
FOSYMA

Système multi-agent

---

# Rapport

---

***Réalisé par :***
Kim Saïdi
Saad Mousstaid

***Encadrés par :***
Cédric Herpson
Aurélie Beynier
Vincent Corruble

Avril 2024

# Table des matières

# 1 Introduction

## 1.1 Sujet

Ce projet étant une variante du jeu 'Hunt the Wumpus' s'inscrit dans le cadre de l'UE FoSyMa du M1 ANDROIDE. Dans cette variante, l'objectif est de développer un système multi-agent où les agents doivent bloquer le Golem, en mettant de côté la recherche de trésors.

Le jeu se déroule dans un environnement représenté par un graphe, où les nœuds correspondent aux positions possibles des agents et les arètes aux passages entre deux positions. Dans un premier temps, nos agents doivent explorer la map de manière efficace. Puis, dans un second temps, ces agents doivent coopérer pour chercher et bloquer les golems présents sur la map.

Pour atteindre cet objectif, nous avons développé différentes stratégies d'exécution des agents afin d'assurer une coopération efficace. Nous utiliserons le langage de programmation Java et la plateforme multi-agent JADE pour mettre en œuvre notre système. Les paramètres expérimentaux incluent la taille et type de la carte, le nombre d'agents et de Golem, le rayon de vision et de communications des agents, ainsi que le rayon de l'odeur du Golem.

## 1.2 Explication générale

Pour ce projet, Nous avons adopté une structure de comportement basée sur une Machine à États Finis (FSM). Cette approche permet une gestion séquentielle des actions des agents, facilitant ainsi les transitions entre les comportements de manière automatisée. Notre FSM comprend 11 comportements distincts, parmi lesquels trois sont prédominants : ExploCoop, dédié à l'exploration, Patrol, axé sur la phase de chasse, et BlockGolem, permettant de bloquer un golem. Les autres comportements servent à faciliter la communication entre ces différentes classes comportementales.

