V_{com} = [0, 0]$
- moveLeft : $V_{com} = [0, 1]$ (moteur de droite en marche)
- moveRight : $V_{com} = [1, 0]$ (moteur de gauche en marche)

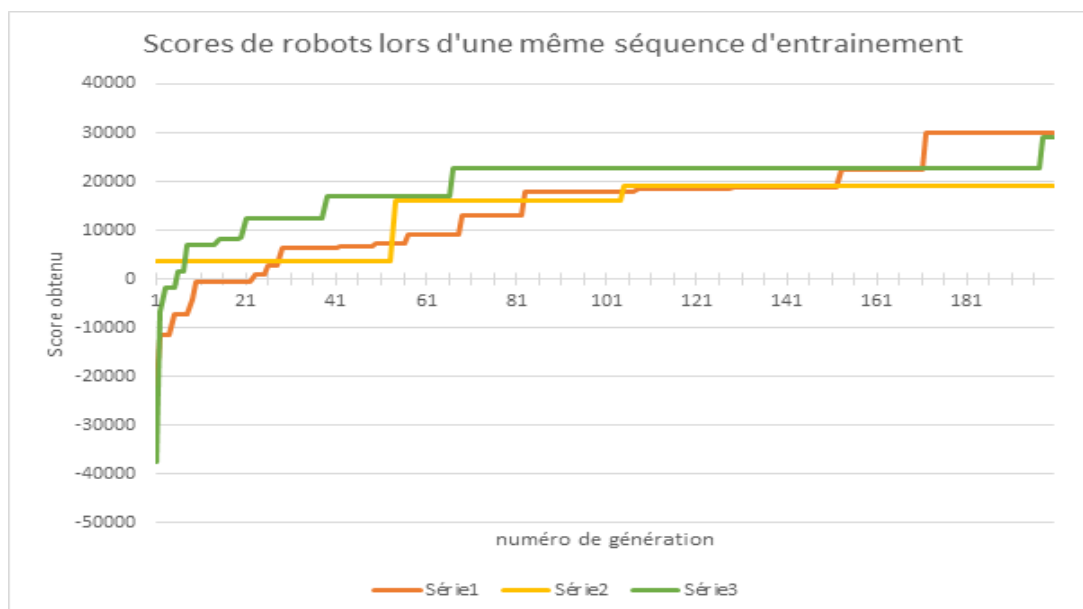
Et donc pour chaque moteur les commandes M de chacun des deux moteurs :

- $M_{left} = 0.6$
- $M_{right} = 1$

Ce qui correspond à un virage à gauche.

II.1.3) Apprentissage génétique

L'entraînement des skills se fait grâce à un apprentissage génétique. L'apprentissage génétique est une méthode algorithmique qui peut être utilisée pour trouver une solution approchée à un problème d'optimisation auquel on ne connaît pas forcément la solution, ou pour lequel il n'existe pas de méthode exacte pour l'atteindre. Ce sont des algorithmes très populaires dans le domaine du Machine Learning. Ils permettent aux IA d'expérimenter un comportement dans leur environnement pour résoudre un problème donné, et dans lequel elles seront évaluées selon différents critères (temps pour trouver la solution, consistance de la solution trouvée...). On se base ensuite sur le résultat trouvé par une IA pour relancer un nouvel entraînement dans le but d'améliorer le résultat trouvé. Si le nouvel entraînement produit un meilleur résultat, il servira de référence pour relancer un entraînement, sinon il sera rejeté. Le résultat progresse ainsi par bond de performance, grâce à une évolution de son réseau de neurones, qui rapproche le robot d'une solution optimale au fur et à mesure que les entraînements se multiplient. En cela, ces méthodes algorithmiques se rapprochent du fonctionnement de la génétique, puisque l'on crée des individus, et les mieux adaptés (en l'occurrence ceux qui ont trouvé la meilleure solution au problème posé) sont conservés et servent de base à une nouvelle génération. On observe en générale des progrès très rapides sur les premiers entraînements, puis les améliorations se font de plus en plus rares au fur et à mesure que les IA progressent.



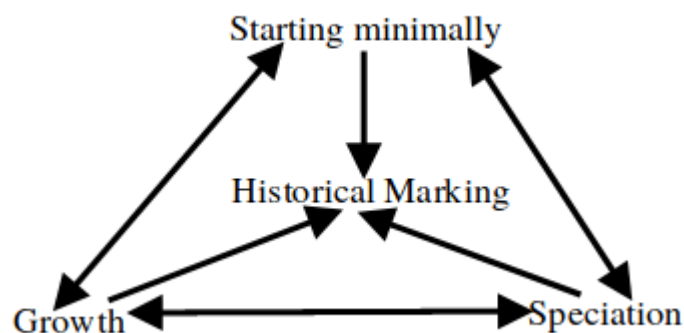
Présentation de scores sur 200 générations de trois séries d'entraînements identiques

II.2) Intérêts du modèle

II.2.1) Apprentissage génétique

Dans le cadre de notre projet, l'apprentissage génétique est une solution adéquate puisque les comportements à apprendre n'ont pas forcément de solution fixe. Les prises de décisions des robots sont conditionnées par un grand nombre de paramètres qu'il serait très difficile de gérer grâce à un autre type d'architecture de comportement (machine à état, subsomption, etc...) compte tenu de la taille des hiérarchies de comportements que l'on souhaite mettre en œuvre. L'apprentissage génétique, via l'ajustement du réseau de neurones durant les phases d'apprentissage, encapsule toute cette partie d'ajustement de variables, trop laborieuse à faire par des moyens plus classiques. De plus, l'apprentissage génétique permet de voir apparaître des comportements émergents, qui peuvent répondre au problème posé, sans qu'elles aient été envisagées par les chercheurs. Cette propension à l'innovation et à l'émergence de solutions inattendues fait prévaloir d'autant plus l'apprentissage génétique.

L'algorithme utilisé pour l'entraînement du réseau de neurones est l'algorithme NeuroEvolution of Augmenting Topologies (NEAT). C'est un algorithme de neuro-évolution qui est donc utilisé dans le projet pour configurer le réseau de neurones. En plus de paramétrer les entrées et sorties du réseau de neurones en interne, NEAT gère également la topologie du réseau (ajout ou suppression de neurones, connexions entre les neurones), sans qu'une intervention humaine n'ait lieu. Le réseau optimise les liens et les couches de neurones en évaluant le réseau actuel par rapport à différents critères. Si le réseau génère une innovation dans le comportement, cette innovation doit être conservée dans les futurs réseaux. L'algorithme conserve également un historique des modifications effectuées et, partant d'une structure minimale, augmente incrémentalement le nombre de neurones pour optimiser au maximum la topologie du réseau, et le rendre moins gourmand en ressources. NEAT est également intéressant dans notre cas puisqu'il peut fonctionner sans qu'il y ait une base de données pour entraîner le réseau de neurones. Dans notre cas, l'entraînement se fait par expérimentations successives, sans qu'on ait de stock de données pour apprendre le comportement, comme dans d'autres processus de Machine Learning, et NEAT permet également de faire cela.



