FoSyMa

Projet - Dedale

Natacha Rivière - 28706745
Léa Movsessian - 28624266

Année : 2023/2024

SCIENCES SORBONNE UNIVERSITÉ

Table des matières

1 Introduction

1.1 Sujet

Le projet consiste en la mise en oeuvre d'un système multi-agents hétérogènes spécialisé dans la chasse de golems. Il est réalisé en `JAVA` sur `JADE` (Java Agent DEvelopment Framework), une plate-forme open-source pour les systèmes basés sur du multi-agents.

Les agents de notre système sont chargés de plusieurs tâches : ils doivent d'abord cartographier leur environnement, en repérant des golems grâce à des « odeurs »qu'ils émettent. Ensuite, ils doivent collaborer de manière intelligente pour encercler et immobiliser efficacement les golems présents sur la carte, les « capturant ».

Nos agents sont soumis à plusieurs contraintes : chacun a une vitesse et une portée de communication propre et ne connait pas les valeurs des autres. De plus, ils ne connaissent pas le nombre de golems présents sur la carte.

Le but est donc de développer des stratégies d'exploration et de chasse adéquates qui optimisent à la fois les communications (le nombre de bits envoyés et reçus), la mémoire et le temps.

Aucune stratégie de chasse entièrement fiable n'ayant encore été découverte avec nos contraintes, nous essaierons donc d'illustrer au mieux en quoi nos solutions sont pertinentes bien qu'imparfaites.

1.2 Structure Globale

1.2.1 Agents

Les Agents sont tous de la classe `ExploreCoopAgentFSM` qui se base sur `ExploreCoopAgent`. Nous y avons ajouté plusieurs fonctions de classe que nous développerons si besoin par la suite.

1.2.2 Behaviours

La structure de nos behaviours est une FSM (Finite State Machine). Chaque comportement est donc représenté par un état, et les transitions entre ces états sont déclenchées par des conditions spécifiques que nous détaillerons par la suite. Cela nous permet de modéliser de manière claire et organisée le fonctionnement de notre système.

Voici comment est structuré notre système :

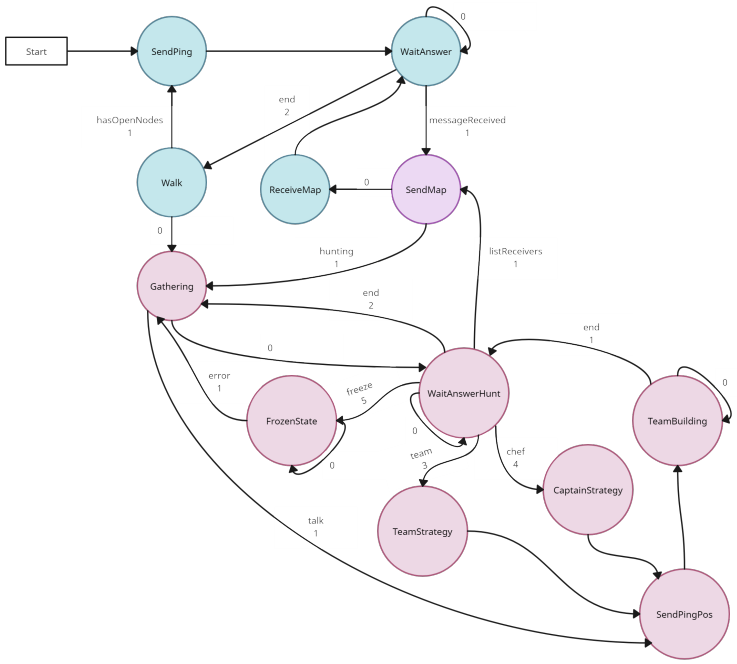

Figure 1 – Structure des behaviours

Les états sont divisés en deux catégories :

- Exploration, en bleu.
- Chasse, en rouge.
- L'exception est `SendMap`, en violet, faisant partie de l'exploration mais étant accessible depuis la chasse pour permettre à des agents en chasse de partager leurs cartes à des agents encore dans la phase d'exploration.

1.2.3 Représentation de la carte

Nous avons modifié la représentation de la carte des agents, que ce soit dans la mémoire ou visuellement.

