Sorbonne Université

Projet Dédale, Fosyma

Rapport de projet

Encadrant :
Cedric Herpson
Aurélie Beynier
Nicolas Maudet

Groupe 8 :
Yannis Elrharbi-Fleury
Axel Foltyn

Table des Matières

1 Introduction

Ce projet a pour but de développer une équipe d'agents coopératifs sur la plate-forme multiagent JADE.

Ceux-ci disposent d'une capacité de communication limitée. Ils sont chargés d'explorer un environnement inconnu, de détecter puis bloquer des intrus présents sur la carte, le plus rapidement possible.

2 Structure d'un agent

Dans notre programme, nous séparons les capacités d'interaction d'un agent avec son environnement (capacités dites sensorielles), et ses capacités de logique (capacités cognitives ou décisionnelles).

Nous distinguerons alors des comportements propres au corps de l'agent, et d'autres propres à son cerveau.

Le corps et le cerveau de l'agent peuvent communiquer entre eux, mais leurs comportements enfants doivent interroger ou solliciter leur parent pour récupérer ou stocker durablement des données.

Ce choix a été fait pour faciliter la compréhension du code et le travail en binôme. Lors du développement la détection des problèmes et les ajouts de fonctionnalités étaient alors plus clairs.

2.1 Capacités d'interaction

Les capacités dites sensorielles de l'agent lui permettent d'interroger et d'interagir directement avec son environnement. Elles permettent de répondre aux problématiques suivantes.

1. Qu'est-ce qu'il y a autour de moi ?
   - observation : j'observe l'état du graphe autour de moi.
   - écoute : je récupère des informations sur les autres agents (sont-ils proches ? que font-ils ?).
2. Comment interagir avec mon environnement ?
   - déplacement : je me déplace dans le graphe.
   - communication : j'envoie des informations sur mon état aux autres agents.
   - migration : je change de plate-forme.

2.2 Capacités de logique

Les capacités dites cognitives de l'agent lui permettent de stocker des connaissances, puis de prendre des décisions informées en fonction de celles-ci. Elles permettent de répondre aux problématiques suivantes.

1. Que veux-je faire ?
   - discussion : je souhaite récupérer les informations concernant quelqu'un.
   - découverte : je souhaite découvrir mon environnement (explorer, patrouiller...).
   - chasser : je souhaite chasser les intrus.
2. Que sais-je ?
   - récupération : je mets à jour mes connaissances.
   - échec : mon action précédente à échouée.
   - détection : je détecte quelque chose près de moi (un équipier, un intrus...).
3. Que dois-je faire ?
   - décision : je choisis l'action suivante en fonction de ce que je sais et de ce que je veux.
4. Comment le faire ?
   - exécution : j'essaie de mettre en action ce que j'ai décidé.

Les agents que nous avons développés ne possèdent pas de capacité de déduction. Ils ne sont pas encore capables de prédire les états futurs de l'environnement, cela les aurait rendu plus efficaces dans leurs tâches.

2.3 Connaissances des autres agents

Chaque agent sait quelque chose sur ses équipiers, il stocke ces connaissances dans un objet AgentKnowledge, lui même stocké dans son cerveau. Cet objet permet de stocker les informations suivantes.

- Ai-je déjà eu des nouvelles de cet agent ? Si oui, que faisait-il, où était-il, où allait-il et quand était-ce ?
- Que sait-il de la carte ?
- A quel point veux-je discuter avec lui ?

Cette dernière connaissance est l'attribut double meetUtility. Dans notre code, nous nous en servons seulement lors de l'exploration. Il a comme fonction d'utilité :

$$u(A) = 1000 \times (1 + \text{itWantsToMeetMe}) \times \frac{0.5 \times \text{diffEdge} + \text{diffNode}}{\text{distance}^2}$$

avec itWantsToMeetMe un booléen et diffEdge, diffNode la différence entre sa carte et la carte de l'agent.

Pour savoir quand une nouvelle à été reçue, on utilise la méthode ACLMessage.getPostTimeStamp(), dont la valeur dépend de la machine. Cet information étant utilisée pour mesurer la pertinence d'un message, on en déduit que des problèmes pourraient survenir lors de la migration de certains agents de l'équipe sur d'autres machines.

2.4 Communication

Un agent possède 3 protocoles de communication distincts.

