SCIENCES SORBONNE UNIVERSITÉ

Rapport Projet FOSYMA M1

Optimisation des Stratégies d'Exploration, de Coordination, de Communication et de Chasse des Golems dans un Système Multi-agents

Étudiants :
Robin SOARES
Paul-Tiberiu IORDACHE

Encadré par :
Cédric HERPSON
Aurélie BEYNIER
Vincent CORRUBLE

14 mai 2024

Table des matières

1 Introduction

Dans les systèmes multi-agents, la résolution de problèmes complexes implique la coordination et la collaboration entre plusieurs agents autonomes. Un des scénarios classiques de ce domaine est la chasse des golems, où un groupe d'agents doit traquer et capturer des entités mobiles appelées golems dans un environnement potentiellement complexe. Ce problème présente un défi de taille, car il exige la mise en œuvre de stratégies de recherche et de capture efficaces, tout en tenant compte des contraintes de communication, de perception et de mouvement propres à chaque agent. De plus, les golems peuvent adopter des comportements adaptatifs, rendant la tâche de chasse encore plus ardue. Dans le cadre de ce projet, nous explorerons les défis et les solutions liés à la chasse des golems dans un contexte multi-agents, en mettant en lumière les différentes approches algorithmiques et les techniques de coordination que nous avons utilisées pour résoudre au mieux ce problème complexe.

2 Exploration

Dans notre projet, la phase d'exploration implique la visite du maximum d'emplacements de notre environnement dans un temps donné, représentés dans notre cas spécifique par les noeuds d'un graphe. Par le biais de la communication et de l'exploration coopérative entre nos agents, nous cherchons à mettre à jour leur carte pour obtenir une connaissance complète de l'environnement.

2.1 FSM pour l'exploration

Tout d'abord, nous allons présenter la *Finite State Machine* (FSM) que nous avons choisi d'utiliser pour la stratégie de notre exploration coopérative.

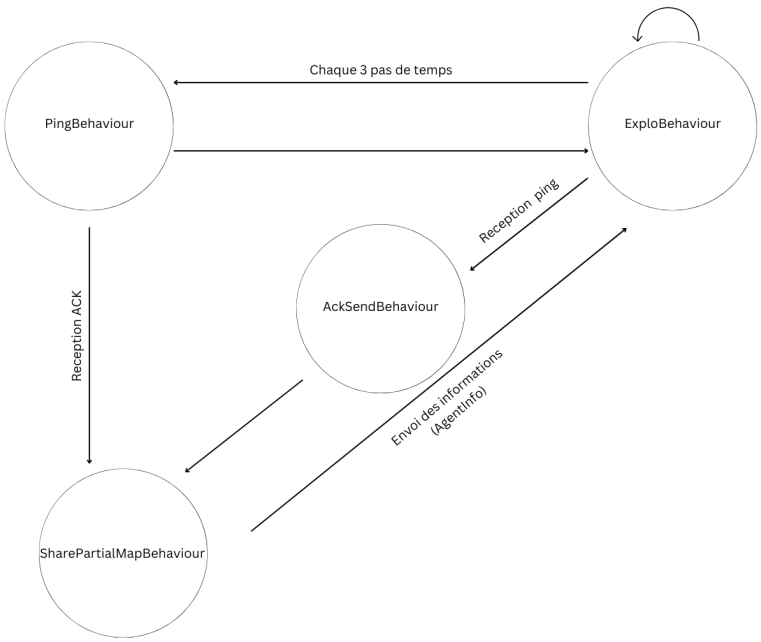

FIGURE 1 – FSM pour l'exploration

Notre stratégie d'exploration des cartes consiste à passer de *ExploBehaviour* au *PingBehaviour* tous les trois pas de temps, ce qui nous permet d'envoyer des pings aux autres agents et d'éviter l'envoi de notre propre carte à chaque pas de temps. En effet, cette approche pourrait surcharger le réseau avec des informations potentiellement non reçues. En utilisant les pings, nous limitons la charge réseau.