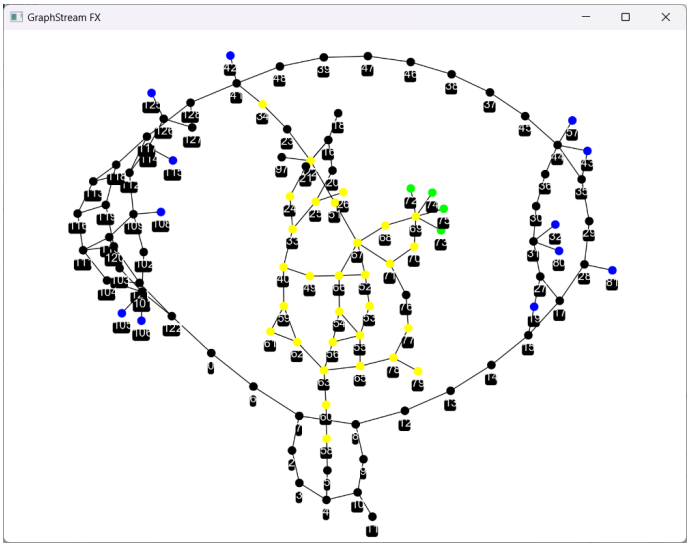

FIGURE 2 – Représentation de la carte d'un agent quelconque

En effet, comme on peut le voir sur la carte ci-dessus, un noeud peut avoir 4 états différents, chacun représenté par une couleur :

- Ouvert, en bleu
- Fermé, en noir
- Ouvert avec odeur, en vert
- Fermé avec odeur, en jaune

Cela permet à l'agent de se préparer à l'étape chasse et à l'utilisateur de mieux visualiser la mémoire.

On définit les noeuds à odeur grâce à un compteur de `stench` et une date de mise à jour tels que, sur le noeud concerné :

- S'il y a une odeur on incrémente son compteur de `stench` de 1.
- S'il n'y a pas d'odeur, on change l'état du noeud et on met le compteur à 0.
- Dans tous les cas, on modifie la date à la date actuelle.

2 Exploration coopérative de l'environnement

2.1 Principe

Nous avons implémenté l'exploration coopérative de l'environnement en suivant les bases données par les Agents Explorateurs fournis au début du projet.

Le concept est en effet le même : Les agents explorent la carte, en allant vers le noeud ouvert le plus proche jusqu'à ce qu'il n'y ait plus de noeud ouvert sur leur représentation de l'environnement.

Pendant qu'ils explorent, ils peuvent croiser d'autres agents et vont se partager des informations pour gagner en efficacité.

2.2 Structure

Ici, nous nous intéresserons uniquement à la partie Exploration, nous allons donc simplifier notre FSM pour développer cette partie.

La partie Exploration est divisée en 5 behaviours :

- `Walk` est le comportement de déplacement de l'agent.
- `SendPing, WaitAnswer, SendMap` et `ReceiveMap` font partie de la communication entre les agents.

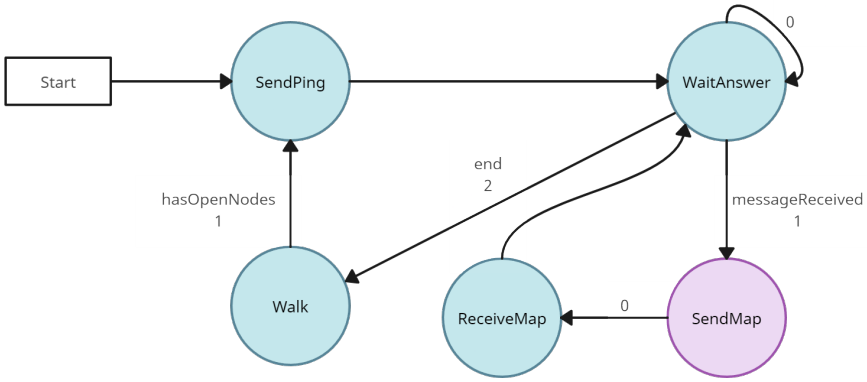

FIGURE 3 – Partie exploration de la FSM

2.3 Déplacement

2.3.1 Algorithme

Le déplacement des agents, défini dans `Walk`, utilise, tout comme `ExploreAgentCoop`, l'algorithme Dijkstra pour trouver le noeud ouvert le plus proche ainsi que le chemin y menant. Les agents vont donc en direction de ce noeud.

L'itinéraire n'est calculé qu'une fois et est conservée en mémoire dans la `MapRepresentation` de l'agent. Cela permet d'éviter de calculer à chaque itération un nouvel itinéraire.

A partir de cela, il y a plusieurs possibilités :

- Il arrive à destination et continue l'exploration jusqu'à ce qu'elle soit terminée.
- Un agent communique avec lui et lui donne les informations concernant sa destination, il ferme le noeud et continue l'exploration.
- Il se retrouve bloqué durant son itinéraire par un golem ou par un agent. Dans ce cas, l'agent retire de sa représentation de la carte l'arête qui relie sa position actuelle au noeud bloqué pour un nombre d'itérations défini. Il recalcule un chemin vers le noeud ouvert le plus proche qui peut toujours être le même noeud si un autre chemin existe. Si le noeud bloqué est le dernier noeud ouvert, on sélectionne un noeud aléatoire et on y va.