Dépendances des composants NEAT

II.2.2) MIND en particulier

Le principal intérêt du modèle MIND est la modularité de sa hiérarchie. Chaque branche de compétence peut être déplacé dans la hiérarchie (à condition d'avoir des entrées identiques) et garder la même efficacité sans avoir à être ré-entraîné. Les subskills d'une compétence peuvent être changés sans que cela affecte le haut de la hiérarchie. Il est donc possible de réajuster les composants de sortie sans avoir un impact global sur tout le comportement du bot, et donc sans devoir réapprendre l'intégralité des comportements, comme cela peut être le cas pour certaines hiérarchies de contrôle. De la même manière, il est possible de greffer ou supprimer toute une branche nouvelle de compétence à une hiérarchie préexistante, en ayant simplement à ré apprendre la compétence liée au module où se fait le rajout / la suppression.

La gestion des informations que perçoit un robot fait également de MIND un système intéressant. Il est en effet possible pour le système de stocker des informations pour être réutilisée par d'autres skills, grâce à un système de variable interne. Ces variables peuvent être utilisées telle quelles, ou alors via leur dérivée, permettant au robot de traiter un bon nombre d'informations. Ces variables peuvent être alimentées par des skills situés haut dans la hiérarchie, et utilisés par des skills plus bas, qui peuvent ne pas être directement reliés dans la hiérarchie.

II.2.3) Limitations

L'une des faiblesses du modèle est l'impossibilité pour une action en bas de la hiérarchie de rétroagir sur les modules supérieurs. L'influence ne se propage qu'en descendant dans la hiérarchie, et un module qui ne pourrait pas agir suffisamment pour éviter un problème à un tick donné ne pourrait pas le "signaler" (impacter l'influence de skill supérieur pour recevoir plus d'influence) à des skills supérieur pour le prochain tick de la simulation. Un exemple simple : l'évitement d'un obstacle. Si la valeur accordée au skill d'évitement d'obstacle, il existe des modèles de hiérarchies dans lesquelles le skill pourrait influencer les modules supérieurs pour recevoir une plus grande partie des ressources par la suite et réussir à éviter l'obstacle malgré tout. Dans MIND, un tel retour d'influence est impossible.

Dans le cadre d'un système multi-agents, il existe également des modèles de hiérarchie qui incorpore une structure toute dédiée à la communication entre les robots. Ici, aucune structure similaire n'existe. Néanmoins, les bots que nous allons entraîner disposent de trois émetteurs et de trois récepteurs de signaux différents, qui peuvent donc leur permettre de s'envoyer des messages simples. Il est donc théoriquement possible d'entraîner des skills de communication qui influencerait les skills d'actions de la hiérarchie. On approcherait alors les comportements que l'on peut avoir dans des systèmes multi-agents simples, avec des communications entre les robots, et potentiellement des comportement complexes de travail en équipe, de répartition des tâches, etc...

Une autre composante du modèle qu'il reste à définir est la capacité du robot à prendre des décisions. Pour l'instant, seul le master skill est à même de prendre la décision sur l'action à effectuer.

III) Stratégies de développement

III.1) Règles du jeu

III.1.1) Initiales

Le but d'EvoAgent étant de gagner une capture de drapeau nous devons clarifier toutes les règles du déroulement d'une partie ainsi que les paramètres des agents afin d'avoir avoir toutes les données nécessaires pour un apprentissage correct. Parmi ces règles certaines étaient déjà intégrées dans EvoAgent :

- Règle principale : récupérer le drapeau et le ramener à la base.
- Condition de victoire ; avoir le plus de points au bout d'un certain temps de jeu.
- Le drapeau apparaît au milieu des bases des deux camps.
- L'environnement où se déplace les agents est rectangulaire, de plus des obstacles sont placés aléatoirement sur la carte.
- Les agents peuvent tirer, et meurt en un seul coup reçu.

Toutefois cela ne suffisait pas à définir entièrement les paramètres d'apprentissage. Il nous fallait plus de modifications sur les sensors des agents et des règles d'une partie.

III.1.2) Modifications effectuées (Règles et robot)

Après une réunion sur la discussion des modifications à effectuer nous avons fait une liste pour que cela soit plus agréable à gérer par notre référent Monsieur Suro car, par la suite, c'était lui qui s'occupait des changements.

- Modification de la vitesse (ralenti) de l'agent lorsqu'il attrape le drapeau.
- Ajout du sensor du barycentre d'un groupe d'agent, utilisé pour le mouvement de flocking
- Pas plus de dix agents par équipe et lorsqu'un agent meurt, il ressuscite autour de la base de son équipe.

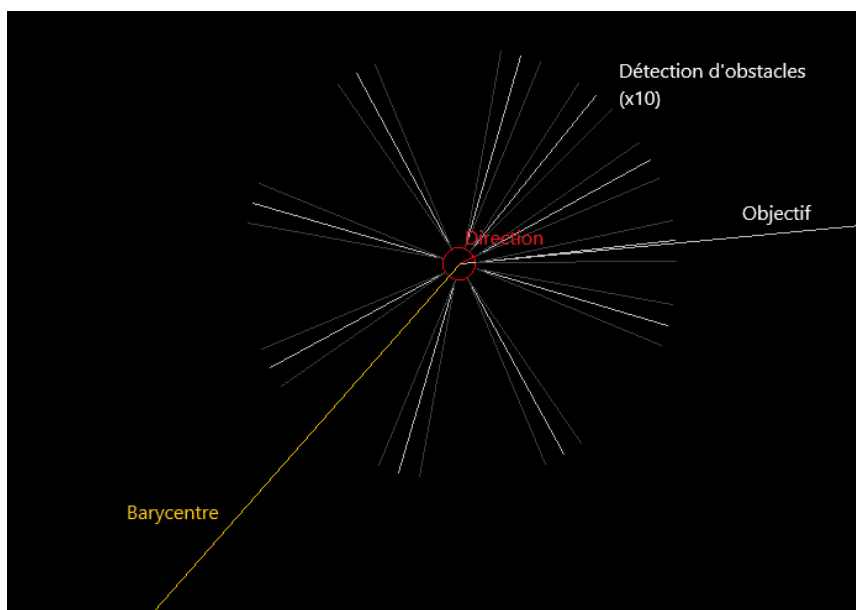
Après ces modifications, nous avons un environnement adéquat pour le développement d'un mouvement de flocking dans EvoAgent.