Lorsqu'un autre agent a reçu notre ping, nous attendons alors un *acknowledgement* (ACK) et un message avec les informations *AgentInfo*, contenant la position actuelle dans l'environnement de l'agent, sa prochaine position, ainsi que la carte de l'agent répondant à notre ping. Dès réception de ces informations, nous envoyons également nos propres *AgentInfo* à l'autre agent.

2.2 Optimisation de l'envoi de cartes

Nous avons décidé de ne pas transmettre systématiquement l'intégralité de la carte à chaque envoi, mais uniquement les noeuds visités depuis le dernier envoi. Cette approche permet de réduire la charge du réseau.

Pour mettre en oeuvre cette méthode, nous avons utilisé les fonctions *getNodesToShare*, *getSharedNodes*, et *addNodesToShare* du fichier ExploreFSMAgent.java, ainsi que la fonction *getPartialMap* du fichier MapRepresentation.java. Nous maintenons des listes distinctes pour chaque agent, contenant les noeuds déjà envoyés et ceux devant être transmis aux agents avec lesquels nous explorons l'environnement.

2.2.1 Complexité de l'envoi de cartes

En termes de **nombre et de taille des messages** : pour l'envoi de cartes, nous envoyons un ping contenant notre nom. Ensuite, le recepteur nous renvoie un ACK et utilise le *SharePartialMapBehaviour* pour envoyer son nom, sa position, sa prochaine position et les noeuds qu'il a visités et qu'il ne nous a toujours pas envoyés. Après réception de l'ACK, nous procédons de manière similaire à l'envoi de nos informations. Ainsi, 5 messages sont envoyés, les informations étant 2 chaînes de caractères et une liste de chaînes de caractères.

En mémoire, nous devons conserver, pour chaque agent, la liste des noeuds que nous lui avons déjà envoyés. Cela donne donc une complexité de O(m*n), avec m le nombre d'agents et n le nombre de noeuds à envoyer.

En termes de **nombre d'opérations**, c'est la méthode *addNodesToShare* qui nous permet de calculer les noeuds à envoyer efficacement. Le principe de cette fonction est le suivant : pour chaque noeud dans la liste "nodes" (liste contenant les noeuds explorés et reçus par un agent), elle vérifie s'il doit être ajouté aux noeuds à partager pour cet agent. Un noeud est ajouté aux noeuds à partager s'il n'est pas déjà dans la liste des noeuds à envoyer partager pour cet agent (*toshare_nodes*) et s'il n'a pas déjà été partagé (*shared_nodes*). Cela nous donne donc une complexité de O(m*n*l) avec m = nombre d'agents, n = nombre de noeuds de la liste "nodes" et l = nombre dans noeuds dans *toshare_nodes* et *shared_nodes*.

La méthode *getPartialMap* est également importante, car elle est utilisée pour l'envoi de la carte. Située dans MapRepresentation.java, elle génère une carte partielle à envoyer en fonction des noeuds à partager calculés dans le paragraphe précédent. Pour chaque

noeud à partager dans la liste *nodesToShare*, elle récupère le noeud correspondant dans la carte initiale, crée un nouveau noeud dans la carte partielle avec le même identifiant, ajoute cet identifiant à une liste de noeuds à envoyer, puis copie tous les attributs du noeud initial vers le nouveau noeud de la carte partielle. Ensuite, pour chaque noeud à partager, elle récupère le noeud correspondant dans la carte initiale. Pour chaque arête adjacente à ce noeud, elle récupère les identifiants des noeuds aux extrémités de l'arête. Si l'un de ces noeuds n'est pas déjà dans la liste des noeuds à envoyer, elle ajoute un nouveau noeud correspondant à la carte partielle, avec tous ses attributs copiés depuis le noeud initial. Enfin, elle ajoute l'arête entre ces noeuds correspondants dans la carte partielle.

