Faculté des Sciences et Ingénierie - Sorbonne Université
Master Informatique parcours ANDROIDE

FOSYMA

Fondements des Systèmes Multi-Agents

Rapport - Dedale project

Réalisé par :

PINHO FERNANDES Enzo

BEN SALAH Adel

Année 2024-25 | Groupe 02

Table des matières

Introduction

Nous avons pu voir depuis toujours, dans la nature, de nombreux systèmes décentralisés qui s'organisent sans chef unique. Par exemple, les ruches d'abeilles qui répartissent la récolte de nectar, les bancs de poissons qui adaptent collectivement leur forme face à un prédateur, ou même les fourmis qui coopèrent pour localiser et transporter des ressources. Nous avons ici des individus avec des capacités et des perceptions limitées, qui agissent localement, et c'est l'interaction de ces actions qui génère des comportements globaux cohérents.

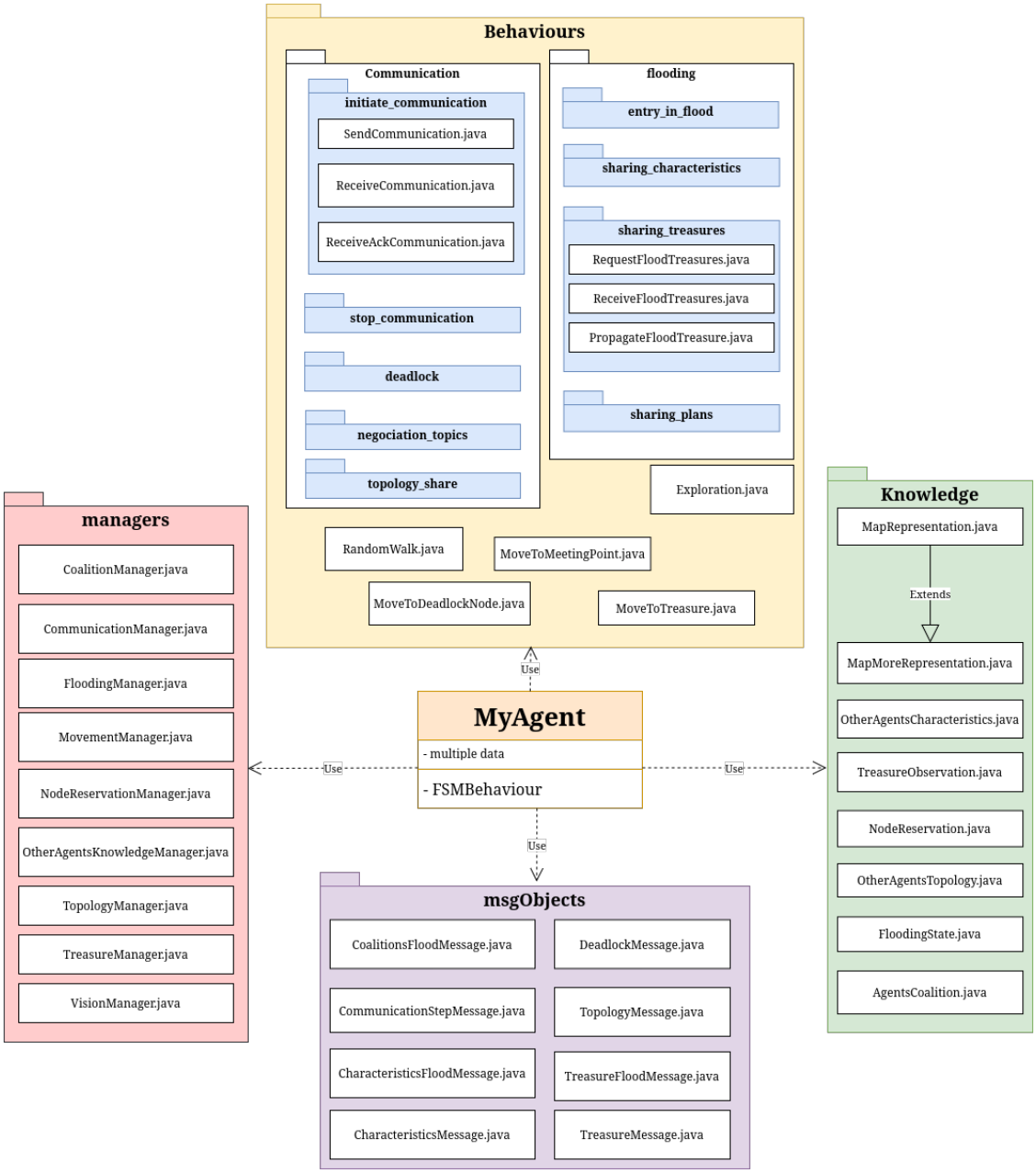
Dans un système multi-agent, nous cherchons à reproduire ce phénomène. On conçoit donc des entités autonomes, dotées de règles plus ou moins simples et capables de communiquer, afin qu'elles puissent coordonner leurs efforts vers un objectif commun.

