Sorbonne Université Sciences

FOSYMA

Projet Dédale

Antonin ARBERET
Jonathan MORENO

Mai 2019

Intro

Dans le cadre de l'UE Fondement des Systèmes Multi-Agents nous avons réalisé un projet de Wumpus multi-agent. L'objectif du projet est de programmer une flotte d'agents qui sera déployée dans un labyrinthe (représenté par un graphe) et devra y collecter un maximum de trésors en un temps imparti. La flotte comporte des explorateurs qui peuvent notamment ouvrir les coffres des trésors, des collecteurs qui peuvent transporter les trésors et un silo dans lequel les trésors doivent être déposés. Ce rapport détaille et justifie les choix d'implémentations que nous avons fait. Le code du projet est disponible sur ce dépôt : https ://github.com/AntoninARBERET/dedale_projet .

Idée générale

Nous avons choisi de séparer le processus en deux grandes parties distinctes : l'exploration et l'exploitation. Nos agents ne peuvent commencer à agir sur les trésors qu'une fois le labyrinthe complètement exploré. C'est principalement des raisons de complexité qui motive ce choix : la gestion d'un graphe dynamique lors de l'exploitation semblait trop complexe à mettre en oeuvre dans le délai imparti.
Nous déployons trois classes agents différentes, chacune ayant un ensemble de behaviours qui lui sont propres. On ajoutent les behaviours communs aux trois classes.