En **nombre d'opérations**, sa complexité est O(n*m*(x+y)) avec n = nombre noeuds de *NodesToShare*, m = nombre d'arêtes, x = nombre d'attributs des noeuds et y = nombre d'attributs des arêtes.

2.3 Traitement des interblocages

Pour prévenir les interblocages potentiels, nous avons adopté la stratégie suivante : lors de la réception d'un message contenant les informations de l'agent répondant à notre ping, nous considérons la prochaine position que cet agent souhaite occuper pour poursuivre l'exploration. Si les deux agents se dirigent vers la même position, nous accordons la priorité à l'agent ayant l'identifiant le plus bas pour faire ce choix, tandis que l'autre devra recalculer son chemin le plus court vers un noeud non visité.

3 Chasse des Golems

Après la phase d'exploration, notre objectif principal est de localiser et de capturer les golems en se basant sur leur odeur émise jusqu'à une distance spécifiée. Notre stratégie de chasse vise à retrouver ces golems et à les bloquer en utilisant le moins d'agents possible, tout en optimisant l'efficacité de la capture.

3.1 FSM pour la chasse

Lors du passage de l'exploration à la chasse des golems, nous avons modifié notre FSM en y ajoutant de nouveaux états. Ainsi, une fois l'exploration terminée, nous passons dans le comportement de chasse (*ChaseBehaviour*). Pour ce faire, nous connectons l'état *ExploBehaviour* de la figure 1 à l'état *ChaseBehaviour* de la figure 2.

Nous passons de l'état *ChaseBehaviour* à l'état *ChaseAckBehaviour* chaque fois quand nous recevons un ping. Ensuite, une transition par défaut nous conduit à l'état *ShareInfosBehaviour*. Cet état nous permet d'envoyer les informations nécessaires lors de la chasse. Ces informations comprennent le nom de l'agent, sa prochaine position, la position du golem si elle a été détectée, nos observations (les noeuds dans notre champ de vision), informations utiles pour la coordination des agents lors de la chasse, ainsi qu'un booléen "block" qui nous permet de savoir si nous avons encerclé le golem. Ces informations sont référencées comme *ChaseInfos*. Dans l'état *ChaseBehaviour*, nous envoyons un ping à tous les agents uniquement si nous detectons de l'odeur.