Notre projet est une variante multi-agents du jeu "Hunt the Wumpus", un des premiers jeux informatiques (Gregory Yobe, 1972). Notre objectif a été de créer un réseau d'agents explorateurs évoluant dans un environnement inconnu, avec peu de connaissances et de capacités, et ayant pour objectif de ramasser la plus grande quantité de trésors possibles. Mais il y a tout de même des Golems (Wumpus) présents dans l'environnement, qui se feront une joie de déplacer et de fermer nos coffres.

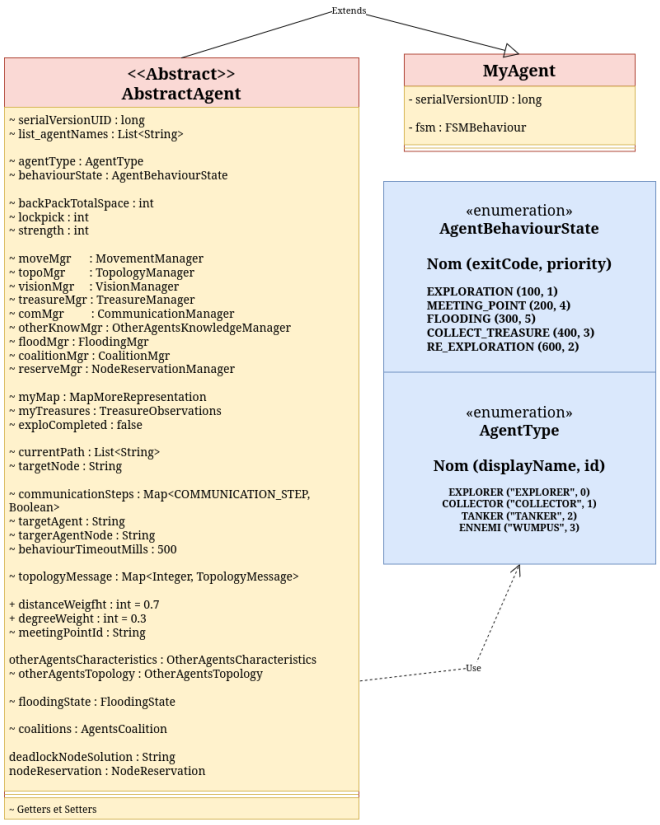
Dans la suite du rapport, nous allons vous présenter l'architecture de notre code, et les choix associés à l'exploration, la communication, la coordination et la collecte. Nous avons eu plusieurs idées intéressantes mais qui n'ont malheureusement pas pu être implémentées à temps.

Architecture générale du code

Diagramme UML simplifié



L'agent



Comme vous l'avez remarqué sur le diagramme UML simplifié plus haut, notre code se découpe en cinq grandes parties. L'agent, les comportements, les connaissances, les objets messages, et les gestionnaires.

Pour l'agent, nous avons gardé une idée simple. Tout agent aurait besoin des variables présentes dans 'AbstractAgent', une classe abstraite pour implémenter d'autres agents si besoin. La classe 'MyAgent', quant à elle, sert principalement à initialiser son 'FSMBehaviour' et à le construire.Nous avions opté pour ce moyen de faire parce que nous pensions, au début, faire des comportements relativement différents entre chaque type d'agent (Silo, Collecteur, Explorateur). Mais nous avons fini par avoir quelque chose d'homogène, alors nous avons opté pour une utilisation d'Énumération simpliste pour représenter leur différence. Ici 'AgentType'.

