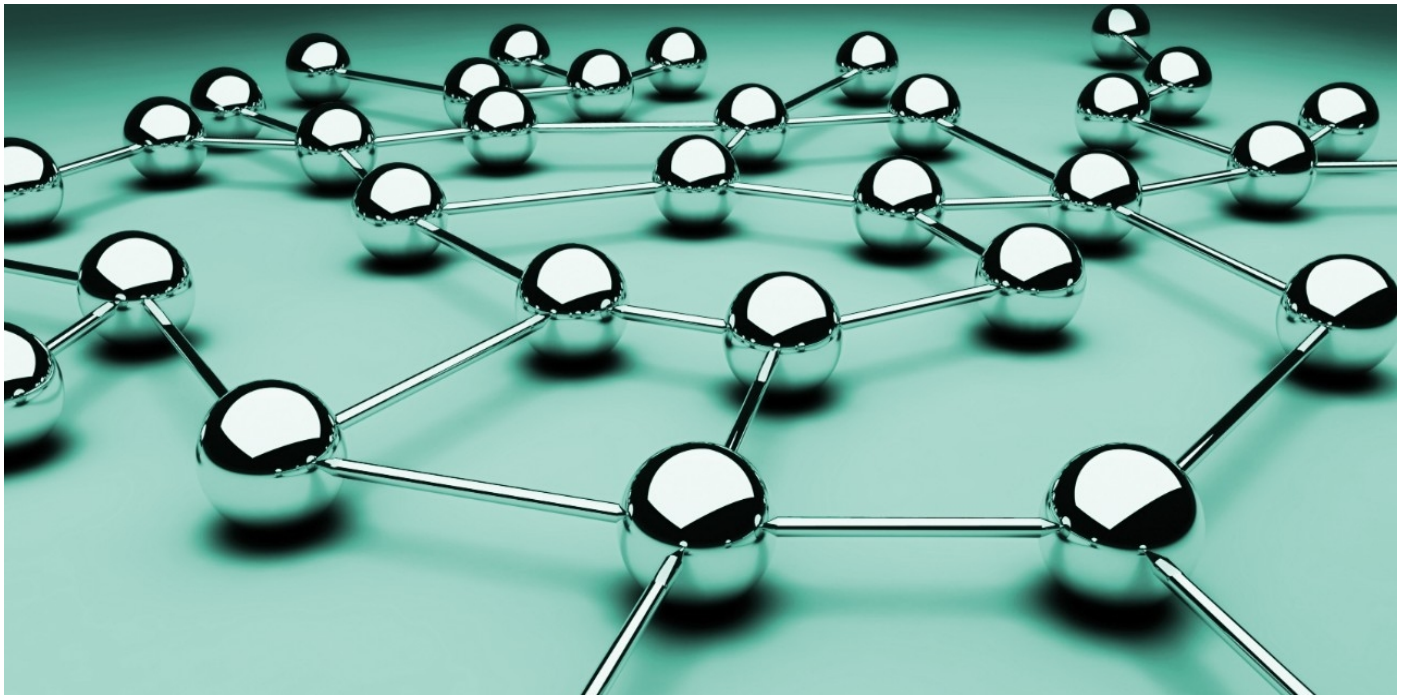
SORBONNE UNIVERSITÉ

FONDEMENTS DES SYSTÈMES MULTI AGENTS

Dédale : Rapport de projet



Lyna SAOUCHA, Anatole STANKOVIC
Groupe 1

Table des matières

Introduction

Dans le cadre de l'UE FoSyMa, nous avons travaillé pendant toute la durée du semestre sur un projet consistant à développer en binôme une variante multi-agent du jeu « Hunt the Wumpus », un des premiers jeux informatiques (Gregory Yobe, 1972). Dans la version d'origine, le joueur doit tuer un monstre, le Wumpus, caché dans une caverne tout en ramassant des trésors. Dans notre variante, une équipe d'agents coopératifs doit explorer un environnement inconnu. Des intrus sont présents sur la carte. Cette année le but de nos agents est d'explorer l'environnement et de ramasser des trésors disséminés dans ce même environnement. Nous allons implémenter un système multi agent dans lequel les agents explorent l'environnement dans un premier temps, puis réalisent la collecte des trésors trouvés dans un second temps.