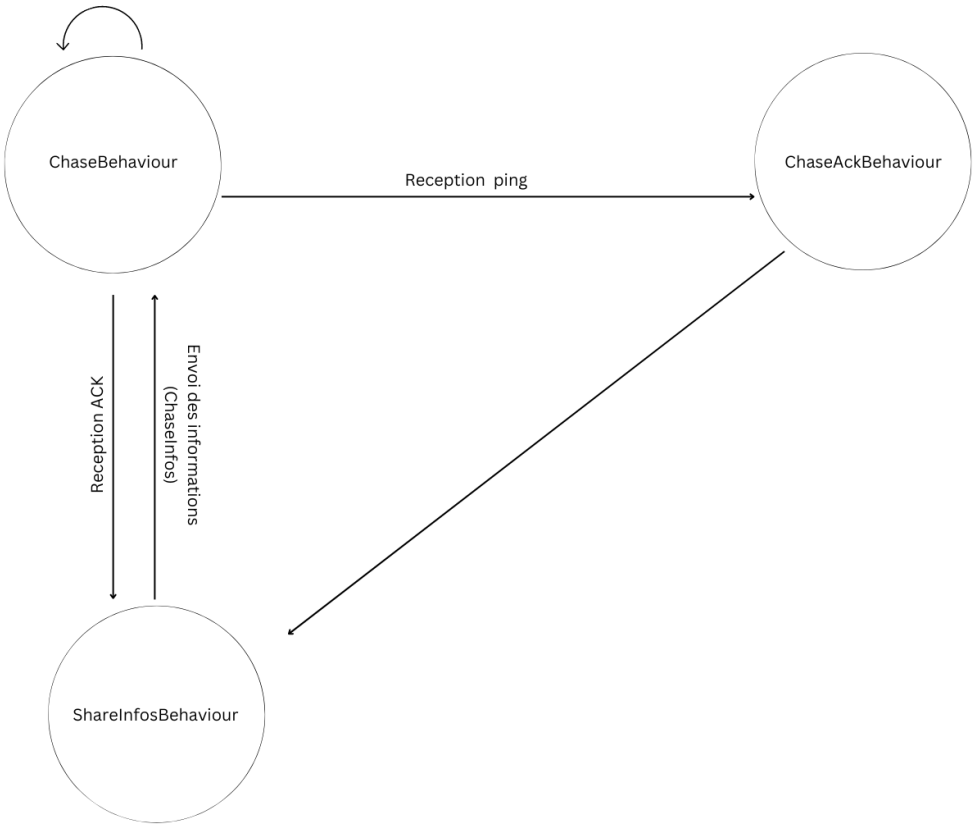

FIGURE 2 – FSM pour la chasse

3.2 Stratégie de chasse

Pour commencer, dans notre stratégie de chasse, nous avons opté pour une approche généralisée de notre code afin qu'elle puisse être applicable à la plupart des environnements.

Pour détecter le golem, nous avons décidé d'adopter une marche aléatoire tout en mémorisant notre dernier mouvement. Cette méthode vise à minimiser les retours au même endroit et à maximiser l'exploration de l'environnement.

3.2.1 Détection du golem

Pour notre stratégie de chasse (ChaseBehaviour.java), nous nous concentrons sur la détection du golem à l'aide de l'odeur qu'il émet. Ensuite, notre objectif est de rester dans la zone d'odeur en marchant de manière aléatoire dans la zone d'odeur du golem, tout en envoyant des pings à nos alliés pour venir. La détection de la position du golem se fait lorsqu'un agent tente de se déplacer vers un noeud contenant de l'odeur. S'il ne reçoit pas de message d'un agent sur ce noeud ou se deplaçant vers celui-ci, cela signifie qu'il s'agit du golem. En procédant ainsi, nous communiquons la position du golem ainsi que les informations *ChaseInfos* pour faciliter la chasse.

Dès que plusieurs agents chassent le même golem, en connaissant les positions ainsi que les prochaines positions de nos alliés en utilisant les informations *ChaseInfos*, nous

avons implémenté une méthode *getShortestPathWithoutPassing*, qui se retrouve dans MapRepresentation.java. Cette méthode nous permet de trouver le chemin le plus court vers un noeud sans passer par certains autres noeuds, dans notre cas spécifique les positions des alliés. Sa complexité en **nombre d'opérations** est le calcul d'un plus court chemin avec l'algorithme Djikstra, donc O(m + nlog(n)), avec m = nombre d'arêtes et n = nombre d'arêtes, sans passer par certain noeuds, que nous supprimons, avec leurs arêtes, du graphe. La suppression se fait en O(n*m) avec n = nombre de noeuds à supprimer et m = nombre d'arêtes correspondantes aux noeuds supprimés. Cette approche joue un rôle important pour ramener nos alliés autour d'un golem et donc de le bloquer.

Nous tenons à préciser que pour l'instant, nous bloquons le golem temporairement ; le véritable blocage sera mis en place dans la partie suivante.

3.2.2 Complexité de la stratégie de chasse

En termes de **nombre et de taille des messages**, nous adoptons encore un protocole de type ping-ACK. Nous envoyons tout d'abord un ping, suivi de nos informations *ChaseInfos*, si nous recevons un ACK, à chaque mouvement lorsque nous détectons de l'odeur dans l'environnement.

En **mémoire**, nous conservons évidemment les *ChaseInfos*, dont nous connaissons le contenu grâce aux paragraphes précédents, ainsi que le plus court chemin vers le noeud où nous voulons nous déplacer. Par conséquent, la complexité de la mémoire est de O(m*n), avec n étant le nombre de noeuds du plus court chemin et m le nombre d'agents.

