



M1 INFORMATIQUE - PROJET DÉDALE
RAPPORT FoSYMA

[Groupe 11]

Mouna Benabid

Manel Khenifra

Lien de notre dépôt Git

Table des matières

1	Introduction	3
2	Partie 1 - Exploration	3
2.1	Complexité temporelle	5
2.2	Remarques	5
3	Partie 2 - Communication	5
3.1	Exploration	5
3.1.1	Complexité	6
3.2	Chasse	6
3.2.1	Complexité	7
4	Partie 3 - Chasse	7
4.1	Remarques	9
5	Partie 4 - Coordination	10
5.1	Exploration	10
5.1.1	Complexité	10
5.2	Chasse	10
5.2.1	Complexité	11
5.2.2	Remarques	11
6	Conclusion	11

1 Introduction

Ce projet a été mené par Mouna BENABID et Manel KHENIFRA dans le cadre de l'UE FOSYMA du deuxième semestre de Master 1 ANDROÏDE à Sorbonne Université. L'objectif de ce projet est de coordonner plusieurs agents de manière à ce qu'ils puissent explorer la carte sur laquelle ils se trouvent, et chasser puis bloquer le ou les golems qui s'y trouvent avec eux, et ce de manière la plus efficace possible.

On peut noter que le système implémenté doit être opérationnel sur tout type de carte. Le tout a été développé sur l'environnement Dédale, un environnement pour étudier la coordination multi-agents, les problèmes d'apprentissage et de prises de décision.

2 Partie 1 - Exploration

Dans un premier temps, nous nous sommes intéressées à l'exploration multi-agents. L'objectif était de permettre le partage d'information afin que les agents puissent partager leur progrès, de manière à ce que les autres puissent en profiter et ajouter les noeuds explorés à leurs propres noeuds explorés, et n'aient ainsi pas à s'y rendre eux-même. Cela permet ainsi d'empêcher l'interblocage des agents lors de l'exploration, où les deux agents tentent indéfiniment d'atteindre le noeud de l'autre. Le partage de leur carte permet ainsi de noter ce noeud comme exploré puisque l'autre agent l'a effectivement exploré, et de se diriger vers un noeud ouvert.

Cependant on remarque qu'après un partage de cartes, les agents qui ont partagé leurs cartes ont les mêmes noeuds ouverts et auront ainsi une tendance à se suivre. Ce n'est pas idéal car cela n'optimise pas du tout le temps d'exploration et ne prend pas avantage de l'exploration coopérative multi-agents. De plus, cela entraînerait un échange constant de messages qui seraient inutiles.

C'est là que notre algorithme d'exploration entre en jeu. Pour un agent, l'exploration se fait dans un FSM Behaviour qui alterne entre trois états jusqu'au critère d'arrêt.

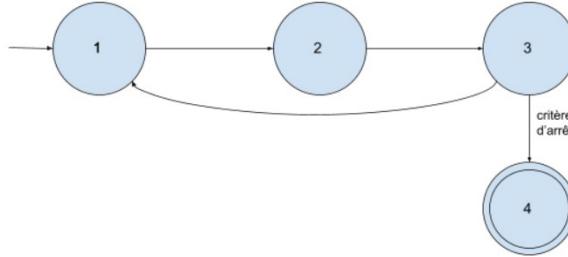


FIGURE 1 – FSM Exploration Behaviour

Le **premier état**, que l'on entre dès le lancement des behaviours à la création de l'agent, est celui où l'agent observe les noeuds autour de lui et choisit son prochain noeud parmi les noeuds observables. A la fin de cet état l'agent entre automatiquement au **deuxième état** qui est l'étape de communication dont on parlera plus en détail dans la *Partie 2 : Communication* ci dessous. Une fois la communication terminée, l'agent entre au **troisième état**, qui est l'état où, si l'agent n'a pas de noeud ouvert observable, il choisit vers quel noeud ouvert de sa carte se diriger. C'est à la fin de ce troisième état que l'agent se déplace puis retourne en état 1. Enfin, le *critère d'arrêt* est l'absence de noeuds ouverts dans la carte, ce qui signifie que l'agent ne se déplace pas et passe au **quatrième état**, qui sera l'état de transition qui va enclencher les behaviours nécessaires au protocole de chasse.