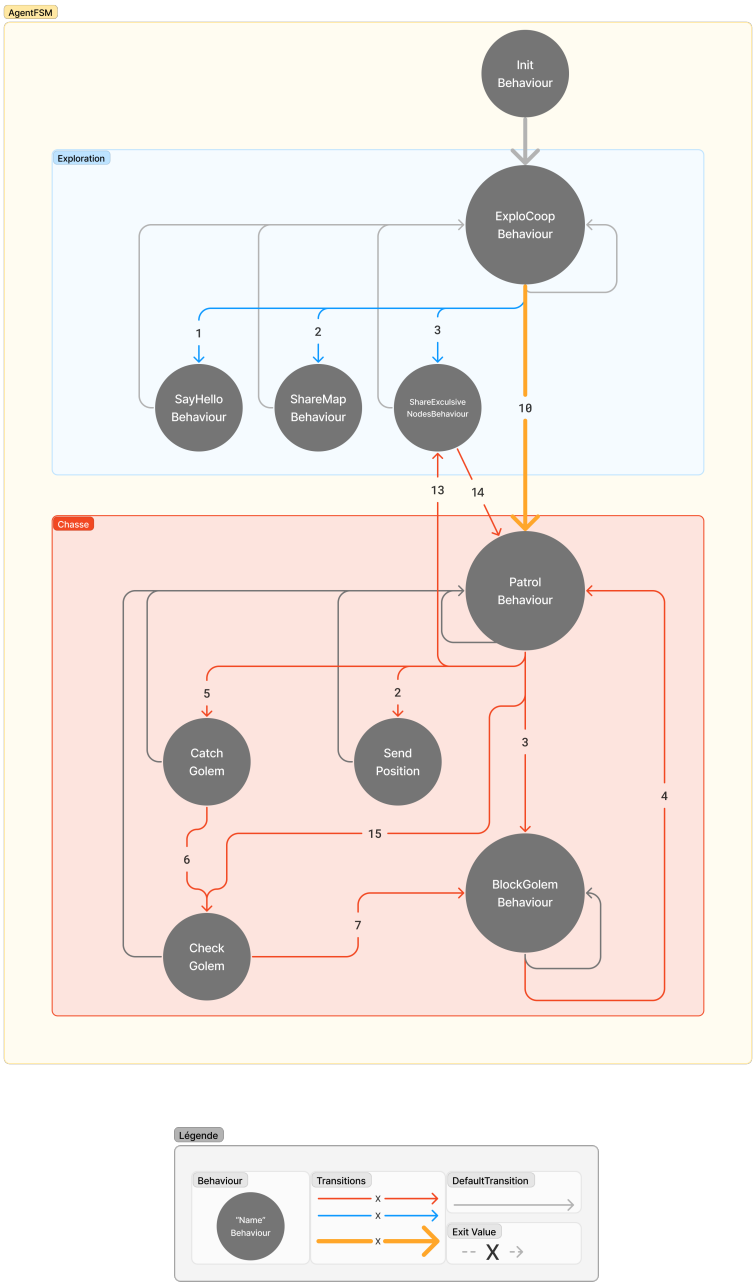


FIGURE 1 – Structure de notre FSM

# 2 Choix d'implémentation

## 2.1 Exploration

### 2.1.1 Le Principe

Les agents débutent leur exploration de la carte en priorisant les mouvements en profondeur, suivis par une stratégie en largeur. Dans cette approche, chaque agent concentre ses efforts sur l'exploration des nœuds ouverts les plus proches. Lorsqu'il n'y a plus de nœuds ouverts dans son voisinage immédiat, l'agent se déplace vers le nœud ouvert le plus proche pour poursuivre son exploration, ou alors vers un noeud aléatoire selon le cas d'une réception de certains messages que l'on détaillera par la suite.

Ce système présente l'avantage de centraliser ses opérations autour d'un comportement principal : l'exploration efficace de la carte. Il permet également de traiter efficacement les messages reçus d'autres agents avant de procéder au prochain mouvement. Cela nous permet de filtrer les messages non pertinents pour notre objectif immédiat, nous concentrant ainsi initialement sur une exploration rapide de la carte. Cette approche prépare ensuite la transition vers la phase de chasse une fois l'exploration terminée.

### 2.1.2 Communication

À chaque nombre prédéfini d'actions, représentant des mouvements, effectuées par un agent (par exemple, tout les 5 mouvements), celui-ci diffuse un message *HelloProtocol*. L'objectif de ce message est de signaler sa présence et de permettre à d'autres agents à proximité de répondre en envoyant leur carte s'ils communiquent pour la première fois. Dans le cas contraire, seuls les nœuds exclusifs de l'agent recevant *HelloProtocol* sont envoyés, ce qui permet éventuellement de réduire au maximum la taille du message, en fournissant toutes les informations manquantes liées à la topologie de la carte pour le premier agent (explication du mécanisme dans la section suivante). On peut voir dans la

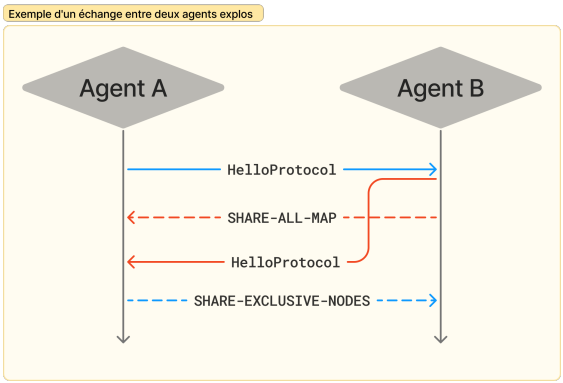


FIGURE 2 – Exemple d'un échange entre deux agents Explorateurs

figure ci-dessus un échange dans lequel les agents A et B envoie un *HelloProtocol* quasi-simultanément et réussissent à s'échanger proprement leurs maps d'un système de priorité des messages reçus par un agent dans la méthode `checkReceivedMessage()`. Chaque agent vérifie d'abord s'il reçoit un *SHARE-ALL-MAP*, ensuite *SHARE-EXCLUSIVE-*

*NODES*, et enfin *HelloProtocol*. L'agent passe au traitement d'un autre protocole seulement après avoir traité tous les messages reçu pour chaque protocole de partage.

Ce mécanisme favorise une communication efficace entre les agents tout en minimisant la complexité des données échangées.