II.2) Le robot

II.2.1) Composants

Les bots que nous avons entraînés disposent de plusieurs composants, qui lui servent à remplir deux rôles majeurs : percevoir son environnement et agir en conséquence. La première catégorie regroupe donc les capteurs du robot. Ceux-ci lui permettent de récolter toutes sortes d'informations sur les robots qui l'entourent, les objectifs qu'il doit atteindre, et la topographie de la carte. Les capteurs servant à avoir des informations sur les robots et objectifs ennemis ne sont pas les mêmes que ceux dédiés aux alliés. Chaque capteur récupère l'information consacrée à chaque tick de la simulation. Il est également possible d'obtenir la dérivée de chacun d'entre eux afin d'obtenir les variations des valeurs. Par exemple pour la position d'un ennemi qui se déplace, on récupèrera sa vitesse et donc il sera possible d'anticiper sa trajectoire. Les capteurs qui lui permettent de détecter les obstacles du terrain sont au nombre de dix, et sont tous orientés dans une direction différente. Il y en a 8 disposés en cercle autour de lui, et deux autres qui sont orientés entre le premier et le deuxième, et entre le huitième et le premier, pour affiner la perception du robot sur ce qui se trouve en face de lui. Il dispose également de trois canaux de réception pour des signaux envoyés par d'autres robots.

Ces signaux peuvent être envoyés grâce à des actionneurs. Les autres actionneurs dont dispose le robot lui servent à contrôler ses deux moteurs (gauche et droite) qui lui permettent d'avancer, ainsi que les éléments de son canon : la gâchette, l'orientation du canon et son rechargement.



Représentation de capteurs du robot en simulation

II.2.2) Intégration de MIND pour le robot

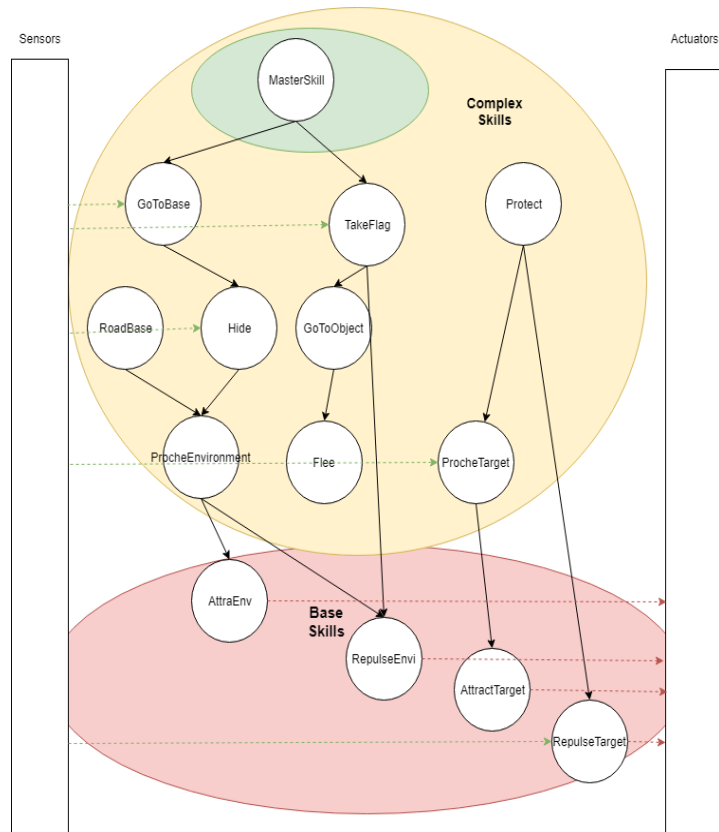
Dans l'architecture MIND, chaque capteur peut être une entrée potentiel d'un skill, et fournir une information pour que le comportement associé au module se réalise correctement. Les informations sur les capteurs peuvent être configurée par le code pour désigner à quels éléments du monde les informations perçues par le robot se rapportent. Les informations obtenues peuvent soit être utilisées directement par le skill, soit être stockées dans des variables pour que d'autres skills dans la suite de la hiérarchie puissent les utiliser. Les moteurs quant à eux peuvent être soit activés par des skill hardcodés, comme c'est le cas dans ce projet, ou bien directement par un skill appris. C'est principalement l'influence reçue en entrée qui déterminera l'ampleur de la commande moteur envoyée par ces skills.

III.3) Hiérarchie

III.3.1) Cheminement

Pour le développement d'un apprentissage correct nous devons tout d'abord être d'accord sur une bonne stratégie pour un meilleur résultat recherché. La première partie de notre TER était basé sur la conception d'une stratégie gagnante pour battre l'équipe intégrée dans EvoAgent. Comme vue précédemment, le déroulement de nos réunions avec notre référent était de lui apporter une hiérarchie de skill correct pour une stratégie de jeu. Il nous a fallu un temps d'adaptation pour bien comprendre le fonctionnement des hiérarchies ainsi que pour en produire des correctes.

Voici un des exemples que nous avons pu développer lors de nos réunions. Nous nous sommes concentrés, au début, sur une hiérarchie globale, c'est à dire une stratégie complète. Toutefois, dans cet exemple, nous sommes venus à la conclusion que certain skill développé pouvait être rassemblé en un skill et être réutilisé. Il nous fallait donc recommencer et en proposer une nouvelle. Cet exemple décrit la complexité de créer une stratégie qui convient au résultat souhaité.



Exemple de proposition de hiérarchie

III.3.2) Limitations et modifications

La complexité de créer une hiérarchie de skill est de trouver la meilleure stratégie possible pour un résultat voulu. Lors de nos schémas nous n'étions pas sûr de ce que pouvait donner une stratégie après l'apprentissage. De plus cela permet aussi d'observer des comportements émergents non voulu après avoir suivi un schéma. Un exemple de comportement émergent est après un apprentissage du skill GoToTarget, l'agent pour aller le plus rapidement à la cible ne faisait pas une ligne droite vers celui-ci comme voulu mais allait en direction de la cible mais de manière circulaire. Il avait appris qu'utiliser qu'un des moteurs de roue était plus efficace ou rentable par rapport aux fonctions de récompense pour se diriger vers l'objectif. En fonction des mélanges de sous skill, des comportements non voulus peuvent apparaître et il faut ainsi revoir sa hiérarchie et l'adapter et de même pour notre environnement de test.