Notre classe 'AbstractAgent' étend 'AbstractDedaleAgent' afin de pouvoir utiliser les commandes liées à Jade et principalement au projet Dedale. On y stockera toutes leurs connaissances et gestionnaires en attributs, pour un accès simple.

Il n'y a pas beaucoup de choses à dire sur cette manière de faire. Il était juste plus simple et modulable d'utiliser les énumérations que de refaire et d'entretenir différentes classes pour les différents types d'agents dans ce scope. Sur une plus grande échelle, il serait tout de même préférable d'utiliser d'autres classes et interfaces.

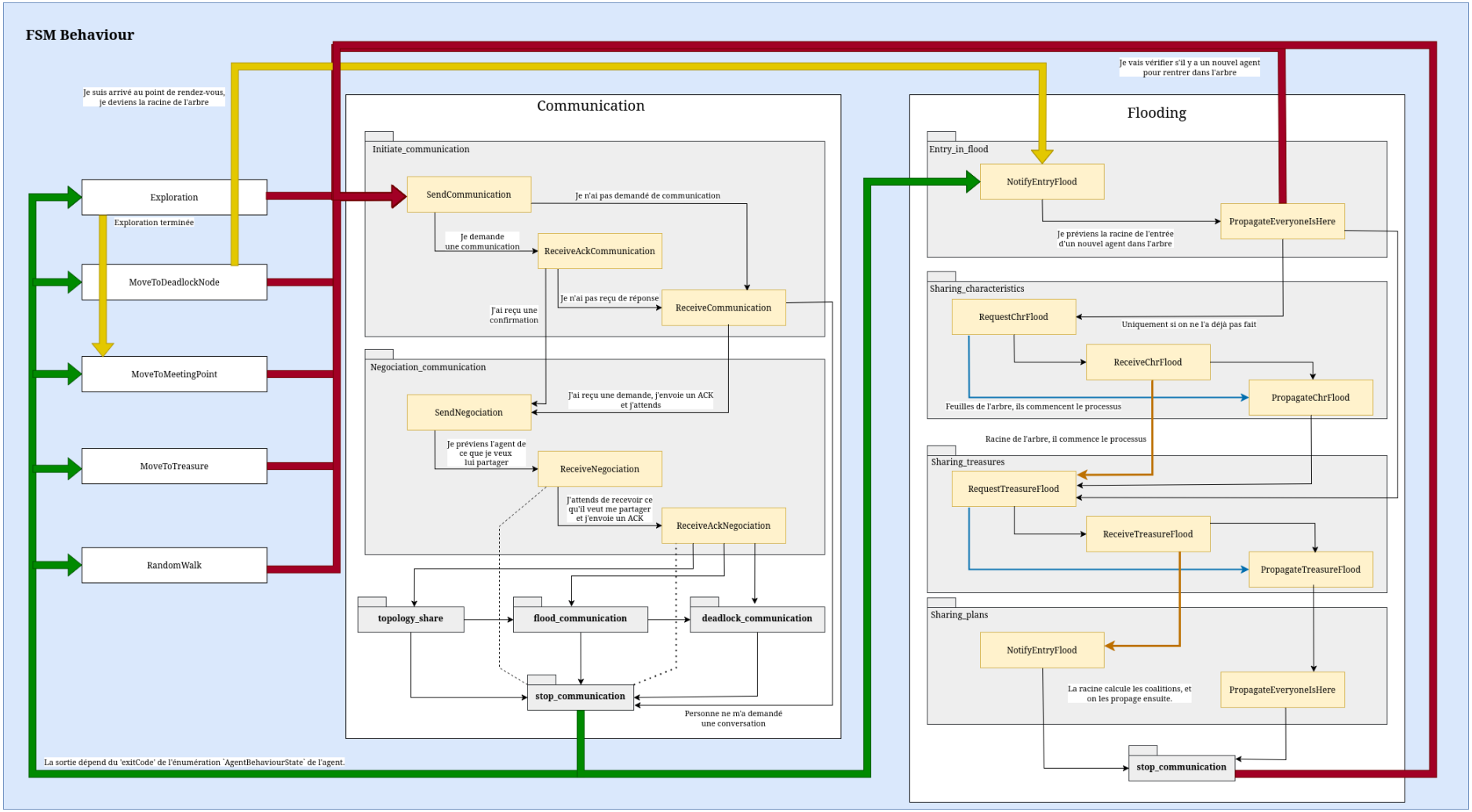


FIGURE 1 – Notre FSM global.