Ce cassage d'arêtes permet d'éviter de se retrouver bloqué indéfiniment dans certains cas. Par exemple, si deux agents sont censés passer par le noeud de l'autre, ils seraient restés bloqués pour une durée infinie.

De plus, même si l'arête est rajoutée au bout de i itérations et que son précédent objectif est de nouveau le chemin le plus court, l'agent ira tout de même jusqu'à sa nouvelle destination. Cela évite de potentielles boucles infinies où, dans le même esprit que l'exemple précédent, deux agents pourraient faire des allers-retours dans le même couloir et ne jamais pouvoir s'en échapper.

De même, la sélection aléatoire d'un noeud lorsque le noeud bloqué est le dernier noeud ouvert permet d'éviter de rester bloqué à cause d'un golem dans un couloir. En effet, si l'on ne met pas cette condition et qu'un golem va sur un noeud d'arité 1 et que ce dernier s'avère être le dernier ouvert, les agents vont tenter de se rendre sur le noeud, cassr l'arête et ne pas bouger.

2.4 Communication

Les agents explorateurs fournis au début du projet envoyaient leurs cartes à chaque pas qu'ils faisaient. Cela envoyait beaucoup de données à chaque itération. Nous avons donc décidé de changer cela pour minimiser le nombre de données envoyées.

2.4.1 Algorithme

Pour cela, au lieu d'envoyer la carte à chaque pas, nos agents procèdent tel que illustré sur la figure 4, à droite :

- L'agent envoie un ping à tous les agents sauf ceux avec qui il a communiqué récemment pour vérifier s'il y a des agents à proximité.
- Il attend sans bouger jusqu'à une date déterminée ou jusqu'à ce qu'il reçoive une réponse.
- S'il reçoit une réponse :
  - Si le message est un `Yes`, l'agent passe à la behaviour `SendMap`.
  - Si le message est un `Ping`, l'agent envoie `Yes` et passe à la behaviour `SendMap`
- Une fois dans `SendMap`, il crée sa liste de destinataire avec tous les agents lui ayant envoyé un `Ping` ou un `Yes`.
- Pour chaque agent, il vérifie en mémoire la dernière version envoyée par ledit agent, si elle existe.
- Il envoie la différence entre sa carte et la dernière carte envoyée par l'agent concerné pour chaque agent de sa liste de destinataire.
- Lorsque toutes les cartes ont été envoyées, il reçoit les cartes dans la behaviour `ReceiveMap` et les stocke en mémoire en l'associant à l'agent concerné.
- Il fusionne sa carte avec celles reçues en prêtant une attention particulière aux noeuds à odeur. Pour ces derniers, on garde l'information du noeud ayant la date de mise à jour la plus récente entre le noeud de notre carte actuelle et celui de la carte reçue.
- Une fois que toutes les cartes attendues ont été traitées ou que le timeout de `ReceiveMap` est fini (pour éviter d'attendre indéfiniment une carte d'un agent qui est malheureusement parti), il retourne dans `WaitAnswer` jusqu'à la fin du délai, au cas où il aurait raté un `Ping` ou un `Yes`.
- A la fin du délai, il fait un pas. Il recommence jusqu'à ce que sa carte soit complète.

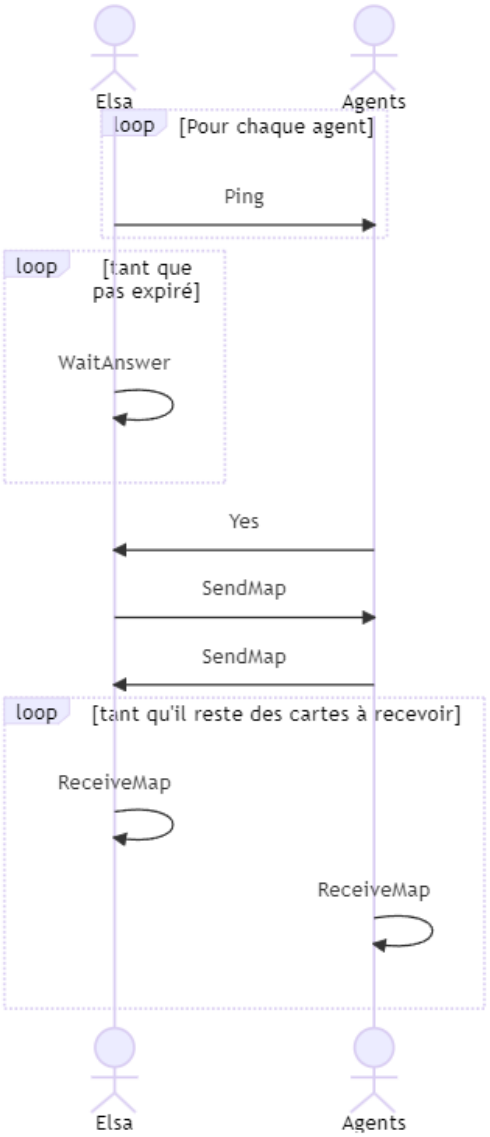

FIGURE 4 – Schéma de l'algorithme de communication durant l'exploration

2.4.2 Complexité

On considère ici la complexité dans le pire des cas.