### 2.1.3 Stratégies et limites

**Éviter les spams**

* Une diffusion régulée : Chaque agent tente d'initier une échange avec ces coéquipiers à proximité avec un *HelloProtocol* toutes les `nb` actions. Pour se faire, on incrémente un compteur initialisé à 0 pour chaque agent qui sera incrémenter à chaque appel de la méthode `action()` dans ExploCoopBehaviour. Lorsque le compteur atteint un multiple de ce nombre, l'agent passe dans SayHelloBehaviour.
* Gérer les rencontres récentes : Lorsqu'un agent A reçoit un message contenant le protocole *HelloProtocol* d'un agent B, l'agent A ajoute l'agent B dans sa liste des agents qu'il a rencontré récemment afin d'ignorer d'éventuelles spams de la part de ces agents. L'agent A retire l'agent B de la liste des agents spameurs après un nombre fixé de mouvements (dans notre cas 6) pour reprendre les communications avec lui.

**Partage de la Map**

Si l'agent A reçoit un *HelloProtocol* de la part de l'agent B, il peut adopter l'un des deux comportements ci-dessous en fonction d'un critère de reconnaissance. Chaque agent maintient une liste `"list_friends_map"` qui contient des couples (agent, carte) représentant les agents rencontrés précédemment ainsi que les cartes connues de chaque agent. Cette liste est mise à jour au fur et à mesure des échanges entre les agents.

* Partage complet de la carte (`ShareMapBehaviour`) :
  Lors de leur première communication, l'agent A transmet intégralement sa carte à l'agent B. En conséquence, l'agent A ajoute l'agent B à sa liste `"list_friends_map"`, accompagné de la carte partagée.

* Partage des noeuds exclusifs (`ShareExclusiveNodesBehaviour`) :
  Si l'agent B figure dans `"list_friends_map"`, alors l'agent A partage une carte ne contenant que les noeuds que l'agent B ne connaît pas encore, les noeuds que l'agent B connaît mais avec des arêtes manquantes, ainsi que les noeuds fermés mais toujours ouverts pour l'agent B.

**Limites**

* **Exploration en file indienne** : Après avoir réussi un échange en recevant la map d'un autre agent, notre premier agent génère parfois un chemin différent de celui qu'il empruntait jusqu'ici. Dans certaines situations, le nouveau chemin peut conduire notre agent à un nœud parmi ceux que l'autre agent connaît déjà et vient de parcourir s'ils étaient proches durant l'échange, ce qui peut parfois les maintenir à proximité, voire se retrouver en file indienne.
* **Message non réceptionné** : Dans de rares cas où un agent A envoie sa map ou ses noeuds exclusifs à un agent B, se dernier peut par malheur ne pas réceptionner le message, par conséquent l'agent A met à jour sa `"list_friends_map"` avec

une map erronée. Celà peut altérer les prochains échanges entre ses deux agents. On essaie d'éviter au maximum ce cas de figure en priorisant les messages reçus contenant des maps.

L'exploration prend fin lorsque tous les nœuds ouverts ont été explorés, et à ce moment-là, notre agent passe directement en mode de chasse.

## 2.2 Chasse

Une fois l'exploration terminée, les agents adoptent le comportement de chasseur. Notre stratégie principale repose sur l'appel à l'aide des chasseurs environnants. Si l'un d'eux parvient à localiser un Golem, il envoie un message aux chasseurs de son rayon de communication pour demander de l'aide. Dans notre implémentation, les chasseurs ont principalement trois modes : un mode PatrolBehaviour qui sert à trouver des Golems, un mode CatchGolem qui s'active lorsqu'un chasseur vient en aide à un autre pour bloquer le Golem. Et enfin, un mode BlockGolem qui permet de bloquer le Golem. Nous allons voir ci-dessous comment ces agents parviennent à chasser grâce à une bonne communication.

### 2.2.1 Mode Patrouille

#### Déplacements

Le but d'un chasseur est d'identifier la position du Golem, mais il est impossible de déterminer avec exactitude sa position. L'idée est donc de procéder par élimination en réduisant au maximum les possibilités.