III.3.3) Hiérarchie finale

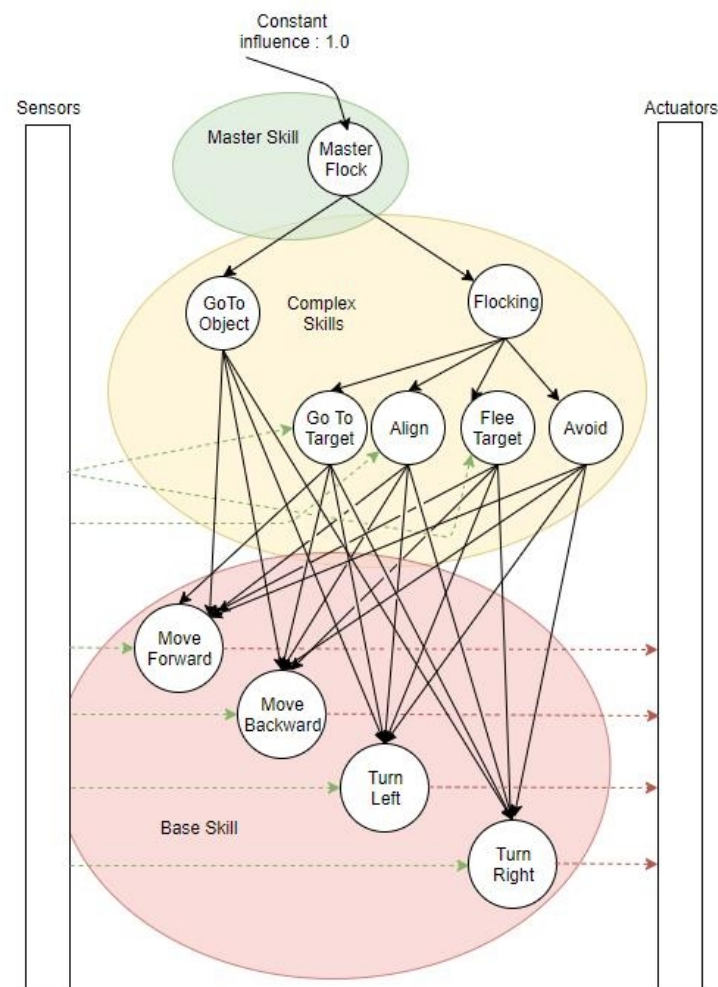
Après plusieurs exercices de conception, nous avons décidé de développer une hiérarchie du mouvement flocking. Voici ci-contre le schéma final sur lequel nous sommes basés pour le développement dans EvoAgent. Le mouvement se découpe donc en plusieurs complex skill GoToTarget, Avoid, Align et FleeTarget.

GoToTarget est un skill déjà appris de base dans l'environnement, nous avons donc tout simplement repris cet apprentissage. Ce skill permet à l'agent d'aller en direction d'une cible.

Align est un des skill que nous devons développer. Le principe de ce comportement est qu'un agent s'aligne en fonction d'un groupe d'agent perçu dans son champ de vision. Celui-ci sera développé plus en détail dans une autre partie.

FleeTarget permet à l'agent de fuir un autre agent si celui-ci est trop proche l'un de l'autre.

La combinaison de ces trois skills permettrons un mouvement de flocking avec un groupe d'agent. C'est donc avec de modèle de hiérarchie que nous nous sommes basés pour la suite de la conception.



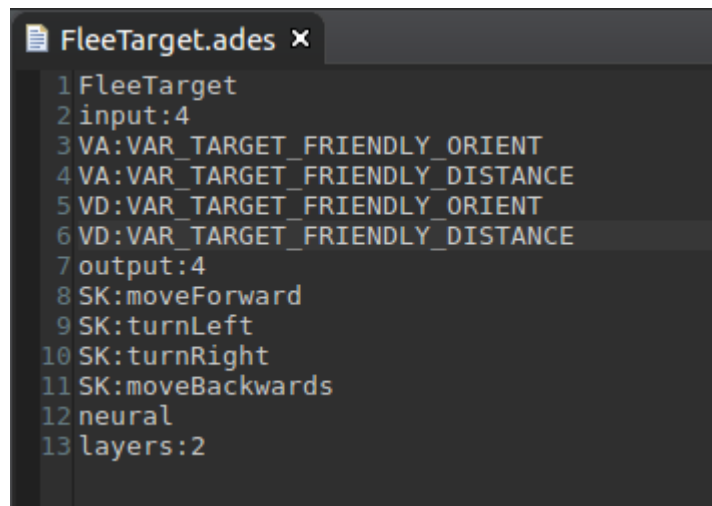
Hiérarchie finale

IV) Présentation des expériences

IV.1) Général

IV.1.1) Environnement de développement

Pour pouvoir générer nos expériences, nous utilisons comme support le programme de Mr. Suro. Pour cela, pour chaque skill que nous souhaitons apprendre ou implémenter, nous créons un dossier du nom du skill avec à l'intérieur un fichier de description, d'extension « .ades », qui va nous servir à spécifier les caractéristique du module au sein de la hiérarchie. Il est possible de déclarer deux types de skill différents, soit des hardcoded skill (compétence directement programmée en java) soit des skill appris (via réseaux de neurones). Chaque skill est défini avec un nombre X d'entrée et Y de sortie, qui peuvent être soit directement les données d'un capteur, soit une variable définie par un skill supérieur. Pour les skills hardcoded, il y a simplement une classe java à écrire, où nous définissons manuellement les sorties de notre skill à partir des différentes valeurs en entrées. Généralement, ces skills sont utilisés pour l'apprentissage d'autres compétences afin de définir les variables de notre bot, ou pour remplacer un comportement que nous n'avons pas encore appris dans la hiérarchie.