Les comportements

Sur la page précédente, vous pouvez voir l'architecture de notre comportement à automate fini (FSMBehaviour), ainsi que toutes les transitions possibles, ou presque. J'ai omis les transitions de timeout dans les comportements d'attente de réponse de la part d'un autre agent. De plus, 'topology_share', 'flood_communication' et 'deadlock_communication' suivent la même logique que leurs deux comportements précédents.

Nous avons décidé d'utiliser un FSMBehaviour pour sa capacité à mieux "prévoir" ce qu'un agent doit faire, dans quel ordre. En effet, avec les autres types de Behaviour (SimpleBehaviour, TickBehaviour...) ne structurent pas explicitement des transitions conditionnelles entre comportements, pouvant rendre la charge de travail complexe.

Mais nous nous sommes heurtés à plusieurs difficultés avec ce choix. En effet, il devient très difficile de changer dynamiquement de comportements selon des résultats différents, on devait donc toujours y réfléchir à deux fois avant de se lancer dans une nouvelle idée.

Chaque "état" est un sous-comportement de notre FSM. Nous avons principalement utilisé des 'OneShotBehaviour', représentant des comportements atomiques. Elles ne s'exécutent qu'une fois. Mais il nous est arrivé d'utiliser également des 'SimpleBehaviour', afin de pouvoir nous créer facilement un timer, notamment pour attendre l'une des réponses des agents.

Présentation de nos choix d'implémentation

L'exploration

Le premier défi que rencontrent nos agents est l'exploration d'un environnement totalement inconnu pour eux. Ils doivent perdre le moins de temps possible, afin de ne pas laisser le golem déplacer trop de fois des trésors, et donc de faire perdre la valeur globale présente sur la carte. Ils commenceront tous avec une carte vierge, qu'ils rempliront petit à petit.

Pour cela, nous avons adopté une stratégie assez simple. Chaque agent visera en priorité les nœuds ouverts de la carte, de préférence les plus proches de ce dernier. S'il a plusieurs choix devant lui, alors il en choisira un au hasard. Mais de plus, nous avons choisi d'utiliser un partage de topographie partiel entre agents. Il y a deux particularités :

— Le partage de la carte est partiel, on ne donne uniquement ce dont on est pas certain de ce que l'autre agent possède. Cela suppose de devoir garder en mémoire ce qu'il reçoit, et ce que l'on lui envoie. Cela explique en parti pourquoi, pour les communications, nous avions commencé avec un système de ACK.

— De plus, les agents devront découvrir X nouveaux noeuds avant de pouvoir repartager leur carte avec quelqu'un, afin de limiter les mises à jour inutiles.

Bien sûr, cette méthode est facilement critiquable, notamment pour sa complexité mémoire et temps. Devoir enregistrer les noeuds et arêtes d'agents peut vite devenir pesant suivant la topologie en elle-même et le nombre d'agents présents. Pour X agents, N noeuds et M arêtes, nous aurions donc une complexité de O(X*N*M) minimum ! Au niveau du temps, plus la complexité est haute, plus il sera lent de choisir minutieusement quels noeuds nous devrions envoyer ou non à l'agent. Mais d'un autre côté, cela veut également dire avoir moins de temps à perdre à fusionner les deux cartes. L'idée nous est venue bien après, mais nous aurions pu plutôt attribuer à chaque noeud et arête un Set de noms d'agents (des pointeurs de préférences) afin de ne pas stocker indéfiniment des Strings.

Mais ce n'est pas le seul défi de l'exploration. Durant l'exploration, nous enregistrons également nos observations, plus particulièrement ceux des trésors croisés en cours de route. Etant donné que les Golems peuvent les déplacer et les refermer, chaque agent les enregistreront, leur position, leurs attributs mais également à quel moment il a été visité la dernière fois ! Pour cela, nous nous contentant d'un 'System.currentTimeMillis' étant donné qu'on est sur la même machine. Bien sûr, sur d'autres, il faudrait penser à une autre stratégie. Ils peuvent même tenter d'ouvrir et de récolter des coffres à la volée, sans y prêter plus attention.