L'environnement est représenté par un graphe, dont les nœuds sont les différentes positions possibles et les arcs les passages entre ces nœuds. Un nœud peut contenir un agent à la fois et peut détenir un trésor avec un type et une quantité définie. Nous avons développé différentes stratégies afin d'assurer une bonne coopération et une bonne coordination vers le cheminement de leur objectif. Nous utilisons le langage de programmation Java et la plateforme multi-agent JADE.

Chaque semaine, un nouveau concept a été ajouté au projet afin d'aboutir à une version multi-agent complète du jeu. La version finale de notre projet fera l'objet d'une soutenance et d'une démonstration sur machine. Ce rapport d'une dizaine de pages décrit les choix techniques effectués, les algorithmes utilisés ainsi que les protocoles de coordination et de communication mis en œuvre. Nous préciserons les garanties de terminaison et leur complexité (nombre de messages, temps, espace).

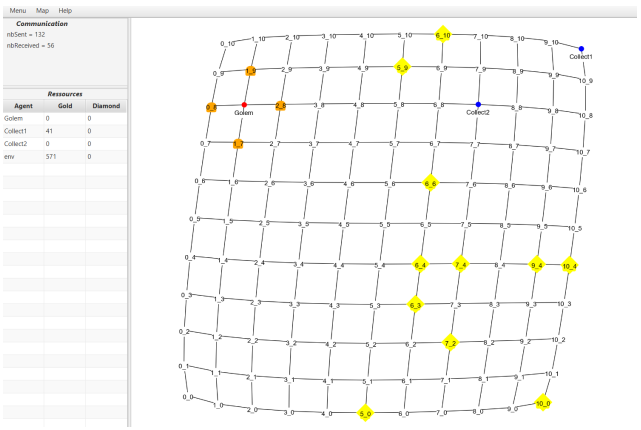


Figure 1 : Fenêtre de notre programme.

1. La gestion des behaviours de nos agents

Pour définir les comportements de nos agents (behaviours) nous avons utilisé le FSM (Finite State Machine). Ce schéma de comportement nous permet de lier et organiser plus aisément les behaviours des agents en définissant au préalable les transitions possibles ou non entre les différents comportements. On définit alors les transitions par défaut et enfin celles qui demandent de respecter certains critères. Pour simplifier la compréhension de ce que nous avons implémenté, nous avons dessiné un graphe où chaque état est un behaviour et chaque arc orienté une transition entre deux états.

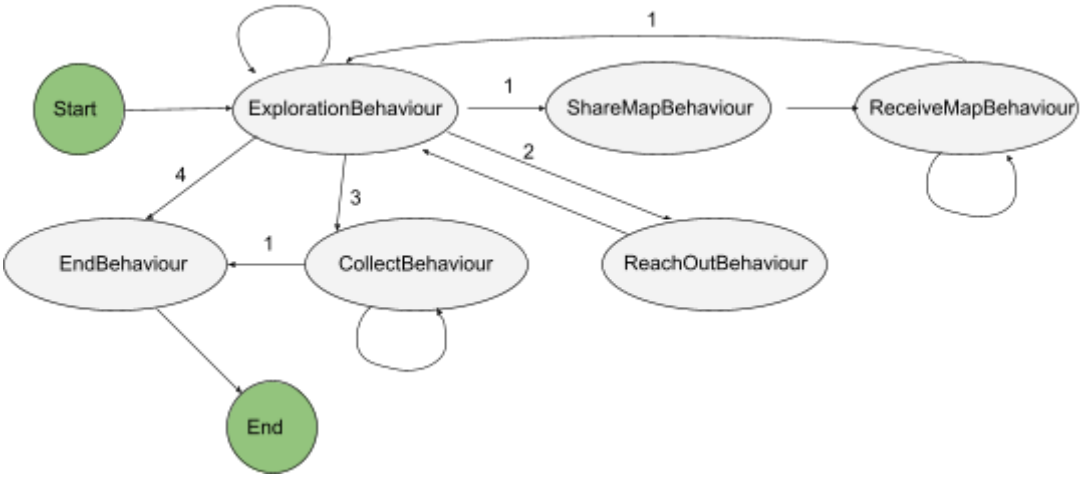


Figure 2 : Graphe des connections entre les behaviour

Notre FSM comprends 6 behaviours :