1. Messages envoyés à chaque instant
   - PingPosition : j'envoie un objet sérialisable ayant pour attributs les nouvelles à mon sujet (ce que je fais, où je suis, où je vais).
2. Messages envoyés lorsque des agents sont détectés à portée.
   - SharePath : j'envoie à tous les agents autour de moi le chemin que je m'apprête à prendre (ils pourront alors me retrouver).
   - ShareMap : j'envoie aux agents intéressés, ou qui je pense pourraient en bénéficier, la partie de ma carte qui leur manque.

L'écoute de ces messages s'effectue grâce à ListenBehaviour qui s'exécute à chaque pas de temps. Ce comportement d'interaction ouvre les messages de la boîte aux lettres puis les traite en fonction de leur protocole.

Dans une grille de taille n, avant d'envoyer le plus gros message (ShareMap), on effectuera dans le pire des cas $n^2 + 4 \times n^2 = 5 \times n^2$ opérations. Si k agents forment l'équipe, alors on pense que la chance de croiser un autre agent est de l'ordre de $\frac{(k-1)}{n}$. Ainsi, on a une complexité de l'ordre $5 \times k \times n$.

Note : nous avons choisi de ne pas utiliser le système des pages jaunes pour communiquer. L'agent est seulement en interaction avec celles-ci lors de son initialisation pour récupérer les

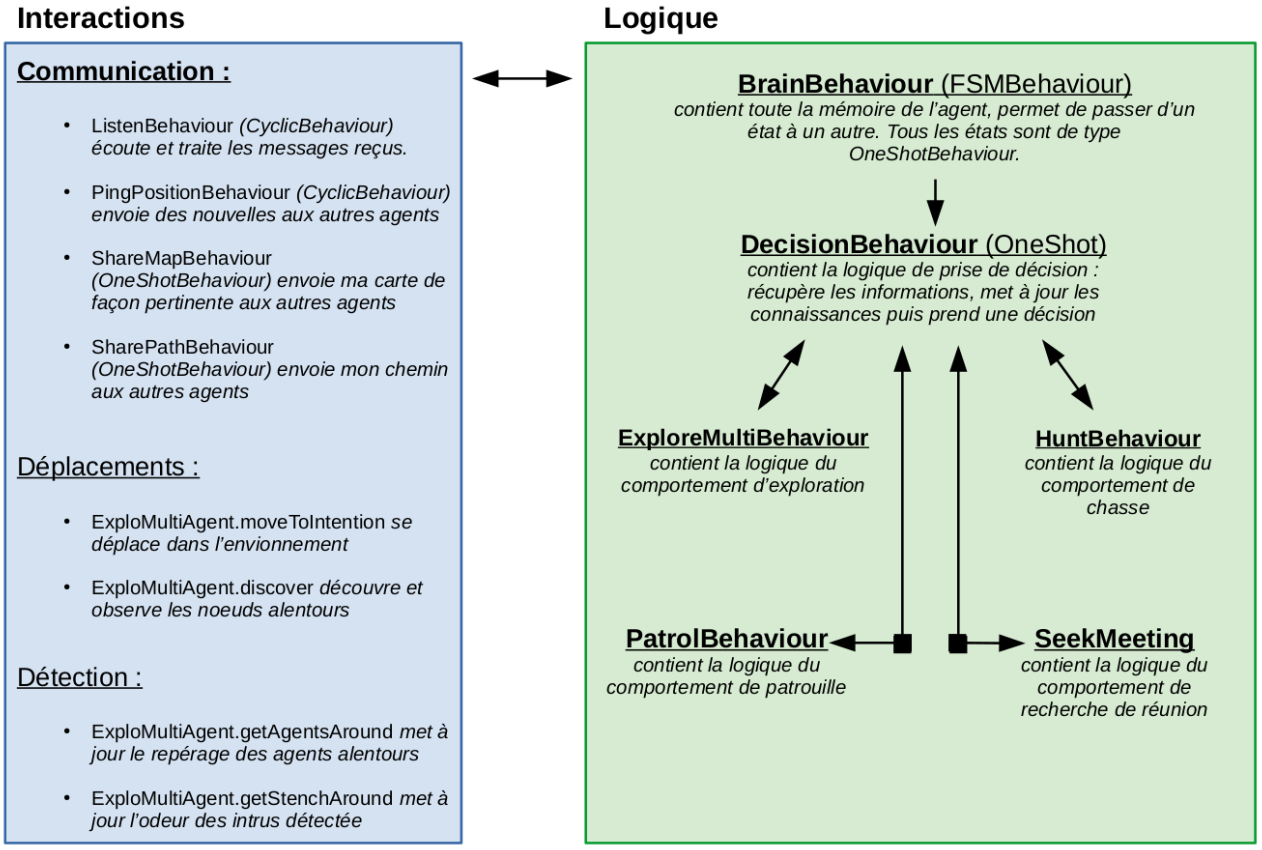
autres agents du système.