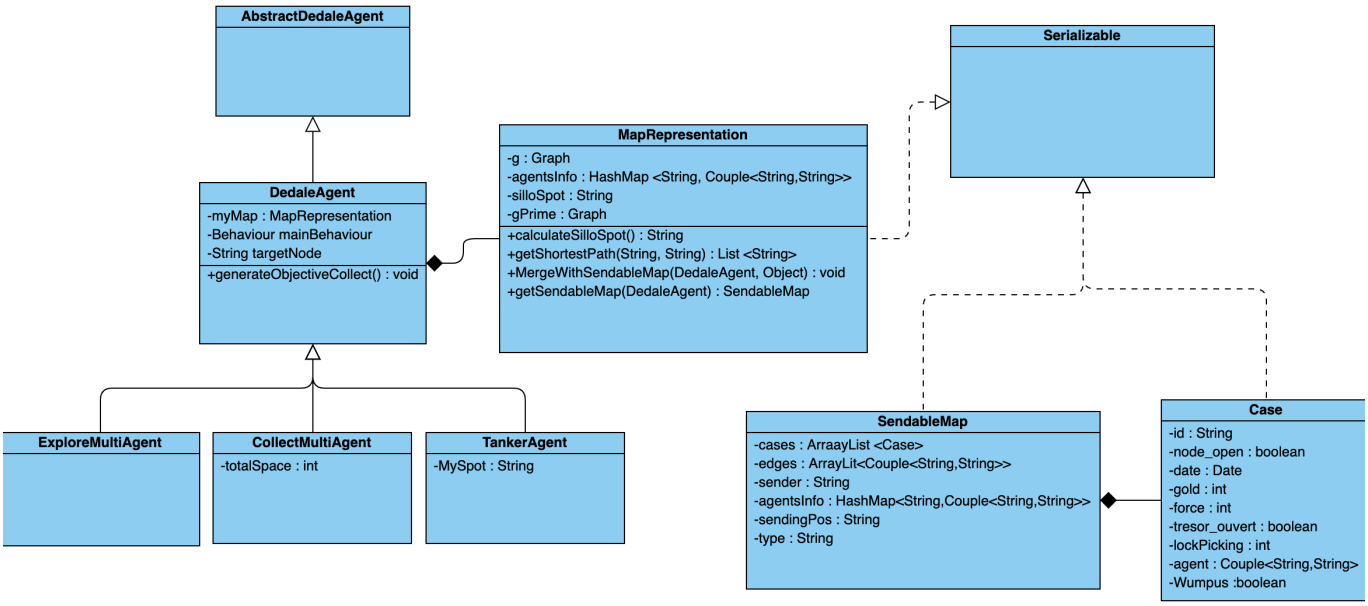

FIGURE 1 – **Caption - Source**

Les behaviours qui gèrent l'exploration, la collecte, l'ouverture ou la position du tanker ont un statut particulier puisque c'est toujours l'un d'entre eux qui est au centre du comportement de l'agent. Lorsqu'un d'eux est en cours, l'agent le garde dans une variable pour pouvoir y avoir accès. On s'en sert notamment pour les suspendre et éviter les conflits avec le behaviour de gestion des interblocages. L'autres particularité de ses behaviours sont qu'ils gèrent la majeur partie des déplacements de l'agent. Ils comportent donc tous un noeud cible représentant l'objectif actuel de l'agent : le prochain noeud ouvert, le prochain coffre à atteindre ou l'emplacement consacré au tanker. Cette variable est conservée dans l'agent dans targetNode, et mis à jour dans les behaviours directement pour pouvoir communiquer son objectif aux autres agents en cas de blocage.
Les autres behaviours sont principalement des dédiés à l'envoi et à la réception d'informations.

Exploration et partages des connaissances topologiques

Exploration

Lors de la phase d'exploration, tous les agents explorent la carte à l'exception du silo. Les agents maintiennent une liste de noeuds connus non explorés dits ouverts. Chaque agent se dirige vers le noeud ouvert le plus proche. Quant au silo, il reste sur son noeud de départ lors de l'exploration et se contente de gérer ses interblocages.

Représentation de la carte en mémoire

Chaque agent maintient une représentation de la carte dans une instance de la classe MapRepresentation. Elle contient trois éléments essentiels : un graphe, dont les noeuds portent des attributs dans auxquels on stocke les informations relatives à chaque noeud, une liste d'arêtes ainsi qu'une HashMap dans laquelle sont stocké les agents avec leur rôles et le dernier noeud ou ils ont été aperçut.

Les attributs de chaque noeud sont :

- String id : identifiant du noeud
- Boolean node_open : vrai si le noeud est ouvert
- Integer gold : contient la quantité d'or sur le noeud
- Boolean tresor_open : vrai si il y a un trésor ouvert
- Integer lockPicking : crochetage nécessaire
- Integer force : force nécessaire
- Date date : date à laquelle le noeud a été visité pour la dernière fois
- Boolean wumpus : vrai sur le dernier noeud ou le wumpus a été vu (pas utilisé finalement)
- Couple<String,String> agent : Couple agent, type la ou chaque agent a été vu en dernier (obsolète, le HashMap était plus simple à utiliser)

Ping, envoie de carte, fusion

L'exploration collective nécessite une méthode de partage de l'information. Les agent ne pouvant communiquer que par messages ACL, il était nécessaire de transmettre les informations topologiques dans une structure de données serializable.
Nous utilisons donc la classe SendableMap, qui contient une liste d'instances de la classe case représentant chacune un noeud du graphe, ainsi que la liste d'arêtes et la HashMap d'agents. La méthode getSendableMap de MapRepresentation génère la SendableMap correspondante pour pouvoir l'envoyer.

Pour faciliter le partage de connaissances, les agents s'invitent mutuellement à partager leurs cartes. Pour cela les agent ont un comportement permanent consistant à envoyer un ping à tout agent a porté, encapsulé dans la classe PingBehaviour. Chaque agent maintient à jour la date du dernier ping répondu pour chaque autre agent. A la réception d'un ping, l'agent envoie sa carte si la date de la liste est suffisamment ancienne, puis la met à jour pour éviter de rester sur place a envoyer des cartes.