- **ExplorationBehaviour** : ce behaviour est responsable de la phase d'exploration. C'est dans cette étape qu'on choisit les nœuds suivants.
- **ShareMapBehaviour** : ce behaviour permet de partager sa carte avec les autres agents. Il est déclenché lorsqu'on reçoit un message "hello" ou lorsqu'on reçoit la carte de quelqu'un.
- **ReceiveMapBehaviour** : ce behaviour permet d'attendre et de réceptionner une carte partagé par un agent tiers. On y rentre après avoir envoyé sa carte. Si au bout de 10 tours de boucles, on n'a rien reçu, on retourne à l'exploration.
- **ReachOutBehaviour** : toutes les x secondes, on envoie un message "hello" aux autres agents pour envoyer notre carte. Dès qu'on reçoit un bonjour en retour, on passe dans ShareMap pour envoyer sa carte.
- **CollectBehaviour** : Une fois l'exploration finie, on passe à CollectBehaviour qui est responsable de la phase de collecte.
- **EndBehaviour** : comportement de fin de jeu.

Les transitions par défaut sont marquées avec des arcs sans valeurs. Les transitions sous condition sont marquées avec une valeur n écrite sur l'arc. De ce fait, on quittera ce behaviour pour emprunter l'arc de valeur n (avec la variable *this.exitValue* dans le code), sinon on quittera le behaviour pour aller dans le comportement par défaut.

2. L'exploration de la carte

2.1. Le Principe

Tout d'abord, il est important de préciser que nous avons un seul type d'agent appelé *OurAgent* qui s'occupe de l'exploration et de la collecte. Le comportement responsable de l'exploration de la carte est *ExplorationBehaviour*. Celui-ci a pour transition par défaut lui-même.

Au départ si la carte d'un agent est nulle on l'initialise. Puis l'agent commence l'exploration en explorant en priorité les nœuds qui se trouvent le plus proche de lui. Pour cela, il parcourt la liste des nœuds ouverts et choisit le nœud à distance la plus proche. A chaque nœud visité, il l'ajoute à sa carte développant ainsi sa connaissance de la topologie de la map. Si sur le nœud courant se trouve un trésor l'agent ajoute dans une Hashmap appelée "knowledge", ce nœud, le type de trésor présent et sa quantité. Chaque agent maintient ainsi sa connaissance de la répartition des trésors sur la map. L'agent ouvre également les trésors sur sa route. L'agent va explorer sans se soucier de ramasser les trésors dans un premier temps et une fois que tous les agents auront terminé l'exploration ils pourront ensemble passer à la collecte.

Lorsque l'agent est bloqué pour cause d'un obstacle (autre agent ou wumpus) en face de lui, il décide au bout d'un certain nombre de tours de boucle de choisir de façon aléatoire un nouveau nœud objectif. Il va calculer le plus court chemin vers ce nœud puis y aller. De cette façon, on évite les blocages à l'infini et on contourne les obstacles. Dans le cas où un plus court chemin n'est pas trouvable jusqu'au nœud en question, on décide au bout d'un certain nombre de tours de boucle de "fermer" le nœud et de passer à autre chose. Il s'agit peut être d'un nœud qui a été envoyé dans une carte mais pour lequel on ne connaît pas de chemin. Cette implémentation permet d'éviter une perte de temps considérable ou que l'agent arrête l'exploration.

2.2 Forces

L'avantage de ce système est sa rapidité et la fluidité qu'il offre quant à l'exploration. A chaque fois qu'un agent se déplace pour aller vers un nœud qui n'a pas encore été exploré, il prend en priorité un nœud non-visité adjacent à sa position. En revanche, s'il n'y a pas de nœud adjacent non-visité, il va calculer pour tous les nœuds à explorer celui dont le chemin est le plus court et se diriger vers ce point. L'implémentation permet aussi de gérer à tout moment les messages reçus et de choisir les prochains mouvements de case en fonction des informations reçues. Nous avions au début opté pour une stratégie de choix du nœud aléatoire qui était beaucoup plus lente et ne nous permettait parfois pas d'aller au bout de l'exploration. Le partage d'informations entre les agents est équilibré et participe au bon fonctionnement du comportement. Il termine et la transition vers les autres comportements est fluide.