Pour le déplacement des agents, en **nombre d'opérations** c'est seulement un calcul d'un plus court chemin sans passer par certain noeuds, dont la complexité est donnée dans la section 3.2.1.

4 Blocage du golem

4.1 FSM pour le blocage du golem

Tout d'abord, nous allons présenter nos modifications sur le FSM dans la figure 2.

Dès que nos agents parviennent à bloquer temporairement le golem dans *ChaseBehaviour*, ils essayent de le pousser dans un coin. En utilisant des pings, ils vérifient leur état et leur position ainsi que celles de leurs alliés, puis passent à l'état *ToCornerBehaviour*.

Dans cet état, nos agents s'efforcent de bloquer efficacement le golem en le poussant dans un coin. Une fois cette action accomplie, les agents qui bloquent le golem passeront dans l'état *BlockBehaviour* et enverront des messages à leurs alliés pour leur indiquer qu'ils peuvent partir chasser un autre golem. De plus, si un agent se trouve perdu lors de l'état *ToCornerBehaviour*, il passera également en mode chasse et donc dans *ChaseBehaviour*.

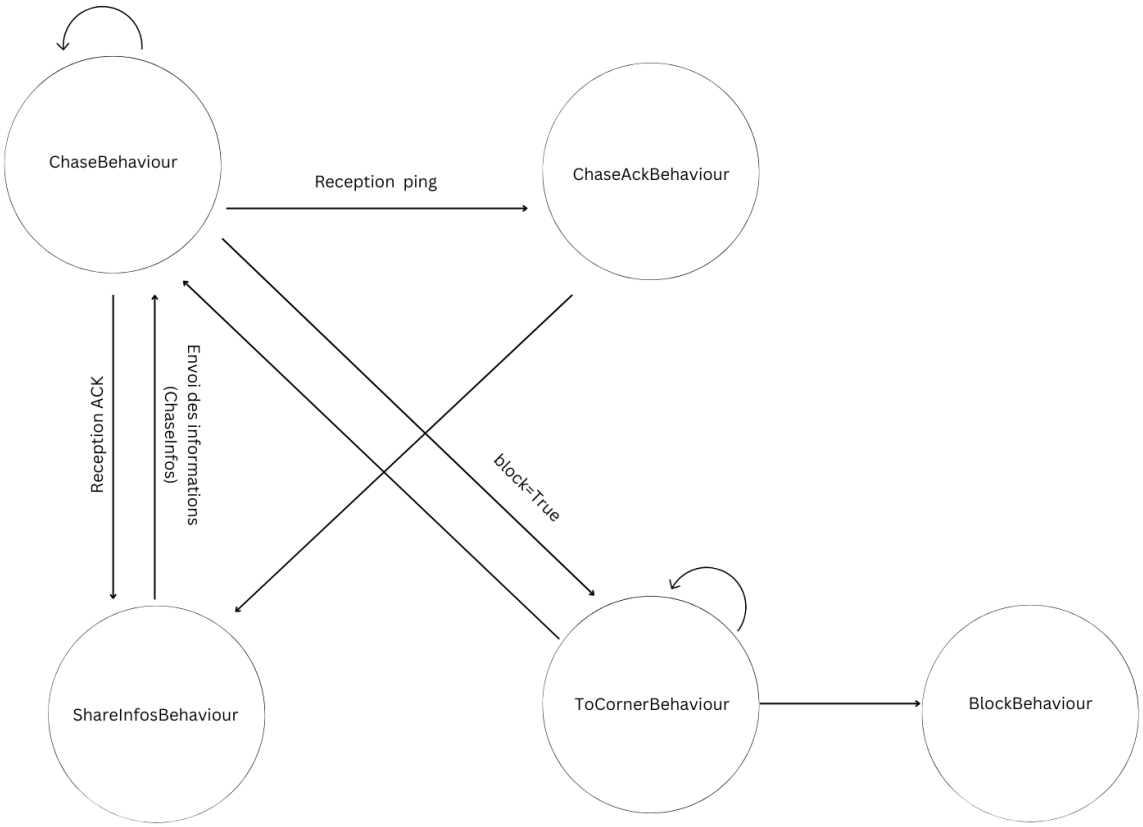

FIGURE 3 – FSM pour la chasse avec blocage

4.2 Blocage efficace du Golem

Tout le long du voyage pour amener le golem dans un coin du graphe, nos agents vont vérifier si tous les agents autour du golem sont dans l'état *ToCornerBehaviour* et s'ils sont assez pour couvrir tous les noeuds pour ne pas que le golem s'enfuit. La méthode nous permettant de réaliser cela est la méthode *check* du fichier *ToCornerBehaviour*. Si tout va bien après ce test, nous passerons à la suite de notre traitement.

Pour bloquer efficacement le golem, nous avons décidé d'implémenter une fonction appelée *getNodesWithSmallestArity*, qui, selon les informations de notre carte, renvoie une liste de noeuds ayant la plus petite arité. Cette fonction est située dans MapRepresentation.java et sa complexité en **nombre d'opérations** est O(n), avec n = le nombre de noeuds du graphe.