En effet, nous avons considéré que la communication parfaite avec les pages jaunes permettait de proposer des solutions allant à l'encontre du principe de capacité de communication limitée.

Ainsi, le seul moyen pour un agent d'obtenir des informations sur un autre (rôle, position, etc..) est d'utiliser un des protocoles ci-dessus.

3 Détail des comportements

Ci dessous la structure des comportements de l'agent, construite d'après ce qui précède. (Disponible en annexe)



On ne détaillera ni la détection (observation) ni le déplacement car ces derniers étaient en partie fournis avec le code du projet.

3.1 BrainBehaviour

BrainBehaviour est un comportement sous forme de machine à états finis. C'est le comportement parent de toutes les capacités de logique. Ses attributs constituent donc la mémoire de l'agent : ses connaissances des autres agents, ses états passés, etc...

Ses états prennent la forme de OneShotBehaviours. Il est constitué d'un comportement central DecisionBehaviour, et de différentes feuilles correspondants aux différentes décisions qu'il

est possible de prendre. Le comportement central est alors exécuté de manière cyclique : BrainBehaviour prend une décision puis l'exécute, avant de recommencer un autre cycle de prise de décision.

Lorsque l'agent sait que la chasse à l'intrus est terminée on considère que le jeu est fini. L'agent reste toujours capable de communiquer avec les autres au cas où ceux-ci chercheraient d'autres intrus.

3.2 DecisionBehaviour

DecisionBehaviour permet de répondre aux problématiques évoquées précédemment : que veux-je faire ? que sais-je ? que dois-je faire ?

Dans un premier temps, il récupère et met à jour toutes les informations contenues dans BrainBehaviour, puis prend une décision en conséquence.

Comme ce comportement interroge ses connaissances des autres agents, son coût algorithmique est proportionnel à $k \times n^2$ dans le cas d'une grille de taille n.

Le fait que toutes les données de ce comportement soit recalculées à chaque appel (donc de manière cyclique), pose un problème d'efficacité algorithmique de nos calculs. Nous aurions pu organiser nos données de manière plus efficace pour que la récupération d'information soit moins coûteuse, ou réduire la fréquence des prises de décision.

3.3 ExploMultiBehaviour

Après avoir observé les alentours, l'agent sait où se trouvent les noeuds ouverts. Il sait aussi si des agents (et qui) se trouve à proximité.

Si un de ces noeuds est directement accessible, alors il l'explore.

Sinon, il classe par ordre croissant des distances les noeuds ouverts dont il a connaissance. Il effectue ensuite une négociation (implicite) avec les agents alentours : le choix du noeud suivant se fait en fonction de l'ordre lexicographique de l'agent par rapport à son entourage.

Si des agents alentours ont un nom plus grand, alors ils ont la priorité pour les noeuds les plus proches.

Cette négociation permet aux agents de mieux se répartir l'exploration des zones. Sans cela, bien qu'il y ait un partage de carte, des agents côte à côte se suivraient et exploreraient tous la zone de noeuds ouverts la plus proche.

De plus, à chaque pas d'exploration il est possible que le déplacement précédent ait échoué (à cause d'un intrus bloquant le passage, ou d'un autre agent qui n'étant pas en cours d'exploration). Si c'est à cause d'un intrus, alors il vérifie qu'il ne l'ait pas accidentellement bloqué (il a alors remporté la partie), sinon, l'agent se déplace d'un pas vers un noeud aléatoire.

Cette méthode peut comporter quelques problèmes. En effet, puisque l'exploration ne se fait pas seulement selon des critères de distance aux noeuds ouverts, il est possible que deux agents se bloquent mutuellement en fonction de la topologie (dans un chemin de largeur 1 par exemple).

Dans certains cas, comme le comportement s'effectue sur un seul pas de temps, il est possible que l'agent hésite entre deux noeuds équidistants. Il alternera indéfiniment entre deux noeuds jusqu'à ce qu'un autre agent viennent lui partager sa carte.

Pour remédier à ces problèmes, nous avons décidé qu'un déplacement aléatoire avait dans tous les cas 10% de chances de se produire. L'agent n'alternera alors plus entre deux noeuds indéfiniment, et en cas de blocage dans un chemin étroit, il en sortira au bout d'un certain temps (de l'ordre de $(2 \times 0.1)^l$ avec l la longueur du chemin).