Communication

Messages

On pose $n+1$ le nombre d'agents. On considère ici une communication entre un agent A et n agents :

- L'agent A envoie un `Ping` en broadcast, c'est-à-dire n agents. Il envoie donc n messages.
- Les n agents répondent un message `Yes`, ce qui fait également n messages.
- Dans le pire des cas, les agents n'ont jamais discuté entre eux. L'agent A envoie donc une carte complète à chaque agent. On a n messages de la taille de la carte de l'agent A.
- Chaque agent fait de même, ce qui fait de nouveau n messages de la taille de la carte de chaque agent.

Au total, on a donc une complexité en nombre de messages envoyés en $O(4n)$, soit $O(n)$.

Données

Pour le nombre de données envoyées, on considère l'envoie d'un `Ping` et d'un `Yes` comme négligeable par rapport à l'envoi d'un message de la taille de la carte.

On pose m la taille de la carte, qui comprend les noeuds, l'état de chaque noeud, le compteur de stench, la date de mise à jour et les arêtes.

Dans le pire des cas, comme indiqué plus haut, les agents n'ont jamais discuté entre eux et connaissent tous les noeuds et toutes les arêtes. [1] L'agent A envoie donc n messages de taille m et les autres agents font de même, envoyant chacun un message de taille m, soit n messages de taille m.

On a donc une complexité en terme de données envoyées de $O(2 \times n \times m)$, soit en $O(n \times m)$.

Temporelle

On suppose que, dans le pire des cas, le temps d'envoi de chaque message ne se superpose pas. On pose e le temps mis pour l'envoi d'un message. Comme indiqué plus haut, on a $4 \times n$ messages envoyés. On a donc une complexité temporelle (uniquement pour l'envoi des messages) de $O(4 \times n \times e)$, soit une complexité de $O(n \times e)$.
Pour le reste de l'algorithme :

- Il attend au moins une réponse avant de passer dans `SendMap`. On ajoute donc un temps e.
- Il boucle sur le nombre de messages reçu pour créé sa liste de destinataire, soit n.
- Il calcule la différence entre sa carte et la carte des autres. Ici, dans le pire des cas, on a déjà communiqué au moins une fois avec tous les agents. On appelle ici la méthode `getDiff` de la classe `MapRepresentation`. On a deux boucles :
  - Une sur les noeuds uniquement
  - Une sur les noeuds avec une boucle imbriquée sur les arêtes de chaque noeud.

  En posant p le nombre de noeuds et q le nombre d'arêtes de la carte, la méthode a une complexité temporelle en $O(p \times (q+1))$, soit en $O(p \times q)$. [2]
- Une fois toutes les réponses reçues, il boucle sur ces dernières (dans `ReceiveMap`) pour les fusionner avec la sienne. On a donc une boucle n qui appelle la méthode `mergeMap` de la classe `ExploreCoopAgentFSM` :
  - Elle appelle la méthode `mergeMap` de la classe `MapRepresentation`. Cette dernière a la même complexité temporelle que `getDiff`, ayant les mêmes boucles `for` que celle-ci. Soit en $O(p \times q)$
  - Elle boucle sur le nombre d'arêtes précédemment retirée suite au cassage d'arêtes. Cependant, cette boucle est négligeable.
- Finalement, il retourne dans `WaitAnswer`.

Au total, on a donc une complexité temporelle en $O(4 \times n \times e + n + e + n + p \times q \times 2)$, soit en $O(n \times e + p \times q)$.

1. Il leur reste au moins un noeud ouvert
2. On considère ici que, dans le pire des cas, chaque noeud est lié à chaque arête de la carte.

Mémoire

On considère que la complexité en mémoire concerne la mémoire des agents pendant l'exécution de l'algorithme. Chaque agent possède en mémoire pour l'exploration :

- La liste des noms des autres agents, soit n fois la taille en mémoire d'une `String`.
- Sa propre carte soit, dans le pire des cas, la carte entière.
- La dernière carte de chaque agent soit, dans le pire cas, n fois la carte entière.
- La liste des agents proches de lui, soit dans le pire des cas n fois la taille en mémoire d'une `String`.
- La liste des maps reçues, soit n fois des cartes entières.
- Quelques variables de type `int` négligeables

On pose s la taille en mémoire d'une String et r la taille en mémoire de la carte. Oa donc une complexité en mémoire pour un agent en $O(2 \times n \times s + r + 2 \times n \times r)$ soit en $O(n \times s + n \times r)$. On peut considérer que la taille d'une `String` en mémoire est négligeable par rapport à la taille de la carte. On a donc une complexité en mémoire finale en $O(n \times r)$.