* Position des agents : On récupère la position des agents qui sont dans notre rayon de communication. Puisque les agents sont asynchrones, tous ne finissent pas l'exploration à la même vitesse. Les chasseurs récupèrent donc la position des explorateurs avec le *HelloProtocol* pour ne pas les confondre avec un Golem.
* Position des odeurs du Golem : On récupère les noeuds, parmis les noeuds observables où l'odeur du Golem est présente. Cette liste est vide si le Golem est loin ou sans odeur.

Une fois ces informations rassemblées, le chasseur peut désormais choisir quels nœuds éviter (nœuds sans odeur ou occupés par d'autres agents) et quels nœuds privilégier (nœuds avec odeur). Ce procéder nécessite que les chasseurs envoient en permanence leur position.

#### Chasse collective

Si un chasseur ne parvient toujours pas à accéder à son prochain nœud malgré toutes ces informations, c'est qu'il est certainement tombé sur un Golem ! Dans un premier temps, il envoie un message *NeedHelpProtocol* communiquant aux autres chasseurs à proximité la position du Golem, puis adopte le comportement BlockGolem. A l'inverse, si un chasseur reçoit ce message, il va essayer d'attraper le Golem avec CatchGolem.

### 2.2.2 Mode Block / Catch

**Immobiliser un Golem**

Ce comportement vise à vérifier si l'agent continue de bloquer le Golem (il ne peut pas accéder à la position du Golem) et, si tel est le cas, le chasseur bloque en envoyant un message *IAmAnAgentBlockGolemProtocol* en continu, qui communique la position du Golem ainsi que de sa propre position (nous verrons un peu plus tard, l'importance pour un chasseur qui bloque un Golem d'envoyer sa propre position). S'il ne le bloque plus, il retourne en PatrolBehaviour.

**Attraper un Golem**

Si un chasseur reçoit un message d'appel à l'aide, il adoptera le comportement CatchGolem. Ainsi, le chasseur parcourt le plus court chemin pour atteindre la position du Golem. L'objectif est de ne pas atteindre la position du Golem. S'il y parvient, cela signifie que le Golem est déjà parti et le chasseur retourne patrouiller. S'il est bloqué, il faut déterminer s'il s'agit du Golem ou d'un autre agent. Pour ce faire, il vérifie à l'aide de CheckGolem. Dans ce comportement, le chasseur compare la position du Golem avec ses nœuds observables. Si la position du Golem fait partie de ses nœuds observables, alors le chasseur est indispensable pour bloquer le Golem et adopte BlockGolem, sinon il retourne en PatrolBehaviour.

### 2.2.3 Communication

**Agent ou Golem ?**

Le problème principal de cette stratégie est de bien savoir différencier les agents des Golems. En effet, lorsqu'un chasseur bloque un Golem, celui-ci n'envoie plus sa position mais seulement un message *IAmAnAgentBlockGolemProtocol*. Donc, dans PatrolBehaviour, lorsqu'un chasseur est bloqué par quelque chose, il est nécessaire de vérifier si c'est un agent en Mode Block ou simplement un Golem. Ainsi, si je suis un chasseur H2 bloqué par quelque chose et que j'ai reçu ce message, plusieurs possibilités sont à prendre en compte :

* Je suis entouré d'agents qui me prennent pour un golem : J'envoie un message *JeNeSuisPasUnGolemProtocol* contenant ma position qui sera réceptionné dans BlockGolem. Si tel est le cas, les agents bloquant vérifieront à la position reçu s'il s'agit du Golem qu'ils sont entrain de bloquer. En effet, avec un grand rayon de communication, il se peut que des agents reçoivent ce message mais bloquent bien un Golem ! Mais s'ils se sont en effet trompés, ils retourneront en patrouille.
* Un agent bloque un Golem et ma position peut être utile (Figure 3) : Dans ce cas là, j'essaye de CatchGolem et si ma position est inutile, je retourne en patrouille.
* Un agent bloque un Golem et ma position est utile (Figure 4) : Dans ce cas là, je vais BlockGolem.