Ensuite, l'agent avec l'identifiant le plus petit parmis les agents qui emmenent le golem dans le coin est désigné en tant que leader. C'est celui-ci qui va communiquer son plus court chemin en partant de la position du golem vers les noeuds de plus petite arité à ses alliés pour qu'ils puissent tous avoir en mémoire le même chemin. La méthode nous permettant de calculer cela se trouve dans le fichier MapRepresentation.java et elle s'appelle *getShortestPath*. Sa complexité **en nombre d'opérations** est celle d'un algorithme de Djikstra, donné dans la section 3.2.1.

Avec le chemin établi, nos agents vont pousser le golem en l'entourant à chaque fois, ne lui laissant qu'une seule possibilité de mouvement : le prochain noeud du plus court

chemin vers le noeud de plus petite arité choisi.

Pour pousser le golem dans le coin, nous devons communiquer pour nous assurer que nous le bloquerons sur le prochain noeud du plus court chemin. Pour ce faire, le leader, en recevant les positions de ses alliés, calcule leur prochaines positions afin de pouvoir entourer le golem au prochain mouvement. L'agent se trouvant sur le prochain noeud du plus court chemin devra changer sa position pour laisser le golem avancer. Ensuite, l'agent qui doit se rendre sur la position du golem tentera de s'y déplacer jusqu'à ce qu'il y parvienne. Lorsqu'il y arrive, il notifie aux autres agents qu'ils doivent se dépêcher d'aller aux noeuds bloquants choisis au préalable. Ensuite, ils répètent ceci jusqu'à ce qu'ils atteignent le coin. De plus, si un agent ne se retrouve pas dans l'équipe formée pour ramener le golem dans un coin, il devra se déplacer afin de donner la priorité aux agents qui sont autour du golem. Ainsi, les agents près du golem envoient un message aux autres pour leur demander de changer leurs positions. Nos agents font aussi attention à ne pas rentrer dans une voie sans issue et laisse le golem s'y avancer pour être sur de le bloquer avec un seul agent.

Dès que le golem est bloqué dans un coin, les agents qui le bloquent passeront dans le comportement *BlockBehaviour*. Leur rôle dans ce comportement est d'envoyer des messages aux autres agents environnants pour les informer de partir chasser un autre golem potentiel.