En considérant tous les agents, on aura donc une complexité en $O(n^2 \times r)$.

2.5 Limites

Le fonctionnement de l'exploration a plusieurs limites plus ou moins importantes. Certaines, malheureusement, empêchent le fonctionnement de nos stratégies dans certains cas.

Problèmes de fonctionnement

Si, sur la carte, il y a un golem qui ne bouge pas, alors nos agents ne pourront jamais finir l'exploration de la carte. Plusieurs solutions ont été envisagées, dont le fait de passer directement au blocage de ce golem. Le problème ici est que le golem peut aussi avoir une vitesse de déplacement extrêmement lente et être sur un noeud qui cache une partie de la carte. Nous n'avons pas trouvé de solution satisfaisante ou implémentable dans les temps et ne pouvons donc pas capturer un golem immobile.

Problèmes d'optimisation

Notre exploration contient plusieurs problèmes d'optimisation, certains avec un effet négligeable, d'autres plus impactants :

- Il n'y a pas d'optimisation de déplacement. C'est-à-dire que nos agents peuvent se suivre pendant une grande partie de cette étape, ce qui ralentit l'exploration de la carte.
- Pendant le partage de cartes, ils peuvent envoyer plusieurs fois les mêmes informations de cartes parce qu'on considère que leurs messages peuvent ne pas arriver. Ceci est un choix de notre part, cependant cela reste à noter.
- Pendant la communication, les `Ping` et `Yes` peuvent être doublés. En effet, un agent A peut envoyer un `Ping` au même moment qu'un agent B. Ils vont donc tous les deux se renvoyer un `Yes`. Cependant ils ne vont pas s'envoyer deux fois la même carte. Ce problème est donc assez limité et négligeable.

3 Capture des Golems

3.1 Principe

La stratégie que nous avons choisie pour la chasse des golems se base sur les stratégies de chasses mises en place dans nos campagnes. Nos agents repèrent des traces de golems puis organisent une battue afin de rabattre le « gibier »vers des points stratégiques.
Pour ce faire, les agents se rassemblent et forment des équipes, dirigées par un chef de ligne, qui indique aux agents les noeuds sur lesquels ils doivent se placer pour forcer, ou inciter si l'équipe est trop réduite, le golem à aller vers son trépas.

FIGURE 5 – Illustration de la stratégie de chasse

3.2 Structure

La partie Chasse est divisée en 7 behaviours :

- **Gathering** et **TeamBuilding** qui gèrent la création d'équipes.
- **WaitAnswerHunt, SendPingPos** qui gèrent une grande partie des communications de la chasse.
- **CaptainStrategy, TeamStrategy, Frozen** qui gèrent la coordination proprement dite de la stratégie de chasse.

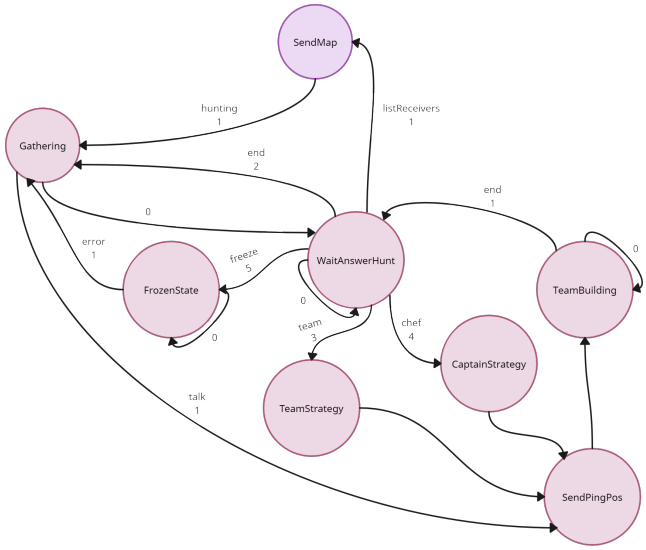

FIGURE 6 – Partie chasse de la FSM

Le bahaviour **SendMap**, qui sert à l'exploration, est accessible depuis la chasse car si un agent en chasse reçoit un Ping d'un agent en exploration, il lui enverra sa carte avant de poursuivre.

3.3 Diagnostic

3.3.1 Algorithme

Lorsqu'un agent se déplace sur la carte et qu'un déplacement lui est refusé, il veut établir s'il se trouve face à un golem ou face à un agent. Pour cela, il utilise un diagnostic. Il y a 3 cas possibles :

- Le noeud en face ne porte pas de stench. Ce n'est pas un golem, l'agent continue son exécution sans changement.
- Il sait qu'un agent est sur le noeud en face de lui, car il a reçu un ping de cet agent.
- Il n'y a pas d'agent et le noeud a une odeur, il envoie donc un message de diagnostic et attend une réponse.