Lorsqu'un agent reçoit une carte, il la fusionne avec la sienne. Il ajoute toutes les nouvelles arêtes et les nouveaux noeuds, avec les informations associées. Si un noeud reçu est déjà présent dans la carte de l'agent, une comparaison sur les dates est faite et le noeud le plus récent est conservé. Ce fonctionnement de ping et fusion de carte se poursuit lors de la phase d'exploitation, ce qui permet de propager au plus le informations récentes sur le contenus des coffres et de découvrir les emplacements ou le wumpus a déposé des ressources.

Lorsque les agents ont terminé leurs explorations, une phase de transition vers l'exploitation a lieu.

Placement et déplacement

Calcul de la position du silo

A la fin de l'exploration, chaque agent va se rendre à côté du silo et y envoyer sa carte. On s'assure que le tanker a donc un carte complète. Le silo quitte alors son noeud de départ pour rejoindre un noeud plus intéressant. Parmis les 10% de noeuds dont la distance moyenne vers tous les autres est la plus faible, il va choisir celui dont le coefficient de clustering est maximal. Sa position est donc relativement centrale dans le graphe, et le noeud choisi à un grand nombre d'arêtes entre ses voisins. Cette position est calculé par la fonction calculateSilloSpot de MapRepresentation. L'agent qui check le silo calcul lui aussi cette position. Tout agent entrant dans la phase d'exploitation sait donc où se trouve le silo.

Calcul de chemin en fonction du silo

Avant le placement du silo, les agents se déplacent vers leur noeud cible en calculant le plus court chemin par la méthode Dijkstra fournie avec le graphe. Si un agent (en particulier le silo) est sur ce chemin, un interblocage est géré.

Une fois qu'un agent a calculé la position du silo, il crée une copie du graphe auquel il retire le noeud du silo. Les agents vont tenter d'effectuer les prochaines recherches de chemin sur ce nouveau graphe. Si il en trouve un, il peut l'emprunter sans passer par le noeud du silo. Sinon il calcule le chemin sur le graphe initial et gère un blocage avec le silo. Cela permet d'éviter les mouvements du silo et de faciliter les dépôts de ressources.

Organisation de l'ouverture et de la collecte

Lors de la phase d'exploitation, les explorateurs ouvrent les coffres et les gardent, les ramasseurs ne font que ramasser.

Génération d'objectifs

Au début de la phase d'exploitation, les explorateurs et les ramasseurs génèrent une liste de noeuds à visiter dans l'ordre, et potentiellement en boucle. La liste est définie en fonction des ressources des chaque noeud mais aussi des capacités des autres agents récupérées via les DF. Les explorateurs classent les noeuds contenant un trésor par ordre croissant de nombre d'agents nécessaire à l'ouverture (en se comptant parmi eux), puis par ordre décroissant de quantité d'or disponible.
Les ramasseur font le même calcul sans s'inclure dans les ouvreur. De cette manière les explorateurs sont supposés se retrouver sur les noeuds sur lesquels la coopération est nécessaire, et les ramasseurs explore les noeuds dans leur ordre d'ouverture.

Séquence d'ouverture et garde

Lors de l'ouverture d'un trésor, seul l'explorateur qui a lancé la méthode openLock est considéré comme l'ouvreur. Son rôle suivant est de garder le noeud pour éviter que le wumpus ne déplace l'or. Si le noeud n'a qu'un voisin et donc est au fond d'un couloir, le garde se place à l'entrée de ce couloir. Sinon il reste sur le noeud. Lors de la gestion des interblocages, il laisse passer les autres agents s'ils en ont besoin. Lorsque le coffre est vide, le garde est mis au courant soit car il est sur le noeud, soit par l'agent sortant du couloir. Il se rend alors à l'objectif suivant de sa liste.