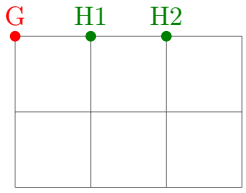


FIGURE 3 – Je peux être utile

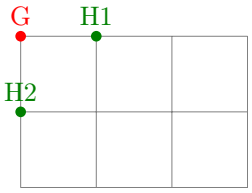


FIGURE 4 – Je suis utile

**De quel Golem parle-t-on ?**

Si plusieurs Golems se retrouvent sur la map, il faut faire attention à ne pas les confondres. Revenons à la situation du paragraphe précédent. Je suis un chasseur H3 bloqué par quelque chose. Si le rayon de communication des agents est grand, je peux très bien être bloqué par un Golem G2, mais quand même recevoir un message *IAmAnAgentBlockGolemProtocol* d'un agent qui bloque un Golem G1 un peu plus loin sur la map (Figure 6). C'est donc dans ce cas de figure qu'il est important de réceptionner la position de l'agent en Mode Block.

Si sa position correspond à mon noeud qui bloque alors ce n'est pas un Golem, j'essaye d'aider et adopte le Mode CATCHGOLEM (Figure 5). En revanche, si les positions ne concordent pas, alors je suis sur la piste d'un autre Golem (Figure 6).

*Remarque* : Dans la Figure 5, le chasseur va tenter d'aider mais se rendra compte (dans CATCHGOLEM puis dans CHECKGOLEM) que son aide est inutile.

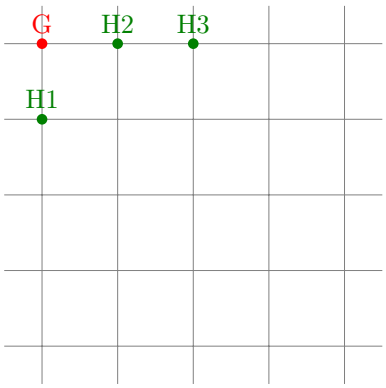


FIGURE 5 – Le même Golem

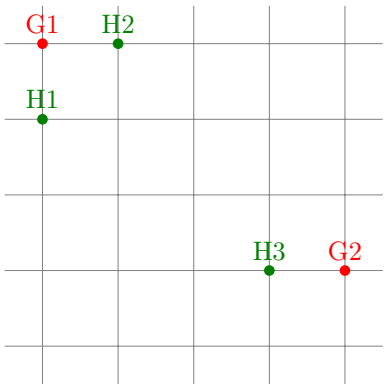


FIGURE 6 – Deux Golems différents

# 3 Résultats

## 3.1 Points fort

Nos stratégies sont fondamentalement simples et générales, ce qui les rend efficaces sur différents types de cartes, qu'elles soient grandes ou petites. De même, peu importe les caractéristiques des agents, qu'ils aient un petit ou un grand rayon de communication, ou qu'il y ait un ou plusieurs Golems, nos agents parviennent presque toujours à les bloquer de manière très rapide. Nous examinerons cependant les limites de cette approche dans la

section suivante, afin de préciser les situations où elle pourrait être moins efficace. Voici un exemple de résultat :