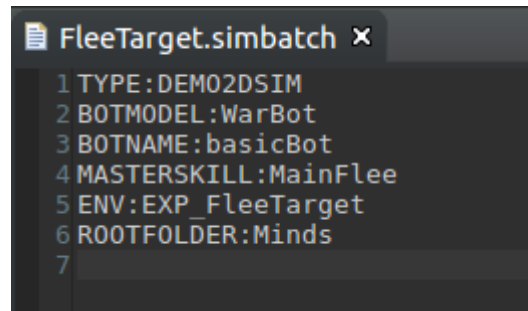


```
1 FleeTarget
2 input:4
3 VA:VAR_TARGET_FRIENDLY_ORIENT
4 VA:VAR_TARGET_FRIENDLY_DISTANCE
5 VD:VAR_TARGET_FRIENDLY_ORIENT
6 VD:VAR_TARGET_FRIENDLY_DISTANCE
7 output:4
8 SK:moveForward
9 SK:turnLeft
10 SK:turnRight
11 SK:moveBackwards
12 neural
13 layers:2
```

Exemple de fichier de description

Pour les skills appris, nous définissons également un fichier d'apprentissage, d'extension « .simbatch », dans le dossier des tâches (Tasks). Le fichier d'apprentissage se définit par le skill master et le skill qui sera appris (Learning skill). Nous pouvons ainsi avoir un skill master hardcoded qui définit des variables du Learning skill, et faire ainsi apprendre le Learning skill, tout comme nous pouvons ré-entraîner un skill dont dépend une autre compétence, en utilisant ladite compétence comme master skill. Nous définissons ensuite le nombre de

générations ainsi que la durée d'évaluation de chaque génération. Chacune d'entre elle sera également définie dans un environnement d'entraînement qui lui servira à évaluer le robot pour la tâche affiliée, et calculera son score tout au long de la simulation, jusqu'à garder le meilleur bot possible à la fin de l'apprentissage. (ref II.1.3)



```
FleeTarget.simbatch x
1 TYPE:DEMO2DSIM
2 BOTMODEL:WarBot
3 BOTNAME:basicBot
4 MASTERSKILL:MainFlee
5 ENV:EXP_FleeTarget
6 ROOTFOLDER:Minds
7
```

Exemple de fichier d'apprentissage

IV.1.2) Développement d'une expérience

La majeure partie du développement que nous avons eu à faire consiste en la conception et l'implémentation de plusieurs environnements. Ces derniers héritent de la classe `SimulationEnvironnement2D`, dérivé ensuite en `SingleBot` ou `MultiBot` selon si on veut un ou plusieurs bots dans l'environnement. Chaque environnement possède plusieurs fonctions de classe, notamment une fonction `init()` qui définit l'état initial de la simulation, et une fonction `postStepOps()` qui est exécuté après chaque tick d'apprentissage. La partie la plus importante du développement se situe dans la fonction `init`, avec la définition des fonctions de récompenses et les fonctions de contrôles. En plus de définir la taille de notre environnement, la présence ou non d'obstacles ainsi que les variables de nos bots au besoin, nous devons définir les fonctions de récompenses et de contrôles qui seront au cœur de l'évaluation. Les premières sont des fonctions qui permettent de donner un score à chaque tick de simulation en fonction des différentes actions que le robot a pu effectuer durant ce dernier. Le score final de notre bot étant la somme des scores à chaque tick, scores qui sont eux-mêmes la somme de la somme de chaque récompense (positive ou négative) de chaque fonction de récompenses du tick pour chacun des robots. Les fonctions de contrôles quant à elles, peuvent par exemple permettre de stopper la simulation et ainsi économiser du temps de calcul.

Enfin, la fonction `postStepOps` permet de modifier l'environnement pour améliorer l'émergence de nouveaux comportements. A la fin de chaque tick, une fois les actions des robots accomplies, on peut évaluer différentes valeurs de distance, d'orientation, etc... afin de modifier la simulation en cours. Il est par exemple possible de déplacer un objet que les robots cherchent à atteindre, afin de les forcer à se déplacer plus dans leur environnement.

IV.1.3) Observation des résultats de l'expérience

Une fois l'apprentissage terminé, pour analyser le comportement de notre robot, nous allons le passer en mode observation, il restituera ainsi simplement ce qu'il a appris pour que nous puissions observer son comportement. Nous pouvons ainsi repérer les défauts de notre bot et essayer de modifier l'environnement pour corriger ces défauts, en ajustant les valeurs des fonctions de récompenses ou en complexifiant l'environnement par exemple. Le réseau de neurones du robot ayant obtenu le meilleur résultat sera celui conservé à la fin d'une série d'entraînements, et celui utilisé par la démo. Nous conservons néanmoins un historique du meilleur score après chaque génération dans un fichier .out.

IV.2) Flee

IV.2.1) But du comportement

Le comportement « Flee » ou comportement de fuite est un comportement qui, comme son nom l'indique, doit faire fuir un agent le plus efficacement de sa cible (en termes de distance et de temps). Nous désirons donc que notre robot s'éloigne le plus rapidement possible d'une cible que l'on définit via une variable. Il doit s'adapter au potentiel changement de position de la cible durant le déroulement de l'expérience, en changeant de direction si besoin est.

IV.2.2) Environnement de l'expérience

L'expérience se déroule dans un environnement relativement simple lui aussi. Elle est ainsi définie par un environnement sans obstacles de taille suffisamment grande pour permettre au robot de s'éloigner suffisamment loin.

L'expérience possède quatre fonctions de récompense. Une fonction principale si le robot se rapproche de la cible, avec une récompense négative, ainsi le robot sera puni s'il se rapproche de la cible, et se voit récompensé s'il s'en éloigne. La récompense est proportionnelle à l'écart de distance entre deux ticks consécutifs. Les trois autres fonctions proposent des récompenses moins importantes.

Une fonction qui récompense la vitesse du robot, donc plus il va vite plus il sera récompensé. Une autre pour forcer le robot à avancer en avant, et enfin une dernière pour encourager le robot à avancer en ligne droite.

L'expérience possède également une fonction de contrôle. Elle a pour but de supprimer les robots qui ont des comportements peu intéressants. Nous allons ainsi arrêter l'expérience si le robot se rapproche trop de la cible. Nous pourrions ainsi évaluer directement

le bot suivant et gagner du temps, mais voir aussi plus rapidement un comportement intéressant émerger.

IV.2.3) Expériences complémentaires

L'expérience a été modifiée à plusieurs reprises, notamment afin de raffiner le comportement du robot. D'abord, nous privilégions les robots qui sont les meilleurs à s'éloigner de la cible, peu importe la vitesse. Nous avons entraîné ensuite plusieurs générations pour avoir le meilleur éloignement possible. Enfin, nous avons modifié l'expérience en augmentant les récompenses pour la vitesse afin d'avoir un robot qui va non seulement s'éloigner de la cible mais plus efficacement.

De plus, pour que le robot puisse également s'adapter si la cible se déplace, nous allons mettre dans de nouvelles conditions notre robot si celui-ci s'éloigne assez de la cible. Le robot est ainsi remis au centre de notre environnement, et la cible va elle aussi se positionner de manière aléatoire dans une position située à proximité du centre, sans jamais se superposer au robot lors de son repositionnement pour éviter de le gêner.

IV.3) KeepSameSpeedAsTarget

IV.3.1) But du comportement

Ce comportement a pour objectif de donner à notre robot la même vitesse qu'une cible et cela le plus fidèlement possible sans pour autant avoir la même direction de déplacement, ainsi si la cible est initialement orienté vers la droite avec une vitesse de déplacement de 2 et que notre robot est orienté vers la gauche il devra continuer d'avancer vers la gauche à une vitesse de 2.