La communication

Comme présenté plus haut, nous avons opté pour une communication type "Demande, Réponse, Ack". Au début, nous avions des problèmes de coordination entre les différents agents, notamment par l'utilisation des TickerBehavior. Nous avons donc introduit un système de discussion à deux personnes.

Pour commencer, après chaque "état" du FSM, ils finiront par retourner dans le comportement 'SendCommunication', où ils vérifieront qui est présent autour d'eux, et s'ils ont quelque chose à leur dire ou demander. Afin de vérifier ça, ils possèdent plusieurs facteurs à prendre en compte :

— Leur état actuel. S'ils sont dans l'état 'AgentBehaviourState.EXPLORATION', ils pourront communiquer avec quelqu'un uniquement pour lui partager une carte si possible, ou pour régler un problème d'interblocage. De même dans l'état 'RE_EXPLORATION'. Pour l'état 'FLOODING', ils doivent juste regarder s'ils ont déjà contacté quelqu'un dans cet état. Et enfin, dans 'COLLECT_TREASURE', ils doivent gérer toujours les interblocages MAIS ne doivent pas essayer de se décaler pour passer quand ils sont en coalition pour l'ouverture d'un coffre par exemple.
— Mais également si eux ont reçu un message ! Par défaut, ils accepteront toujours une communication. On part du principe qu'un agent ne peut pas savoir en avance si son confrère a fait une révélation de dernière minute, et ça permet également d'éviter des problèmes au niveau du flooding.

Malheureusement, le fait d'utiliser ce type de système accroît grandement le nombre de message, pour pas grand chose. En théorie, une fois qu'ils passent en mode communication, ils ne bougent plus, nous aurions pu se passer de ce système, il y aurait eu que des faibles pertes. Nous envoyons certes beaucoup de message, mais il ne faut pas oublier que ces messages sont souvent vides de contenus, avec seulement une performative, un nom, et un envoi à un seul agent.

Le problème se creuse au niveau des interblocages où nous avons eu BEAUCOUP de mal à gérer, au point qu'ils bloquent 2-3 fois sur 4 la collecte, et casse notre système de flooding. Niveau communication, il arrive aux agents de rester bloquer entre eux, et de constamment s'envoyer des messages pour se débloquer, mais ils remplissent donc continuellement la boîte aux lettres de leur congénère, empêchant donc leur accès.

Nous prenons toujours le premier message qui vient dans ce projet, ce qui nous limite grandement, nous aurions pu attendre, en récolter plusieurs, et ainsi choisir la meilleure course d'action par rapport à ça. De plus, se limiter à une seule personne durant une conversation est vraiment limitant, alors que ça pourrait être très utile au niveau interblocage et partage de carte.

La coordination et la collecte

Pour la collecte, nous avons décidé, post exploration, de choisir un point de rendez-vous sur la topologie. Elle sera unique avec notre algorithme, qui nous permettra donc de pouvoir y aller sans se demander si les autres y arrivent. Notre méthode est très simple, pour chaque noeud, nous lui attribuons un score étant une moyenne pondérée entre la "distance avec chaque noeud" et "le nombre de noeuds adjacents à celui-ci". Cela permettrait, avec les bons paramètres, de rester relativement proche des potentiels coffres, d'éviter des rares interblocages (si seulement c'était suffisant), mais également d'avoir une chance d'attraper des gens n'ayant pas terminé leur exploration et leur partager la topologie complète !

L'objectif de ce point de rendez-vous est de se partager les charactéristiques une fois, et leurs connaissances sur les nouveaux coffres découverts ou perdus à chaque fin de recherche. Pour cela, le premier agent arrivé sur le point de rendez-vous deviendra la racine d'un pseudo flooding protocol ! Voici comment il marche :