C'est ici des états 1 et 3 que nous allons parler.

État 1 : Dans cet état, on récupère les nœuds observables par l'agent et on les ajoute, s'ils sont de nouveaux nœuds, à la carte de l'agent. On ajoute également les edges qui relient la position actuelle de l'agent aux nœuds observables. C'est pendant ce processus d'ajout des nœuds et edges à la carte qu'on vérifie si l'un des nœuds est un nouveau nœud, et donc un noeud ouvert. Le premier nœud ouvert trouvé sera ainsi stocké dans la variable privée de l'agent *nextNode*, qui représente le nœud sur lequel l'agent se déplacera pour la prochaine itération du FSM. Parce que la variable *nextNode* est une variable privée de l'agent, accessible et changeable à travers des méthodes publiques, on pourra donc accéder à sa valeur dans l'état 3. S'il n'y a aucun nœud ouvert observable, *nextNode* prendra pour valeur null.

État 3 : C'est dans cet état qu'on récupère la valeur de *nextNode*. Si la carte de l'agent n'a plus de nœud ouvert, on quitte le behaviour sans déplacement et on passe en état 4. Si ce n'est pas le cas et que le *nextNode* récupéré a pour valeur un nœud, l'agent se déplace sur ce nœud à la fin du behaviour. En revanche, si *nextNode* a pour valeur null, cela signifie que l'agent n'a pas directement accès à des nœuds ouverts, et c'est dans ce cas qu'on établit un algorithme pour choisir le nœud ouvert qui sera la destination de l'agent.

Dans ce cas, la première étape est de récupérer la liste des positions des autres agents, que l'on a stocké en variable privée de notre agent et à laquelle on a ajouté la dernière position envoyée par un agent lors de la communication, accompagnée de son nom afin de pouvoir remplacer l'ancienne position par la dernière envoyée. On appellera cette liste *otherAgentsPosition*. On calcule ensuite le chemin le plus court entre la position actuelle de l'agent et chaque nœud ouvert de sa carte, et on les range dans une liste qu'on appellera *nodeUs*, de manière à ce que la distance, donc la taille du chemin, soit croissante. On note que dans le pseudo-code, *nodeThem* est, de même, une liste des nœuds ouverts et de leur distance à la position de l'agent selon *otherAgentsPosition*, rangée de manière croissante.

```
Si otherAgentsPosition est vide OU nodeUs n'a qu'un élément (il n'y a qu'un noeud ouvert dans la carte) :
|   nextNode = premier élément du chemin le plus court menant au noeud ouvert le plus proche (soit le premier noeud de la liste nodeUs);
Sinon :
|   i = 0;
Tant que nextNode est null faire :
|   j = 0;
|   Pour other chaque élément de la liste otherAgentsPosition faire :
|   |   nodeThem = liste des nœuds ouverts et de leur distance à la position de other rangée de manière croissante;
|   |   Si le ième noeud de nodeUs est le même que le premier noeud de nodeThem :
|   |   |   Si distance du ième noeud de nodeUs > distance du premier noeud de nodeThem :
|   |   |   |   i = i + 1;
|   |   |   Couper la boucle Pour.
|   |   j = j + 1;
Fin Pour.
Si j == la taille de otherAgentsPosition :
|   nextNode = premier élément du chemin le plus court menant au ième noeud de nodeUs;
Sinon Si i == la taille de nodeUs :
|   Si la taille de nodeUs > la taille de otherAgentsPosition :
|   |   nextNode = premier élément du chemin le plus court menant au dernier noeud de nodeUs;
|   Sinon :
|   |   nextNode = premier élément du chemin le plus court menant au noeud ouvert le plus proche (soit le premier noeud de la liste nodeUs);
Fin Tant que.
```