IV.3.2) Environnement de l'expérience

L'environnement d'expérience de KeepSameSpeedAsTarget est un environnement vide avec notre robot localisé au centre, il récompensera le robot à chaque tick en fonction de sa différence de vitesse par rapport à la cible en s'assurant que le robot ne s'éloigne pas de sa position initiale. Ces récompenses serviront d'évaluation au robot dans les exercices suivants :

- **Exercice 1** : Une cible générée devant le robot, dans la même orientation que celui-ci, se déplace tout droit à une vitesse constante
- **Exercice 2** : Même chose que l'exercice 1 mais avec une vitesse variable
- **Exercice 3** : Même chose que l'exercice 2 mais avec une cible et généré avec une position et orientation aléatoire.

- **Exercice 4** : Même chose que l'exercice 3 mais la cible se déplace aléatoirement
- **Exercice 5** : Une cible générée aléatoirement mais restant fixe (vitesse nulle)

Les exercices sont faits de tels sorte qu'il est assez facile pour les agents de réussir les premiers selon les fonctions de récompenses de l'environnement. Cela permet de faire émerger un comportement cohérent rapidement, tandis que les derniers exercices seront plus difficiles à réussir pour nos robots mais leurs permettront ainsi de se perfectionner et de sélectionner le robot qui s'adapte le mieux aux situations rencontrées.

IV.4) KeepSameOrientationAsTarget

IV.3.1) But du comportement

Ce comportement a pour objectif de donner à notre robot la même orientation de déplacement qu'une cible et cela en restant au même endroit, ainsi si la cible se déplace vers la droite, notre robot devra s'orienter vers la droite le plus vite possible et cela sans changer de position.

IV.3.2) Environnement de l'expérience

L'environnement d'expérience de KeepSameOrientationAsTarget est un environnement vide avec notre robot localisé au centre, il récompensera le robot à chaque tick en fonction de sa différence d'orientation par rapport à la cible en s'assurant que le robot ne s'éloigne pas de sa position initiale. Ces récompenses serviront d'évaluation au robot dans les exercices suivants :

- **Exercice 1** : Génération d'une cible autour notre robot avançant tout droit dans une direction aléatoire et avec une vitesse variable.
- **Exercice 2** : Génération d'une cible qui se déplace aléatoirement dans l'environnement
- **Exercice 3** : Génération d'une cible aléatoirement dans l'environnement se déplaçant vers le robot
- **Exercice 4** : Génération d'une cible qui se déplace en orbite autour du robot
- **Exercice 5** : Génération aléatoire d'une cible fixe (vitesse nulle)

Ces exercices permettent de confronter le robot aux différents cas de variations d'orientation de la cible, quelle que soit sa distance ou ça vitesse.

IV.5) Align

IV.5.1) But du comportement

Ce comportement a pour objectif d'imiter les déplacements d'une cible le plus fidèlement possible ainsi si la cible tourne à droite à une vitesse de 2, notre robot devra tourner à droite à une vitesse de 2, si la cible accélère, notre robot accélèrera et ainsi de suite. Concrètement notre agent devra donc utiliser intelligemment les capacités KeepSameSpeedAsTarget et KeepSameOrientationAsTarget.

IV.5.2) Environnement de l'expérience

L'environnement d'expérience de Align est un environnement vide avec notre robot localisé au centre, il récompensera le robot à chaque tick en fonction de sa différence de distance initiale, d'orientation et de vitesse par rapport à la cible. Ces récompenses serviront d'évaluation au robot dans les exercices suivants :

- **Exercice 1** : Une cible générée dans une position et orientation aléatoire se déplace tout droit avec vitesse variable au cours du temps.
- **Exercice 2** : Une cible générée dans une position et orientation aléatoire se déplace aléatoirement dans l'environnement avec une vitesse variable au cours du temps.

Avec les exercices 1 et 2 les robots auront tendances à simplement aller en direction de la cible c'est pour eux le comportement le plus simple à développer qui leurs permettra de gagner des points (les robots perdent des points au début mais finissent par être aligné avec la cible s'ils la suivent longtemps et finissent par gagner des points).

- **Exercice 3** : Génération d'une cible aléatoirement dans l'environnement se déplaçant vers le robot

L'exercice 3 est conçu pour punir les robots suiveurs, la cible se dirigeant vers le robot, celui-ci ne peut plus se contenter de seulement suivre car sinon il ira dans la mauvaise direction et perdra des points.

- **Exercice 4** : Génération d'une cible qui se déplace en orbite autour du robot
- **Exercice 5** : Génération aléatoire d'une cible fixe (vitesse nulle)

Ces derniers exercices permettent de confronter le robot a des cas limites qu'il ne rencontre pas dans les autres exercices. Ils sont un bon moyen de confirmer si le robot est finalement capable de faire ce qu'on attend de lui.

IV.6.3) Expériences complémentaires

L'expérience pour l'apprentissage de la capacité Align fut initialement développée comme une capacité simple, avec les mêmes fonctions de récompenses mais sans les sub-skills KeepSameSpeedAsTarget et KeepSameOrientationAsTarget, cependant les robots avaient du mal à sortir de leurs comportements de suiveurs même en ajoutant des exercices différents et même après des milliers de générations plus tard la découverte d'un meilleur devenais de plus en plus longue. Nous avons alors décidé de tirer parti de la modularité de l'architecture MIND en développant des sub-skills utiles pour l'apprentissage de Align en espérant pouvoir améliorer la capacité d'alignement.

IV.6) Flock

IV.6.1) But du comportement

Le comportement de flocking est le comportement au sommet de la hiérarchie que nous avons choisi d'entraîner. C'est un comportement que l'on retrouve dans la nature chez les oiseaux ou les bancs de poisson par exemple, qui consiste à se déplacer en groupe d'individu vers un lieu/but commun. En informatique, ce comportement est simulé pour la première fois par Craig Reynolds en 1987 avec le programme « Boids ».