Protocole d'entraide

Si un agent arrive sur un noeud qu'il ne peut pas ouvrir, alors il n'y a plus de coffre ouvrable seul. Il attend et envoie des messages demandant de l'aide aux autres. Les autres agents, à la réception du message se placent sur un voisin de l'agent jusqu'à ce qu'il puisse ouvrir le coffre

Cycle collecte et depot

Les ramasseurs explorent la liste de coffres qu'ils ont générée. Si un de ces coffres est ouvert et contient de l'or, ils en ramasse le maximum. Si leur sac est plein, il se rendent sur le noeud voisin du silo le plus proche et vident leur sac. Si un coffre est fermé, alors l'agent se vers le noeud suivant dans la liste.
Lorsque l'agent trouve un coffre vide (ou le vide) il retire le noeud de la liste.
Arrivé à la fin de la liste, si elle n'est pas vide il retourne au début.

Gestion des interblocages

Trois niveaux d'action

Si un agent ne change pas de noeud bien qu'il ai appelé la méthode moveTo, il se considère comme bloqué. Il incrémente alors un compteur à chaque blocage successif, qu'il réinitialise quand il arrive à se déplacer normalement.
La raison de ce blocage peut être le présence d'un agent allié (on parle alors d'interblocage), ou du wumpus, nous avons donc tenté de mettre au point une réponse cohérente à cette situation basé sur trois cas :

- La méthode douce : l'envoie de carte. Cette méthode n'est utilisée qu'au premier blocage. Beaucoup d'interblocage peuvent être réglés par un simple échange de carte, en particulier lors de l'exploration. En effet, à ce moment du processus, deux agents sur des noeuds voisins n'ont plus de raison de se croiser s'ils ont les même noeuds ouverts.
- La méthode dure : protocole block. Un agent toujours bloqué après un envoie de carte envoie un objet Block à destination du noeud auquel il n'arrive pas à accéder. Si on est bien en situation d'interblocage, l'agent va en recevoir un à son tour. Les deux agents traitent cette réception comme détaillé dans la partie suivante.
- La méthode chaotique : mouvements aléatoires. Si le compteur de blocage dépasse une valeur donnée, l'agent commence à effectuer des mouvements aléatoires. L'idée étant que d'ajouter une forme d'entropie dans le blocage, permettant d'éclater un interblocage généralisé trop complexe pour notre protocole Block, ou de se déplacer pour échapper au wumpus.

Protocole block et priorité

Chaque objet Block contient la position a laquelle l'agent veut accéder, son objectif, et sa priorité et sa position actuelle. La priorité est un entier attribué à l'agent en fonction de son behaviour principal actuel. Les niveaux de priorité sont détaillés dans le tableau ci-dessous. Une priorité forte est caractérisée par un nombre élevé.
L'agent qui reçoit un Block vérifie d'abord sa priorité, si la priorité reçue est supérieure à la sienne, il va céder sa place. Sinon il ignore le blocage. En cas de priorité égale, on tranche par ordre alphanumérique sur l'identifiant des agents.
L'agent qui cède sa place va vérifier la présence d'un noeud accessible qui ne soit pas sur le chemin de l'autre agent. Pour cela il calcule le chemin de l'autre agent et cherche un noeud adjacent à ce chemin qui devient temporairement le noeud cible de l'agent. Cette action est géré dans le behaviour BlockHandlingBehaviour, qui suspend le behaviour principal jusqu'à régler l'interblocage.

Dans le cas d'un couloir avec l'objectif au bout, un tel noeud n'existe pas. L'agent envoie alors un Block à son tour, avec une priorité arbitrairement supérieure à celle reçu précédemment.

Behaviours principaux	niveau de priorité associé
BlockHandlingBehaviour	60
OpenBehaviour (phase ouverture)	50
CollectBehaviour	40
ExploExplorerBehaviour et ExploCollectorBehaviour	30
OpenBehaviour (phase de garde)	20
TankerBehaviour	10

Enfin, pour s'assurer de ne pas entrer en conflit sur l'objectif et les déplacements, le behaviour principal est suspendu tant que le BlockHandlingBehaviour n'est pas terminé.