2.3 Limite, complexité, optimalité et critère d'arrêt

La génération du chemin utilise l'algorithme de Dijkstra, ce qui est optimal pour une génération de plus courts chemins. Au départ, l'agent choisissait un nœud au hasard parmi la liste des nœuds à explorer, ce qui était beaucoup plus lent mais beaucoup moins coûteux en termes de complexité. En effet, la fonction random de Java à une complexité en O(1) tandis que notre version se rapproche du O(n) avec n la longueur de *opennodes*, la chaîne de nœuds ouvertes et cela pour chaque tour du behavior. D'un autre côté, lorsque notre agent est bloqué et doit générer un nœud avec le random, il choisit parfois un nœud lointain, qui le fera boucler beaucoup de fois sur le comportement pour y parvenir, ce qui diminue son optimalité.

On peut se retrouver rapidement avec un comportement très lourd en termes de code, ce qui n'est pas vraiment optimal car il est préférable d'avoir des comportements plus légers mais plus simples et de les appeler au bon moment. Malgré tout, notre exploration ne comporte pas d'incohérences ou de comportements non désirés.

Le critère de terminaison pour la collecte est que tous les agents connaissent la topologie complète de la carte. On peut considérer que l'exploration est plutôt optimale, cependant nous n'avons pas résolu la question de la chasse du wumpus et de son blocage, par manque de temps, nous avons préféré nous concentrer sur réaliser l'exploration et la collecte la plus fonctionnelle possible donc nous voulons souligner que notre code évite le wumpus mais ne le bloque pas.

```
Collect2 : EXPLORATION BEHAVIOR
Collect2 : Je suis à 8_2 et j'observe lobs ->  [<8_2, [<Gold, 82>]>, <7_2, []>, <8_1, []>, <8_3, []>, <9_2, []>]
Collect2 : J'ajoute obj ->  <8_2, [<Gold, 82>]> à mes observations.
Collect2 : il y a un gold <Gold, 82>
Collect2 : J'ajoute NodeId ->  8_2 a ma carte et mon next node est : null
Collect2 : J'ajoute NodeId ->  7_2 a ma carte et mon next node est : null
Collect2 : J'ajoute NodeId ->  8_1 a ma carte et mon next node est : null
Collect2 : J'ajoute NodeId ->  8_3 a ma carte et mon next node est : 8_3
Collect2 : J'ajoute NodeId ->  9_2 a ma carte et mon next node est : 8_3
Collect2 : Mes noeuds ouverts -> [0_10, 1_10, 2_10, 4_9, 3_10, 4_8, 4_7, 5_7, 7_6, 6_7, 8_5, 8_4, 8_3, 9_1, 9_2]
Collect2 : I received a Map from ( agent-identifier :name Collect1@Ithaq  :addresses (sequence http://LAPTOP-3VD9EUPC:7778/acc ))
```

Figure 3 : Une trace d'exécution d'une exploration d'un de nos agents dans la console.

3. La communication et la coordination

3.1 Le principe

Les agents communiquent dans différents cas pour optimiser les différents comportements. A chaque itération, l'agent regarde dans sa boîte au lettre s'il a reçu un message. Il existe plusieurs types de messages possible :

- L'agent reçoit un hello : c'est une invitation à envoyer sa carte. Il migre vers le comportement *ShareMap* pour envoyer sa carte à celui qui lui a envoyé le message.
- L'agent reçoit une carte : il reçoit la carte et va à son tour envoyer sa carte.
- L'agent reçoit des infos sur la collecte : il les fusionne avec les infos qu'il a déjà sur les emplacements de trésors.

En effet, prenons le cas de l'exploration, les agents vont communiquer leur carte (topologie de la map) et recevoir à leur tour une carte pour ainsi mettre à jour leur propre carte. Ce comportement va fortement réduire le temps d'exploration des agents. Il y a plusieurs scénarios possibles :

Scénario 1 : toutes les dix secondes il envoie "Hello" à tout le monde et à chaque tour, il regarde s'il n'a pas reçu de message. S'il reçoit un "hello" en retour, il part dans *ShareMapBehaviour* pour envoyer sa carte.
Scénario 2 : il reçoit un "Hello". Il va envoyer sa carte et aller dans *ReceiveMapBehaviour* pour attendre d'en recevoir une à son tour.
Scénario 3 : il reçoit une carte (après avoir sûrement envoyé un "Hello" un peu plus tôt), il part dans *ShareMapBehaviour* pour envoyer sa carte puis dans *ReceiveMapBehaviour* pour atteindre une carte en retour.