3.4 SeekMeetingBehaviour

SeekMeeting est décidé lorsqu'une des connaissances de l'agent possède une utilité supérieure à un certain seuil (voir formule $u(A)$). Il consiste à s'approcher d'un autre agent.

Les messages de partage étant envoyés lorsqu'un agent est dans les alentours d'un autre, il n'y a pas besoin de prévoir un système s'assurant de la bonne réception des messages.

Si un agent souhaite recevoir les données d'un autre, il suffit qu'il le suive. Une fois les données reçues, il s'éloignera et l'envoie s'interrompra.

Si on ne sait pas où est situé l'autre, alors la distance vers lui est infinie ($u(A)$ tend vers 0) donc SeekMeeting ne sera pas décidé pour cet agent.

Ainsi, soit on connaît sa dernière position, donc on se rapproche de celle-ci ; soit on connaît le dernier chemin qu'il a emprunté, donc on essaie de l'intercepter en se rapprochant du noeud final du dernier chemin connu.

3.5 PatrolBehaviour

Une fois la carte complètement explorée, PatrolBehaviour peut être exécuté.

Si l'agent sait qu'un autre est toujours en cours d'exploration, il part à sa recherche pour lui partager sa carte. Sinon, il parcourt la carte vers des zones où l'intrus avait été repéré pendant l'exploration, ou de façon aléatoire dans le but de détecter une éventuelle odeur.

S'il reçoit l'information d'un autre agent disant qu'un intrus été repéré, alors il se dirige vers celui-ci.

Le problème de ce mode de patrouille aléatoire est qu'il ne permet pas de préparer d'embuscades aux intrus. Il a aussi tendance à éloigner les agents les uns des autres, rendant la chasse plus difficile dans certains cas.

3.6 HuntBehaviour

HuntBehaviour consiste à chasser les intrus sur la carte. Il est déclenché lorsque la patrouille permet à l'agent de détecter une odeur. A chacune de ses exécutions, l'agent conserve dans BrainBehaviour un historique du chemin parcouru.

L'idée est que l'agent ne puisse pas revenir sur ses pas, sauf si cela l'éloignerait du nuage d'odeur. Dès lors qu'il pénètre le nuage d'un intrus, il va y rester, gênant les déplacements de l'intrus.

Cette méthode ne permet pas une coordination entre agents, le blocage de l'intrus est lent et ne s'effectue pas de manière certaine. De plus, les agents peuvent aussi se gêner entre chasseurs.

Pour remédier aux inter-blocages nous avons essayé d'introduire une négociation semblable

à celle de l'exploration en donnant la priorité à ceux ayant le moins de possibilité pour rester dans le nuage. en cas d'égalité nous passons au choix lexicographique avec les agents alentours.

Cette méthode fonctionne correctement pour des odeurs de taille 1, cependant comme elle ne permet pas aux agents de localiser l'intrus, bien que le blocage soit possible pour une taille de nuage supérieure à 2, les agents sont incapables de conclure quant au succès de la chasse. Elle n'aboutira donc pas.

Une solution envisagée consiste à introduire un système d'élection sur la position de l'intrus entre les agents.

4 Conclusion

Notre système fonctionne correctement pour une faible densité d'agents. Cependant, du fait de la quantité élevée d'informations échangée lorsque des agents sont à proximité, surcharger la carte rend les calculs trop longs.

Pour remédier à cela, il faudrait revoir la structure des données stockées dans le cerveau d'un agent. Cela permettrait de rendre les fonctions qui s'en servent moins complexes. Il faudrait aussi ajouter un accusé de réception dans les protocoles de communication pour éviter tout un nombre de calculs inutiles (que nous avons essayé de réduire via le comportement SeekMeeting).

De plus, il n'y a pas de coordination directe entre eux (pas de prise de décision centralisée). Bien que cela rende le programme plus simple à coder, d'autres problèmes apparaissent en aval : difficulté pour localiser les intrus, répartition des zones d'exploration peu efficace, etc...

Il est dommage que nous n'ayons pas su assez exploiter la fonctionnalité de messagerie.

Nous pensons que ces problèmes sont en partie dûs au fait que nous n'ayons pris le temps de bien réfléchir aux structures que nous souhaitions mettre en place que tardivement dans l'avancée du projet.

Avec le recul, nous nous sommes aperçus que nous répondions aux problèmes des différents TP de manière séquentielle. Telle Pénélope attendant Ulysse, à chaque séance nous perdions du temps à défaire ce qui avait été construit.