Si un agent reçoit le diagnostic et est sur le noeud demandé, il répond avec un message `confirm` contenant l'`id` du noeud. Si un golem est sur le noeud, personne ne répond au diagnostic envoyé. Après un temps d'attente, l'agent signale en broadcast qu'il a repéré un golem. Il envoie la position du golem ainsi que la date à laquelle il l'a trouvé. Ainsi, les autres agents peuvent garder la position la plus récente d'un golem.

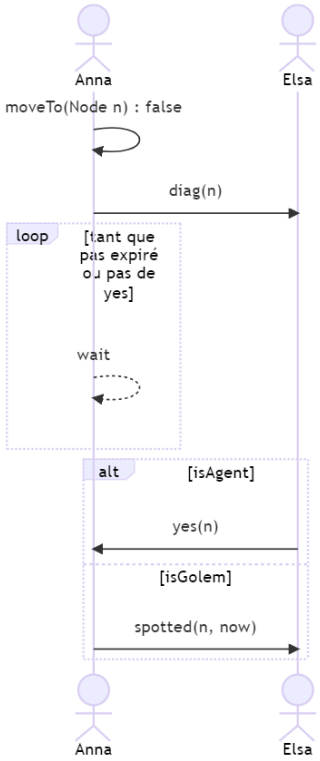

FIGURE 7 – Illustration du diagnostic

3.4 Formation d'équipes

3.4.1 Algorithmes

Pour former des équipes, les agents vont essayer de se rassembler, pour cela, ils choisissent le noeud ayant la plus grande odeur de leur carte et s'y dirigent. Une fois assez proche de ce noeud, ils envoient à tous les autres agents un message contenant leur position actuelle, leur destination et l'étape de chasse à laquelle ils sont, en l'occurrence zéro.
Si un agent est à portée, les communications de la figure 8 se mettent en place.

- L'agent Olaf reçoit un ping d'un agent qui n'est pas dans sa team, il envoie une offre pour rejoindre son équipe.
- Si Anna reçoit une offre, elle ajoute Olaf à son équipe et informe toute l'équipe de la mise à jour. Elle transmet la liste des noms de tous les agents de son équipe.
- A la réception d'une update, chaque agent enregistre les modifications.
- Les agents se partagent les mises à jour.

Pour s'assurer que tous les agents d'une équipe reçoivent les mises à jour, même s'ils ne sont pas à portée de communication de Anna, quand les agents enregistrent une modification, tous les agents envoient en broadcast à leurs coéquipiers la liste des agents de l'équipe.
Si un agent reçoit une update qui ne contient aucune information nouvelle, il n'envoie aucun message.

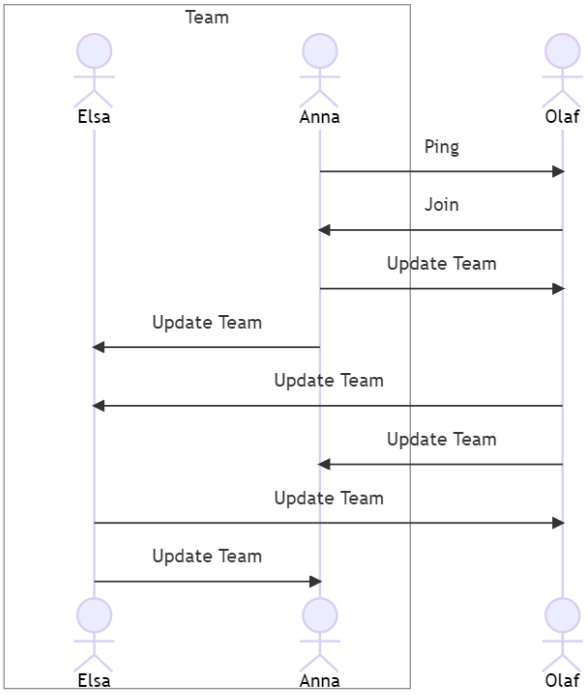

FIGURE 8 – Schéma des communications de la formation d'équipes

3.5 Chasse d'un golem

3.5.1 Algorithme

Lors de la chasse, les agents d'une équipe sont de deux types, le chef et les subordonnés. Le chef est celui qui décide de la stratégie et qui transmet à son équipe les noeuds où ils doivent se rendre.