Concernant la gestion de l'inter-blocage entre les agents, c'est-à-dire lorsque deux agents veulent se déplacer sur le même nœuds, nous avons décidé que si deux agents se bloquent pendant trop longtemps, chaque agent va de manière aléatoire trouver un nouveau nœud pas encore parcouru, calculer son plus court chemin et y aller. De ce fait chaque agent, va vouloir emprunter une case différente et si ce n'est pas le cas on recommence jusqu'à ce qu'ils ne se bloquent plus. Ce comportement est très efficace et règle de manière efficace et en un court temps d'exécution le problème de l'inter-blocage entre les agents.

3.2 Forces

Lorsque notre agent est dans *ReceiveMapBehaviour*, il attend 10 secondes pour recevoir une carte en retour. Dans le cas où il ne reçoit rien, il retourne à l'exploration. De ce fait, on gère le cas où il n'attend pas vraiment de carte puisqu'il en à déjà reçu une sans en demander, mais aussi le cas où il ne reçoit pas la carte pour une raison quelconque.

Le temps d'envoi pour le "Hello" est configurable dans *OurAgent*. On peut donner 10 secondes, ce qui est suffisant lorsqu'on ne laisse pas de temps de latence pour l'exploration mais lorsqu'on ajoute une boucle de latence (qui permet de voir ce que les agents font pendant l'exploration), il est plus judicieux d'augmenter ce temps dans la variable.

Lorsqu'un agent finit l'exploration, il envoie à tous le monde la hashmap "knowledge" comprenant la position des trésors à tous les autres agents. Dans le code on convertit la Hashmap en String pour pouvoir l'envoyer aux agents, et à la réception par les agents elle est de nouveau convertie en Hashmap. Cette partie a été codée en dur en prenant la construction de la Hashmap de Jade. Cela permet au dernier agent qui termine son exploration de peut-être prévenir le déplacement par le wumpus des trésors. L'avantage de l'utilisation de la Hashmap pour le partage d'infos sur la collecte est l'utilisation de la fonction *put()* car si une clé existe déjà dans la hashmap de base, la nouvelle valeur de quantité/type de trésor viendra effacer l'ancienne, ce qui est pratique dans le cas où le wumpus a déplacé des objets ou si un des agents a amassé un trésor.

3.3 Limite, complexité, optimalité et critère d'arrêt

L'inconvénient principal de la communication est la mauvaise gestion des destinataires. En effet, même si l'envoi des messages n'a pas lieu à chaque tour de boucle, chaque message est envoyé à tout le monde. Une façon plus intelligente de procéder serait de, par exemple, envoyer à un seul agent à chaque fois et que les communications se fassent plutôt entre deux agents et non pas de façon centralisée autour de l'agent qui dit "Hello".

Lors de la phase de Collecte, la quantité de messages est multipliée car on vérifie à chaque tour de boucle s'il y a une nouvelle observation sur notre nœud position et on renvoie à chaque fois un message. C'est très mauvais en termes de complexités car on monte rapidement à des centaines de messages envoyés en quelques secondes. Une solution serait de mettre un timer tout comme la *ReachOutBehavior*, qui permettrait d'effectuer cet envoi toutes les 10 secondes par exemple. Cependant les infos ne seraient plus exactes et on ne pourra pas à chaque fin de tour de boucle garantir à tous les agents que tous les trésors sont bien encore à la bonne place.