- La racine arrive au point de rendez-vous. Dès qu'il croise quelqu'un, il lui demandera de rejoindre, et il serait forcé d'accepter. Il lui transmet sa carte.
- Lui et les autres nouveaux noeuds feront de même. Si un nouveau noeud apparait, alors on remonte l'information à la racine.
- Une fois que la racine constatera que tout le monde est là (grâce à une liste de nom passé au préalable), alors il fera descendre jusqu'au feuille un message pour les prévenir de se préparer à l'écouter.
- La première fois, il demandera aux noeuds de leur donner leur caractéristiques pour la collecte. La demande descend jusqu'aux feuilles, puis remontent avec un objet 'CharacteristicsFloodMessage'. La racine compile tout et leur renvoie pour que tout le monde ait tout.
- Il fait de même avec les connaissances des trésors. Là, nous ne sommes pas obligés de leur retransmettre les trésors mis à jour, étant donné qu'ils reviendront forcément au point de rendez-vous, et que la racine est celle qui donnera les coalitions.
- Enfin, la racine calculera les coalitions et leur enverra. Chacun a son rôle, et débute sa mission.

Globalement, bien qu'il y a un bon nombre de messages, je pense que c'est une méthode robuste afin de se coordonner correctement quand la zone de communication est resteindre à un seul noeud adjacent. Bien sûr, un système de Gossip aurait pu être meilleur si les agents sont prêts à ne pas être sûr que les autres agents soient synchronisés. On ne voulait pas prendre ce risque.

Pour la création des coalitions, nous avons décidé d'une règle toute simple. Une coalition doit forcément avoir à minima un agent Collecteur et un agentSilo, en plus des potentiels agent 'helpers' pour ouvrir les coffres. Notre plan initial était, si on avait réussi l'interblocage, de mettre un système de priorité de tel sorte que le collecteur soit toujours sur le coffre, accompagné à côté d'un silo, et enfin des autres. Comme ça, le collecteur peut ramasser et déposer directement le trésor, et repartir ensuite.

De plus, nous cherchions à limiter grandement les "expertises perdues" des coalitions faites. Pour cela, après avoir choisi un collecteur au hasard, nous cherchions la coalition qui permettait d'ouvrir un coffre. Si le coffre est déjà ouvert, alors on lui attribue juste un Silo. Pour choisir quel coffre prendre en premier, nous nous sommes basés uniquement sur la valeur de cette dernière, qu'on utilise par ailleurs dans notre système de priorité qu'on discutera après. Ce n'est clairement pas la manière la plus optimale de faire, parce qu'elle ne prend pas en compte les risques d'interblocage, la distance entre le point de rendez-vous et le noeud du trésor, ou même à quel moment le coffre a été aperçu. Mais étant donné que nous avions un silo tout prêt dès l'ouverture du coffre, l'aller serait éventuellement un peu plus long, mais la récolte aurait pu être très rapide. Mais ça reste très limitant. De plus, cet algorithme est assez lent, et exponentiellement couteux, parce qu'il consiste à calculer quasiment toutes les coalitions possibles après avoir choisi le silo et le collecteur. Ce n'est clairement pas viable dans un système multi-agents avec des centaines/milliers d'agents. Nous aurions pu faire quelque chose de moins optimisé, mais plus robuste et plus rapide, ça aurait été préférable.

Pour les autres agents qui n'ont pas eu de coalitions, nous avions pensé à faire des coalitions uniquement pour ouvrir des coffres, mais nous avons décidé de plutôt faire une équipe de "RE_EXPLORATION" pendant X secondes choisis (de même pour la récolte, ils ont un timer au cas où, et ils reviendront au point de rendez-vous). Cette équipe s'occupera de se balader au hasard, en faisant attention à ne pas embêter les récolteurs, afin de pouvoir détecter des mouvements suspects, des disparitions ou des ré-apparitions de coffre. Nous nous sommes dits que, techniquement, il y a aucun moyen de vraiment choisir si on a eu tous les trésors de la carte, même avec l'exploration initial, alors il ne ferait pas de mal de vérifier. Notamment si on a plus de trésors sous la main.

Une fois qu'une équipe a terminé sa mission, par exemple la récolte, elle passe en 'RE_EXPLORATION jusqu'à que le compte à rebours soit écoulé. Ils reviennent au point de rendez-vous, ils refont le partage de trésors et le calcul de coalition, et repartent.

Malheureusement, notre réalité est tout autre. Par exemple, notre premier 'flooding protocol' marche plutôt bien, mais pour une raison encore inconnue, le deuxième ne s'exécute pas. J'imagine qu'il y a un entrechoque entre interblocage et flooding, qui fait qu'un agent n'a pas forcément répondu, et met donc la racine en suspens.