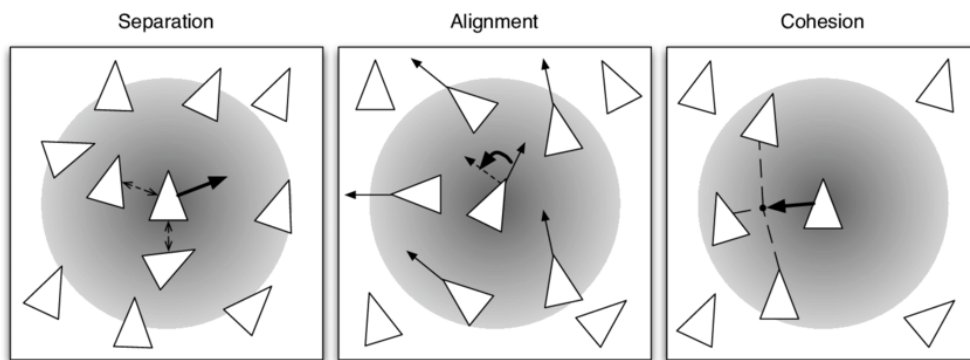
Image d'une simulation de flocking (gauche) et une nuée d'oiseaux (droite)

Il s'agit pour les individus de réussir à se mouvoir au sein d'une nuée, en ayant un champ de perception limité, en suivant quelques règles simples :

- La séparation : chaque individu doit éviter de trop se rapprocher de ses congénères et risquer de gêner leur mouvement, et s'en éloigner le cas échéant.
- La cohésion : chaque individu doit essayer de rester au sein du groupe sans trop s'en éloigner. On se réfère en informatique au centre de masse des individus perçus pour se diriger, appelé barycentre.

- L'alignement : chaque individu doit se diriger dans la même direction que ses congénères. On récupère pour cela la direction prise par les individus qui sont dans le champ de perception de l'individu, et on effectue une moyenne de ces valeurs pour l'orienter.

A cela, on rajoute dans notre cas l'évitement d'obstacle puisque nos agents vont devoir évoluer à terme dans un environnement comportant plusieurs obstacles à éviter. Ce skill, au sommet de la hiérarchie, va s'appuyer sur quatre comportements qui ont pour but de simuler les différentes fonctions du flocking présentées auparavant. Il s'agira donc pour le module Flock de la hiérarchie d'apprendre à coordonner ces fonctions pour obtenir un résultat satisfaisant.



IV.6.2) Environnement de l'expérience

Pour l'expérience qui sert à entraîner nos robots, nous avons placé quatre bots dans un environnement vaste, pour être sûr qu'ils aient la place d'expérimenter les déplacements nécessaires. Nous avons choisi de ne mettre initialement que quatre bots afin de ne pas avoir des temps d'entraînement qui soient trop long dans un premier temps, alors que les bases du comportement sont à apprendre complètement. De plus, dans la première version de l'expérience, aucun obstacle n'est placé dans l'enceinte où évoluent les robots afin qu'ils apprennent avant tout à se positionner par rapport à leur groupe sans qu'aucune perturbation extérieure ne vienne les gêner. La cible qu'ils doivent atteindre ne bouge pas non plus. Elle est située assez loin du centre de l'arène où sont positionnés les bots au départ, mais reste immobile afin que ceux-ci puissent justement apprendre un comportement lorsqu'ils doivent rester à un endroit fixe. Les robots commencent l'expérience en formation carré, afin de n'être ni trop proche d'un autre robot, ni trop éloigné du groupe dès le départ, mais également que chacun d'entre eux aient des conditions initiales similaires.

Pour ce qui est des récompenses, chaque robot est récompensé s'il reste à une distance inférieure à un certain seuil du barycentre du groupe. Il est en revanche puni s'il se rapproche trop d'un autre bot. Ces deux récompenses sont celles qui visent à récompenser l'apprentissage d'un réel comportement de flocking des robots. Les robots sont également récompensés s'ils se rapprochent de la cible qui leur est donnée. La cible et la récompense

associée vise à encourager les robots à se mettre en mouvement, et éviter qu'ils ne restent dans une position statique qui leur ferait tout de même gagner des points. De la même manière, être en mouvement, leur rapporte une quantité de points moins importante que les trois autres fonctions, mais les encourage également à rester en mouvement. L'expérience s'arrête au bout d'un nombre de ticks donné au lancement de l'expérience.

IV.6.3) Expériences complémentaires

Les expériences subsidiaires ont mis en place deux changements importants. En premier lieu, des obstacles ont été ajoutés à l'environnement afin de cette fois-ci mettre les bots dans des conditions plus proches de celles qu'ils auront finalement. Le flocking devra donc cette fois accorder une importance plus importante à l'esquive d'obstacle, puisqu'une collision avec l'un d'eux est fortement pénalisée. On cherche en effet à avoir un mouvement qui soit fluide en toute circonstance, et les collisions entravent non seulement le mouvement du robot qui la provoque, mais également du reste du groupe. Le deuxième changement est le déplacement de la cible lorsqu'un des robots l'atteint. Là encore, l'idée est de favoriser le mouvement, et de faire se déplacer le plus possible les robots dans l'environnement et de se confronter aux difficultés de s'y déplacer.

V) Réflexions

V.1) Résultat obtenus

Résultat pour le Flee :

Après divers échecs dû à une mauvaise compréhension du système des variables du projet, ce comportement marche parfaitement. En effet, même si on a défini la valeur de nos variables dans le init() de notre expérience, les variables doivent être initialisées dans un skill « Master ». Pour ce skill, par exemple, si nous voulons apprendre le skill Flee, nous devons créer un skill « MasterFlee » hardcoded qui va définir les variables et donner les variables en entrée à notre skill Flee que nous souhaitons apprendre. Une fois cette particularité comprise, l'apprentissage c'est fait très rapidement. Nous possédons donc un bot capable de fuir une cible efficacement mais aussi capable de s'adapter au mouvement de cette cible.

Résultat pour KeepSameSpeedAsTarget :

Étant une capacité simple, l'apprentissage de ce comportement n'a pas posé de problèmes particuliers, après avoir créé un environnement d'entraînement, nous avons tout d'abord fait un lancement test sur 30 générations (~1h de calcul sur un cluster avec 28 coeurs). En observant le résultat de cette entraînement test on observe que le robot avance tout droit et arrive à varier un peu sa vitesse avec la cible ce qui est un résultat assez encourageant pour lancer un entraînement de longue durée sur le cluster, 1000 générations (~2j) plus tard le comportement a beaucoup progressé et le robot arrive à faire correspondre sa vitesse avec la cible avec un léger décalage, cependant il lui arrive quelque rare fois de partir en arrière, problème que 1000 générations en plus ont fini par régler.

Résultat pour KeepSameOrientationAsTarget :

Les étapes d'apprentissage furent sensiblement les mêmes que KeepSameOrientationAsTarget : un apprentissage test sur 30 générations donnant un résultat encourageant avec un robot qui reste à sa position et qui s'oriente hasardement dans la même direction que le déplacement de la cible. 1000 générations plus tard le meilleur robot arrive à s'orienter rapidement avec le mouvement de la cible mais un peu trop à droite ce qui nous poussa à engager 1000 générations de plus qui donnèrent un résultat satisfaisant bien qu'encore imparfait.