Calcul de la stratégie

Le chef d'équipe a 4 missions :

- Déterminer où est le golem, deux possibilités :
  - L'agent ne connaît la position d'aucun golem, il considère le noeud ayant le compteur de `stench` le plus élevé.
  - Lors de ses déplacements, en `Gathering` ou pendant la chasse, il est tombé sur un golem ou un agent à portée est tombé sur un golem, il a donc une position précise.
- Calculer le noeud objectif :
  - Pour choisir sur quel noeud l'équipe va essayer de bloquer le golem, le chef détermine tous les noeuds possibles, c'est à dire ceux dont l'arité est inférieure ou égale au nombre d'agents dans l'équipe.
  - Il calcule la distance séparant la position supposée du golem de chaque noeud possible.
  - Il utilise une fonction heuristique pour choisir un noeud en fonction de l'arité de ce noeud et de la distance à parcourir. L'objectif est de minimiser le nombre d'agents requis tout en évitant de trop déplacer le golem car le golem peut s'échapper de la ligne.
- Calculer la ligne de rabattage :
  - Pour forcer le golem à aller vers un noeud sans le perdre d'un déplacement à l'autre, il est nécessaire de former une ligne derrière ce dernier. Le chef doit calculer les noeuds de cette ligne.

- Premièrement, on veut forcer le golem à se rapprocher de notre objectif. S'il est à une distance d de l'objectif, on veut l'emmener sur le noeud voisin à distance $d-1$. Il peut en avoir plusieurs. Pour forcer le déplacement vers ce noeud, il faut bloquer tous les autres déplacements possibles du golem depuis sa position. La ligne continent donc tous les noeuds voisins du golem qui sont à une distance $d+1$ de la destination.
- Ensuite, on veut s'assurer que le golem ne s'échappe pas par « les côtés », on ajoute alors à la ligne des « ailerons ». La ligne s'apparente alors davantage à un arc de cercle derrière le golem, il ne peut ni reculer, ni aller sur les côtés, il doit avancer dans le sens que l'on souhaite. Ces noeuds là sont ceux à distance 2 du golem et à distance d de l'objectif.
- Enfin, le chef calcule la ligne de la prochaine itération, afin de prévoir les déplacements suivants. Il procède de la même façon que précédemment en considérant que le golem a avancé sur le noeud souhaité.

• Envoi de la stratégie :

- Le chef envoie un message à tous ses subordonnés contenant l'itération courante, la liste des noeuds de la ligne, la ligne suivante, et le noeud objectif.

Application individuelle de la stratégie

Afin de connaître les positions des agents, à chaque déplacement, les agents exécutent le behaviour `SendPingPos` et signalent leur position courante et leur destination.

Les noeuds de la ligne étant enregistrés dans l'ordre, le premier noeud de la ligne est nécessairement un noeud voisin du golem. Le chef a besoin d'être à côté du golem pour vérifier qu'il est bien présent sur le noeud objectif une fois arrivé à destination. Par conséquent, il ira toujours sur le premier noeud de la ligne, même si ce n'est pas le plus proche de lui.

A la réception de la stratégie, chaque agent va essayer de se rendre sur les noeuds de la ligne afin de bloquer le golem. Chaque agent calcule le noeud le plus proche de lui, sans compter le premier noeud de la ligne qui est réservé au chef. Deux agents peuvent tout à fait essayer d'aller vers le même noeud mais l'un d'entre eux sera forcément empêché d'y aller, et grâce au retirage d'arêtes, il recalculera le chemin le plus court vers la ligne et ira probablement vers un nouveau noeud.

Une fois arrivés sur la ligne, les agents attendent le feu vert du chef pour avancer sur la prochaine ligne. Ils attendent donc un message contenant la prochaine itération du plan. Le chef décide s'ils peuvent continuer en vérifiant grâce aux positions envoyées par ses agents qu'il a enregistré si tous ses agents sont sur la ligne ou si tous les noeuds de la ligne sont occupés par ses agents. Si la ligne contient moins de noeuds que la taille de l'équipe, les autres agents sont en train d'essayer d'aller sur ces noeuds, donc ils ne sont pas loin. Il calcule alors la prochaine itération et la transmet à ses agents. Cela enclenche le déplacement.

Lorsque les agents ont réussi à rabattre le golem vers le noeud qu'ils souhaitent, le chef va dissoudre l'équipe. Il envoie alors deux types de messages, `freeze`, ou `free` :

- `Freeze` : les agents sont utiles pour bloquer le golem, c'est à dire qu'ils sont sur la ligne. Ils entrent alors dans le state `Frozen`. Dans cet état, les agents ne font que répondre aux diagnostics qu'ils reçoivent.
- `Free` : les agents ne sont pas sur la ligne, ils sont en trop et peuvent quitter l'équipe et partir chasser d'autres golems. Ils retournent alors dans le `GatheringState`.

Cas particuliers

Pendant la chasse, plusieurs cas particuliers peuvent arriver, voici les stratégies adoptées pour gérer ces cas :

Le chef peut ne connaitre aucune `stench` et ne pas savoir où est le golem. Dans ce cas, il choisit un noeud au hasard sur la carte et s'y rend avec son équipe.