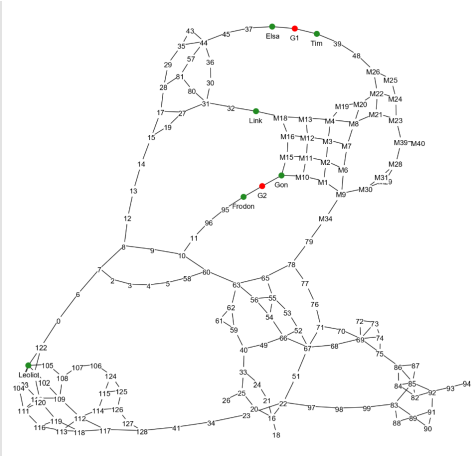


FIGURE 7 – Graph Map

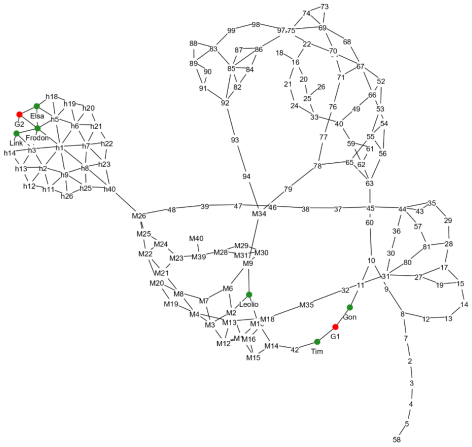


FIGURE 8 – Draft Map

## 3.2 Limites et Complexité

### 3.2.1 Cas où ça échoue

La principale limite de notre code réside dans les caractéristiques du Golem. En effet, lorsque celui-ci possède une odeur de taille supérieure à 1, nos agents ont presque toujours du mal à converger. Dans le meilleur des cas, nos agents parviennent à suivre efficacement l'odeur des Golems, mais échouent à les bloquer.

*Remarque* : Notre code fonctionne aussi bien que les Golems aient une odeur de 1 ou pas d'odeur du tout, et les agents finissent toujours par les bloquer. Nous n'avons donc pas vraiment trouvé la cause de ce problème.

### 3.2.2 Un programme coûteux

Notre implémentation n'est pas très optimale car les agents s'envoient constamment de nombreux messages, ce qui est très coûteux. De plus, il n'y a pas de critère d'arrêt clair pour la partie de la chasse. Tant que les agents bloquent un Golem, ils continuent d'envoyer des messages indéfiniment.

Quelques points vont tout de même alléger le code :

* Calcul du plus court chemin : Dans CATCHGOLEM, ce calcul est fait avec l'algorithme de Dijkstra.
* Partage des Maps : Pendant l'exploration, les agents ne transmettent pas l'intégralité de leur carte, mais seulement la partie nécessaire. Cette approche est particulièrement utile lorsque la carte est volumineuse et donc coûteuse à partager. De plus, ils ne la partagent que sous condition de recevoir un message, ce qui réduit la complexité.

# 4 Points à améliorer

## 4.1 Pour la partie CatchGolem

Pour capturer un Golem, un agent doit calculer le chemin le plus court pour atteindre sa position. Cependant, s'il rencontre un autre agent en cours de route, il retourne patrouiller. Une amélioration potentielle serait de calculer tous les chemins menant à la position du Golem afin de garantir si oui ou non le chasseur peut le bloquer.

## 4.2 Pour la partie Stratégie

### 4.2.1 Optimiser la chasse

Un point que nous aurions pu implémenter avec un peu plus de temps est le partage de la liste des odeurs. En effet, il aurait été intéressant de prévoir que les agents envoient la liste des odeurs du Golem depuis ses nœuds observables aux autres agents autour de lui.

* Point fort : Si un agent n'est pas sur la piste d'un Golem, recevoir cette information peut le conduire dans la zone du Golem.
* Limite : Alourdit le code car les agents doivent envoyer la liste des odeurs chaque fois qu'ils les observent.
* Point à faire attention : Si un agent est déjà sur la piste d'un Golem, cette information n'est pas utile.

### 4.2.2 Optimiser le blocage

Nous avons essayé de créer une fonction permettant de récupérer les nœuds voisins d'un nœud en particulier. Malheureusement, nous n'avons pas réussi à y parvenir. Si nous avions réussi, le blocage des Golems aurait été plus efficace. En effet, une fois que nous aurions récupéré la position du Golem et que nous aurions identifié les agents le bloquant avec le protocole *IAmAnAgentBlockGolemProtocol*, il aurait suffi de nous positionner sur les nœuds restants.

# 5 Conclusion

En conclusion, notre projet répond à la problématique initiale, qui est de bloquer des Golems sur différents types de cartes. Globalement, notre implémentation se révèle efficace et fonctionne très bien dans certaines conditions, notamment lorsque l'odeur du Golem est limitée à un ou moins. Cependant, dans les cas où cette odeur dépasse cette limite, les agents rencontrent des difficultés à converger. Malgré ce point, la communication entre agents traite de tous les cas et leur permet une bonne coordination.

Ce que nous avons apprécié dans ce projet, c'est sa liberté. En effet, le sujet est assez vaste et nous permet d'être créatifs et d'implémenter toutes nos idées. Après une prise en main un peu longue, il a été très stimulant de pouvoir réfléchir et mettre en œuvre toutes nos idées.