Résultat pour Align sans sub-skills :

Après un entraînement intensif de la capacité Align sans utiliser les sub-skills KeepSameOrientationAsTarget et KeepSameSpeedAsTarget, le comportement émergent se retrouve être un robot se positionnant derrière à droite de la cible puis fixant sa vitesse puis s'orientant dans la même direction que celle-ci. Même après des milliers de générations en

plus nous ne virent pas de réelles améliorations de ce comportement ce qui nous poussa à développer des compétences spécialisées dans la vitesse et l'orientation.

Résultat pour Align avec sub-skills :

L'entraînement du Align utilisant les sub-skills KeepSameSpeedAsTarget et KeepSameOrientationAsTarget est toujours en cours, nous espérons qu'il fournira de meilleurs résultats que la capacité précédente et avons hâte de pouvoir comparer les deux résultats lors de notre soutenance.

Résultat pour le Flock :

La première expérience de flocking a vite porté ses fruits, et après une centaine de génération, on a pu observer un comportement de flocking. Cependant, impossible de réellement savoir si ce comportement était la conséquence directe du placement de départ des robots (en carré), ou bien si le comportement avait été correctement appris. Une fois autour d'une cible fixe, les robots avaient également tendance à se placer en file indienne en se déplaçant autour de l'objectif. Ce comportement pouvait également prêter à confusion puisqu'il ne permet pas d'être sûr que les robots aient bien appris à gérer la distance avec le reste des robots du groupe.

L'ajout d'obstacle a par la suite mis en évidence un problème en rapport avec le comportement de cohésion. Les bots ne pouvaient pas percevoir les robots au-delà d'une certaine distance, et finissaient par se séparer en groupe de deux robots et perdaient de vue les autres. Il a donc fallu augmenter leur rayon de perception des robots alliés afin qu'ils continuent à se mouvoir ensemble après avoir évité un obstacle. Un autre problème soulevé par ces expériences est que les robots ne modifient pas leur vitesse pour ajuster leur position par rapport aux autres robots et pour les attendre. Pour obtenir un comportement plus proche du comportement voulu, le skill Align est ré-entraîné et deux subskills sont également créés : SameOrientAsTarget et SameSpeedAsTarget. A l'issue de leur entraînement, le flocking sera à nouveau testé.

V.2) Gestion du projet

V.2.1) Organisation

Pour pouvoir faire gagner le jeu de capture de drapeau à notre équipe d'agents intelligents nous devons chacun développer un ensemble de compétences nécessaires à nos agents pour qu'ils puissent évoluer dans leur environnement puis se défendre contre leurs adversaires. Ainsi avant de pouvoir développer des compétences visant à la victoire de l'agent dans le jeu nous nous sommes d'abord concentrés sur les compétences basiques et minimales permettant à l'agent de pouvoir se mouvoir dans l'environnement du jeu. Après plusieurs réunions nous avons déterminé que la capacité qui serait la plus intéressante à développer serait la capacité de déplacement en groupe dans l'environnement (Flocking).

Nous avons alors naturellement départagé le travail en 4 avec chaque personne s'occupant de développer un environnement d'entraînement pour une des 3 capacités simple (Alignement, Répulsion, Attraction) et un dernier s'occupant de développer un environnement pour la capacité complexe (Flocking).

Chaque mardi après-midi des réunions étaient organisées, sauf exception, pour faire un point sur l'avancement du projet avec notre référent monsieur Suro mais aussi si nos schémas de hiérarchie était correct pour l'avancement du TER. De plus un salon discord (application de discussion) a été mis en place pour une meilleure communication dans le groupe.

V.2.2) Révisions de l'objectif

En développant les différentes compétences nécessaires au « flocking » de nos agents nous nous sommes vite rendu compte que la réalisation d'une hiérarchie complète de compétence pour gagner un jeu de capture de drapeau ne serait pas possible dans les temps impartis. En effet, le coût en temps de développement des environnement d'entraînement, des fonctions de récompenses pour chaque environnement, et surtout le coût en temps de calcul de sélection de meilleurs agents génération après génération s'est avéré bien supérieur à ce à quoi nous nous attendions. C'est pourquoi nous avons décidé de nous focaliser sur le développement du « flocking » et avoir quelque chose de fonctionnel plutôt que de vouloir en faire trop et ne pas avoir un rendu final correct.

VI) Conclusion

Ce projet nous a permis de découvrir le monde de la recherche dans le domaine de l'informatique tout en approfondissant nos connaissances de la programmation orientée agents et des réseaux de neurones. Ce projet nous tenait à cœur car il touchait à plusieurs domaines différents et il pouvait non seulement avoir de l'intérêt pour ceux d'entre nous qui souhaiteraient poursuivre dans la recherche mais également pour la poursuite de notre carrière, une fois en entreprises pour ceux d'entre nous qui choisirons cette voie.

Nous avons pu réaliser au cours de ce TER un skill complexe composé d'un master skill, de plusieurs sub skill et base skill.

Ainsi nous avons réalisé un skill de flocking, comportement de groupe que nous avons vu avec le cours de Mr Ferber, et que nous avons pu ici créer via des réseaux de neurones. Ce projet nous a énormément appris sur le monde de la recherche, le mode de travail en laboratoire, ou encore le système de publications. Du fait des différentes expériences ainsi que par les diverses réflexions et remarques que nous avons pu faire, nous avons également pu contribuer à la thèse et aux divers articles de Mr. Suro.

Bibliographie

A hierarchical representation of behaviour supporting open ended development and progressive learning for artificial agents

François Suro, Jacques Ferber, Tiberiu Stratulat, 2018

Evolving Neural Networks through Augmenting Topologies

Kenneth O. Stanley et Risto Miikkulainen, 2002.

<http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf>

Control Architecture for Robotic Agent Command and Sensing

Terrance Huntsberger, Hrand Aghazarian, Tara Estlin, Daniel Gaines, 2008

<https://ntrs.nasa.gov/search.jsp?R=20080048021>

A Control Architecture for an Autonomous Mobile Robot

Maria C. Neves, Eugénio Oliveira, 1997

https://www.researchgate.net/publication/37649952_A_control_architecture_for_an_autonomous_mobile_robot

Architecture Controlling Multi-Robot System using Multi-Agent based Coordination Approach

Mehdi Mouad, Lounis Adouane, Pierre Schmitt, Djamel Khadraoui, Philippe Martinet, 2011

<https://hal.archives-ouvertes.fr/hal-01714858/document>

Robot Shaping : an experiment in behavior engineering

Marco Dorigo, Marco Colombetti, MIT press, 1998