Le golem peut ne pas être proche de l'équipe, dans ce cas, si le chef constate qu'il n'y a pas d'odeur là où il se trouve, il recalcule le noeud avec la plus grande `stench` et renvoie un nouveau plan à son équipe.

Un agent peut trouver le golem à une position nouvelle, auquel cas, il le signale en broadcast et le chef va recalculer la stratégie pour encercler ce nouveau noeud.

Lorsque la stratégie a abouti et que la ligne est positionnée autour du noeud objectif, le chef essaie d'aller sur le noeud objectif, s'il y parvient, c'est que le golem s'est échappé, il remet la `stench` à zéro sur ce noeud et recommence ses calculs.

3.6 Limites

Le fonctionnement de la chasse à de nombreuses limites empêchant un bon fonctionnement de l'algorithme. L'optimisation en pâtit également mais ces problèmes sont négligeables par rapport aux premiers.

Problèmes de fonctionnement

Notre chasse contient des problèmes de fonctionnement, soit à cause d'un manque de temps pour implémenter une solution, soit à cause d'un manque de solution réalisable :

- Lorsqu'il y a plusieurs golems et qu'un des golems est bloqué, les agents en trop, ne bloquant pas le golem, vont rester chasser le golem déjà bloqué à cause du `stench`. Ceci pourrait être résolu en mettant à 0 le compteur de `stench` une fois que les agents inutiles à l'équipe ont été libérés.
- Si un agent est perdu, l'équipe ne s'en rend pas compte. Dans ce cas, on espère que le golem parte de la zone pour que le chef se promène aléatoirement et retrouve son équipe.
- Si la dernière position connue d'un agent est celle sur laquelle je fais un diagnostic, je peux considérer qu'il est toujours là même si en réalité c'est le golem puisqu'il n'y a pas de date d'expiration pour cela. Cependant, nous n'avons jamais rencontré ce problème, donc nous ne sommes pas sûres que cela puisse vraiment arriver.
- Si deux golems sont proches, les agents n'arrivent pas à les différencier et ils peuvent s'envoyer des informations contradictoires au sein d'une même équipe.

Problèmes d'optimisation

Nous avons également quelques problèmes d'optimisation :

- Notre stratégie de chasse gère mal les `stench` qui couvrent une majorité de la carte.
- Le temps que les agents se mettent sur la ligne, les golems ont le temps de s'enfuir.
- L'heuristique peut être améliorée, un noeud à distance 2 d'arité 1 ne sera pas favorisé face à un noeud à distance 0 d'arité 3. Exemple :

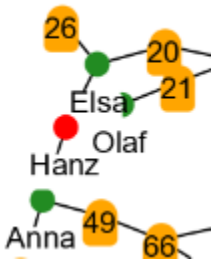

FIGURE 9 – Exemple de problème d'optimisation causé par l'heuristique

- Comme on ne sait pas où est le golem précisément, quand les agents sont sur la ligne, il n'y a pas de moyen de déterminer avec certitude que le golem a avancé dans la bonne direction, ce qui fait perdre du temps.

4 Conclusion

Malgré les problèmes de fonctionnement et d'optimisation rencontrés tout au long de ce projet, la stratégie que nous avons mis en place pourrait obtenir des résultats satisfaisant.

En effet, la stratégie type « battue »a d'ores et déjà fait ses preuves dans la vraie vie, ce qui laisse penser qu'avec plus de temps et de ressources, nous aurions pu implémenter des solutions à nos divers problèmes ce qui nous aurait permis d'avoir un système plus fiable.

Comme expliqué dans les parties développant les limites de nos algorithmes, de nombreuses améliorations pourraient être mis en place comme :

- Gérer les golems immobiles.

- Optimiser les déplacements pendant l'exploration, par exemple en décidant d'un chef temporaire qui envoie des destinations différentes et éloignées aux agents concernés.
- Ne pas autant séparer l'exploration de la chasse.
- Prendre en compte les golems bloqués pour ne pas se laisser piéger par les odeurs de ces derniers.
- Gérer les agents qui se perdent avec un `timeout` qui, si le chef n'a pas de nouvelles d'un agent pendant un temps donné, il le cherche ou le retire de l'équipe. De même, l'agent perdu se retirerait de l'équipe une fois ce `timeout` d'attente dépassé. Cela permettrait également de gérer les agents qui meurent pour une raison quelconque.
- Optimiser l'heuristique de sélection de la cible dans laquelle on veut bloquer le golem. Par exemple, si on a une estimation du nombre de golems, on pourrait calculer grâce au nombre d'agents total combien l'arité maximum du noeud dans lequel on veut bloquer le golem actuel peut être.

Nous avons beaucoup apprécié travailler sur ce projet, et malgré les résultats mitigés obtenus, nous pensons avoir réfléchi consciencieusement pour établir les meilleures stratégies auxquelles nous avons pu penser.