Pour les interblocages, nous avions imaginé un système de réservation de noeuds, en gardant en mémoire la priorité de celui qui a fait cette réservation. Une fois que deux agens se rencontrent, ils s'envoient un objet indiquant les noeuds qu'ils souhaient parcourir, et étant donné qu'ils connaissent déjà les coalitions des autres, ils n'ont besoin que de l'info "as-tu terminé ta mission ?". Rien de plus.

La priorité est la suivante :

— Pour commencer, on vérifie si quelqu'un est bloqué. Si une personne ne peut possiblement pas de décaler (cul-de-sac), mais que l'autre si, alors il devient nécessairement prioritaire. On garde cette variable en mémoire.
— Sinon, on comparera les états. Nous aurons donc la priorité suivante : FLOODING > MOVING_TO_MEETING_POINT > COLLECT > RANDOM_WALK > EXPLORATION.
— En cas d'égalité pour chacune d'entre elles, sauf pour 'COLLECT', pour le moment, nous nous contentons de comparer les noms des agents pour avoir un choix déterministe.
— Dans le cas de deux agents en collecte de trésor, nous comparer en premier lieu la quantité initiale du trésor de leur mission s'ils ne proviennent pas d'une même coalition. Sinon encore leur nom. Dans le cas d'une même coalition, là c'est intéressant parce qu'il nous aurait permis d'exécuter notre plan d'avoir le collecteur directement sur le trésor, avec le silo à côté. Nous avons COLLECTEUR_PRINCIPAL > SILO_PRINCIPAL > OTHERS.

Nous comptions sur un système de transfert de priorité pour éviter les interblocages, mais cela n'a pas l'air suffisant, notamment après un flooding protocol où... tout le monde est réuni au même endroit. Un GOSSIP aurait été plus efficace si tout le monde aurait calculé un endroit unique avec leur nom par exemple, et que petit à petit, chacun se partageait leur information. Bien sûr, plus la topologie est petite, plus cette méthode est difficilement applicable, surtout en la présence d'un golem.

On a pu difficilement travailler sur le golem. Nous avions pas la prétendu d'essayer de le bloquer. D'où notre idée d'avoir des agents qui se déplacent au hasard sur la topologie, pour peut-être le ralentir, trouver d'autres coffres. Mais plutôt que du hasard, nous aurions pu être plus optimisé, de manière décentralisé. Etant donné que la topologie est statique, et que tout le monde la connait, chacun aurait pu s'attribuer un "cluster" qu'il fouillera, évitant un maximum les interblocages avec les autres, tout en évitant de visiter des noeuds déjà visités par ces confrères.

Conclusion

Ce projet nous a montré que les systèmes multi-agents sont très complexes à organiser. Avec une visbilité et des connaissances limités, il est primordial de savoir coopérer, de réfléchir à des plans et s'organiser correctement, avec de bons protocoles. Il est donc important d'avoir une vision claire sur les objectifs de chaque agent et du groupe, des potentiels difficultés qu'ils peuvent rencontrés, de l'environnement, et de leurs contraintes physiques.

Nous pensons que notre travail a beau avoir des idées pouvant être appliqués, il reste très critiquable vis-à-vis de certains choix d'implémentations que ce soit au niveau de la communication, et de certains choix d'algorithmes. De plus, le code n'est clairement pas optimisé, avec un gros raté au niveau de l'interblocage.

Mais il est totalement possible d'améliorer tout ça, en faisant une bonne balance entre performance, optimalité et robustesse. Le plus simple à changer serait d'après moi l'algorithme de choix de coalition. Même si on perd des patrouilles collecteurs, nous auront de nouveaux patrouilles RE_EXPLORATION, ce qui a également son avantage. Nous pouvons bien sûr essayer de faire fonctionner l'interblocage et enfin avoir un comportement qui fait ce qu'on lui demande, ramasser des coffres. Mais nous sommes restés trop bloqués sur des idées que l'on a eu au début du semestre, quand on y connaissait pas grand chose, c'était notre plus grande erreur. Rectifier le tir ensuite fut compliqué.

Ce projet nous a appris beaucoup de choses sur la coopération dans un environnement asynchrone, aux agents individuels mais coopératifs. On vous remercie pour toute l'aide apporté durant ce projet.