Après avoir reçu le message de partir chasser un autre golem, nous explorons l'environnement autour de notre golem si il déploie une odeur supérieure à deux de distance, en recherchant toute autre odeur qui pourrait indiquer la présence d'un autre golem potentiel à cet endroit pendant une période de temps. Si, pendant cette période, nous ne détectons pas d'autre golem, nous calculons ensuite le chemin le plus court vers un noeud situé à mi-chemin entre notre position actuelle et le noeud le plus éloigné. Enfin, nous procédons à notre chasse comme précédemment.

4.2.1 Complexité du blocage dans un coin

En ce qui concerne **le nombre et la taille des messages**, le leader envoie son plus court chemin vers le coin, ce qui est représenté par une liste de chaînes de caractères. De plus, il envoie les prochaines positions aux alliés, ce qui signifie 2n messages avec n étant le nombre d'agents. Les alliés envoient également leurs noeuds atteignables, ce qui équivaut à n messages avec n représentant le nombre d'agents. Ensuite, lorsque l'agent sur le chemin se déplace pour permettre au golem de bouger, il notifie chaque agent du mouvement du golem, générant ainsi encore n messages, avec le contenu de chaque message vide. Nous ajoutons aussi les messages envoyés pour vérifier si tous les agents bloquant le golem sont passés dans l'état *ToCornerBehaviour*. Chaque agent envoie n messages à ses alliés, ce qui donne un total de n*m messages, avec m étant le nombre d'alliés. De plus, lors du blocage du golem dans un coin, encore n*m messages sont envoyés, avec n étant le nombre d'agents bloquant le golem et m le nombre d'agents recevant le message. Ces messages sont envoyés pour communiquer l'intention de partir chasser un autre golem potentiel. En plus, nous considérons également les messages envoyés par ceux se trouvant dans *ToCornerBehaviour* aux agents dans *ChaseBehaviour* pour leur dire de se déplacer de leur chemin vers un coin. Pour ce faire, chaque agent dans *ToCornerBehaviour* envoie un message contenant leur position dans l'environnement, sous forme de chaîne de caractères. Finalement, encore n*m messages sont envoyés à chaque pas de temps, n étant le nombre d'agents se trouvant dans

ToCornerBehaviour et m étant le nombre d'agents dans l'environnement.

En **mémoire**, chaque agent conserve sa prochaine position ainsi que le plus court chemin envoyé par le leader, qui est une liste de noeuds, tandis que le leader conserve également le plus court chemin vers le coin, ainsi que les positions de ses alliés. Par conséquent, la complexité de l'espace mémoire est de O(n+m), où n = le nombre d'alliés et m = la taille du chemin pour le leader et O(m) pour les agents n'étant pas le leader.

En termes **de nombre d'opérations**, le calcul effectué par le leader pour donner aux autres agents leurs prochaines positions lors du déplacement du golem est le plus coûteux. La complexité de cette opération est cubique, en excluant les autres opérations pour les manipulations des chaînes de caractères telles que "compareTo".

5 Resultats et discussions

5.1 Points forts de la stratégie d'exploration et améliorations possibles

Par rapport à une stratégie naïve qui consiste à envoyer la carte entière sans même savoir si les agents la réceptionne, nous avons décidé de faire un partage de cartes moins coûteux pour la plateforme (cf section 2.2), même si nous envoyons plusieurs messages (multiples Ping - ACK) à faible coût, la charge réseau sera grandement diminuée. De plus, nous n'envoyons également qu'une carte partielle qui contribue également à la diminution de la charge réseau.

Notre exploration nous permet de ne pas avoir d'interblocages et de ne pas rester autour d'un point non atteignable (dans le cas par exemple d'un golem qui ne bouge pas) sans avoir déjà tout exploré.

Une manière intéressante pour améliorer l'exploration serait d'isoler pour chaque agent une partie de l'environnement, et lors de la finalisation de l'exploration de chaque sous-partie, de se rencontrer et de partager les informations. Comme nous ne connaissons pas a priori l'environnement dans lequel nous nous trouvons (arbre, graphe, etc.) ni le rayon de communication, nous avons décidé que l'approche choisie serait la plus adaptée pour une exploration générale de l'environnement et la plus adaptée à notre temps alloué pour développer ce projet.