4. La collecte des trésors

4.1 Le principe

Lorsque l'exploration est terminée, les agents peuvent débuter la collecte des trésors. On commence par choisir le type de trésor que l'agent peut ramasser. Pour connaître le type de trésor, on utilise une variable *type_collect* qu'on instancie au début de la boucle. Chaque agent ramasse un seul type de trésor, nous avons donc envisagé que les agents ayant la plus grande capacité pour un type de trésor le ramassait. Dans le sujet cependant il est précisé qu'un agent ramasse le type de trésor du premier trésor qu'il rencontre lors de la phase de collecte. Nous avons pensé à plusieurs conditions possibles :

1. Si la capacité du sac à dos pour les types *Diamond* ou *Gold* est nulle, on attribue le second type à l'agent. (ce qu'on a utilisé dans notre code).
2. Si un agent a une capacité *Gold* supérieure à la capacité *Diamond* et l'autre a une capacité *Diamond* supérieure à la capacité *Gold*, on attribue à chaque agent le type dont il a la plus grande capacité (pas implémenté).

A partir du *knowledge* comprenant l'ensemble des nœuds où les agents ont trouvé des trésors, ils créent une liste de nœuds qui correspondent à tous les nœuds qui contiennent des trésors de même type que leur type attribué. De la même façon que lors de l'exploration, ils vont calculer celui qui est le plus proche et se diriger vers celui-ci. À chaque fois qu'un agent parcourt un nœud et constate que la quantité de trésor sur ce nœud à changé par rapport à ce qu'il connaît, il ajoute cette observation à sa *knowledge* et il envoie à la fin du tour de boucle une nouvelle version du knowledge à tous les agents. Lorsqu'un agent ramasse une quantité de trésor sur un nœud, il renvoie aussi le knowledge pour indiquer ce changement à tous les agents.

4.2 Forces

Le principal avantage de la collecte c'est l'utilisation de la stratégie d'exploration pour essayer d'aller le plus rapidement possible récolter les trésors et la mise à jour constante des informations de tous les agents malgré le Wumpus qui peut empêcher la collecte.

4.3 Limite, complexité, optimalité et critère d'arrêt

La limite principale de notre comportement est le manque de stratégie concernant l'attribution des trésors. Une attribution optimale prendrait en compte la knowledge des agents ainsi que les capacités de tous les sac à dos et utiliserait la programmation linéaire pour répartir au mieux les trésors selon les capacités de chacun. Il faudrait ensuite donner à chaque agent sa liste de nœuds personnalisée pour que chacun puisse amasser les trésors avec un résultat collectif optimal. Il est aussi possible avec notre répartition que tous les agents aient le même type de trésors selon la capacité de leur sac à dos et de manquer alors une partie de trésor. Supposons par exemple que tous nos agents aient une capacité de *Gold* supérieure à la capacité de *Diamond* : tous les agents se verront attribuer le type *Gold* et personne ne pourra ramasser les types *Diamond*. De plus, le problème de la communication énoncé dans la partie précédente est un réel problème quant à la complexité du comportement de la collecte.

Le critère de terminaison pour la collecte c'est que soit tous les agents ont atteint leur capacité maximum pour les trésors ou sinon qu'il n'y a plus de trésors sur la carte. Dans ce cas, les agents rejoignent le *EndBehavior* qui permet d'afficher sur la console que nous sommes arrivés la fin du système.

Conclusion

Dans l'ensemble, notre projet répond à l'objectif de base, qui est de mettre en place un système avec plusieurs agents qui arrivent à communiquer pour explorer un environnement et collecter des trésors.
Nous avons surtout travaillé avec deux, trois agents et sur des grilles, notre code est encore à évaluer sur des instances plus nombreuses et des cartes plus originales pour vérifier si les stratégies implémentées fonctionnent correctement. Malheureusement, notre implémentation ne prend pas en compte le Wumpus et ne cherche à aucun moment à le bloquer mais l'évite seulement. Nous avons fait le choix de nous concentrer exclusivement sur la Collecte et l'Exploration dans le temps imparti.

Le projet peut sembler simple au départ, mais il est en réalité très complexe, qui prend en compte énormément de paramètres. Même si nous pensons avoir une base solide, il y a encore nombre d'améliorations à faire en termes de stratégie, d'efficacité et d'optimalité. Le premier point important à développer est le blocage du Wumpus, suivi de l'optimisation de la communication et de l'affectation des types de trésors lors de la collecte.

Nous remercions chaleureusement toute l'équipe pédagogique qui nous a aidé durant le processus de conception et de développement du projet et notamment Cédric Herpson pour sa grande disponibilité et ses conseils durant le semestre.