5.2 Points forts de la stratégie de chasse et améliorations possibles

Lors de notre stratégie de chasse, comme nous l'avons déjà mentionné, nous avons cherché à implémenter un code aussi général que possible afin qu'il puisse converger et bloquer des golems dans la plupart des environnements. Par exemple, dans le cas d'un environnement constitué d'un arbre, des stratégies de chasse beaucoup plus spécifiques seraient bien plus performantes que notre implémentation. Cependant, nous avons fait le choix de généraliser notre stratégie pour qu'elle puisse être applicable à une variété d'environnements, même si elle peut ne pas être la plus performante dans des situations spécifiques. Dans le cas d'un environnement différent d'un arbre, notre stratégie sera plus performante que la stratégie typique adaptée à un arbre.

Pendant notre chasse, nous avons décidé d'effectuer une marche aléatoire dans le graphe et dans l'odeur. Cependant, nous optimisons cette marche en mémorisant le dernier mouvement effectué par chaque agent. Ainsi, nous évitons de retourner au même endroit lors du prochain mouvement, ce qui permet de maximiser l'exploration de l'environnement. Une amélioration possible consisterait à prendre en compte les noeuds visités il y a plus longtemps et à les visiter en priorité. Cela permettrait également d'explorer l'environnement de manière efficace en évitant de revisiter des zones déjà explorées récemment. Nous sommes conscients que d'autres stratégies de chasse en groupe comme un balayement de graphe auraient pu etre implémentées et auraient été beaucoup plus efficaces que de la chasse aléatoire.

La manière dont nous communiquons pendant la chasse nous permet d'assurer un déplacement fluide des agents et donc d'éviter au maximum les situations d'interblocage. Le fait de ne pas avoir d'interblocage permet donc à nos agents de ramener le golem dans un coin. Cependant, le manque de réactivité des agents bloquant un golem peut, dans certains cas spécifiques, embarrasser les autres agents. Dans ce cas, nous avons décidé de faire partir les agents adjacents vers un endroit plus éloigné de la carte, mais il existe sûrement d'autres solutions intéressantes, comme par exemple leur faire visiter un noeud que nous n'avons pas exploré depuis longtemps.

Le fait de devoir bloquer à chaque fois le golem sur un noeud de plus petite arité sera parfois problématique, surtout dans le cas où nous disposons d'un grand nombre d'agents pour bloquer le golem et donc de pouvoir le bloquer dans un noeud avec une plus grande arité. Cependant, avec notre implémentation, nous cherchons à bloquer les golems avec le moins d'agents possible puisque nous ne connaissons pas le nombre de golems existants dans l'environnement, même si cela peut prolonger la chasse.

Nos agents, lorsqu'ils sont dans l'état *BlockBehaviour* ne sont plus réactifs et cela peut poser problèmes notament s'ils sont entrés dans cet état par soucis d'asynchronisme ou s'ils bloquent une issue importante dans le graphe. Une solution aurait été de pouvoir les ramenés vers l'état *ChaseBehaviour*.

Nos résultats de tests semblent très satisfaisants sur des golems se déplaçant aléatoirement dans le graphe.

6 Conclusion

La chasse des golems dans un système multi-agents soulève de nombreux défis techniques et conceptuels. Nous avons exploré diverses approches algorithmiques et stratégies de coordination pour résoudre au mieux ce problème complexe. Nos résultats semblent concluant, aussi bien au niveau du temps d'execution qu'au niveau de la complexité en mémoire.

Cependant, notre stratégie reste améliorable, avec notamment une potentielle prise en compte de l'incertitude et une anticipation des comportements possibles des golems. Certains cas non traités pourraient avoir un impact négatif sur notre exploration ou notre chasse. Pour le futur, notre stratégie de chasse serait à modifier pour accélérer la detection de golems dans l'environnement.