FIGURE 2 – Algorithme d'exploration

De cette manière, à moins qu'il n'y ait pas assez de noeuds ouverts, on a l'assurance que l'agent le plus proche d'un noeud ouvert se dirigera vers lui tandis que l'agent le plus loin se dirigera vers un autre noeud. Les agents ne se suivront donc pas. On remarque que la liste *otherAgentsPosition* peut contenir des positions d'agents fausses, car reçues il y a longtemps par ces agents. Cependant on considère que ce n'est pas un problème et on garde ces valeurs afin de permettre à notre agent de continuer de s'approcher du noeud de destination choisi, plutôt que de revenir à son premier choix dès qu'on vide la liste *otherAgentsPosition*. Une idée serait d'établir un compteur au bout duquel on supprime un agent et sa position de la liste afin de limiter les calculs, mais cela ne nous a pas semblé assez pertinent pour l'implémenter au risque de ne pas choisir une limite de compteur adaptée.

2.1 Complexité temporelle

Soit N le nombre de noeuds de la carte, E le nombre d'edges, A le nombre d'agents dont on contient la position dans *otherAgentsPosition*, n le nombre de noeuds ouverts de la carte.

Le calcul de *nodeUs* a pour complexité $O(n*(N + E)*\log(N) + n*\log(n))$, car on applique un algorithme de Dijkstra n fois + un algorithme de tri sur n éléments. Dans le pire cas, le calcul de *nodeThem* a pour complexité $O(n*(A*(n*(N + E)*\log(N)+n*\log(n))))$ car on applique l'algorithme de Dijkstra n fois + un algorithme de tri sur n éléments pour chaque affectation de *nodeThem*, et ce A fois pour A passages dans la boucle pour, et n fois pour n passages dans la boucle tant que.

Dans le meilleur cas l'algorithme a une complexité de $O(n*(N + E)*\log(N) + n*\log(n)+ C)$ pour C les comparaisons, affectations et additions. Dans le pire cas l'algorithme a une complexité de $O(n*(N + E)*\log(N) + n*(A*(n*(N + E)*\log(N))) + (A*n^2 + n)*\log(n) + C)$ pour C les comparaisons, affectations et additions.

2.2 Remarques

On remarque que lorsqu'un agent croise un golem durant l'exploration, nous avons deux objectifs incompatibles : bloquer le golem sur la feuille inexplorée et le débloquer pour explorer la feuille. Nous avons choisi de prioriser l'exploration, et ainsi lorsqu'un agent remarque avoir heurté un golem (il n'est pas au noeud sur lequel il se dirigeait et il n'a pas reçu de message d'un agent dont la position était ce noeud), il enlève ce noeud de la liste de noeuds ouverts vers lesquels il souhaite se diriger.

Nous avons fait ce choix pour permettre au golem de ne pas être bloqué à l'exploration afin que les agents puissent terminer leur exploration sans problème et ainsi être sûrs que leur carte est correcte pour une chasse plus efficace.

Nous n'avons cependant pas pris en compte le cas où un golem resterait sur un noeud du début à la fin de l'exploration, refusant de bouger tout du long et ainsi empêchant la fin de l'exploration. Cela s'explique par le fait que ce n'était pas un comportement qui nous semblait réaliste et ainsi nous n'avons pas jugé bon de prendre en compte cette situation.

3 Partie 2 - Communication

3.1 Exploration

Lors de l'exploration, les agents s'arrêtent pour communiquer afin de partager leur position et leur carte dans le but de répartir l'exploration entre eux. Les agents communiquent après chaque déplacement : un agent envoie un ping, si il reçoit une confirmation à son appel il doit s'arrêter

pour échanger sa position et sa carte avec l'autre agent. Ce dernier, étant le destinataire, devra donc s'arrêter lorsqu'il recevra un ping, il enverra un message de confirmation et réceptionnera le partage de carte de l'autre agent, ainsi que sa position. Si il n'y a pas de réponse aux appels lancés, les agents continueront d'explorer.

Nous avons utilisé un FSM Behaviour afin d'ordonnancer les comportements de nos agents sous la forme d'un automate. Le **deuxième état**, présenté dans la *Figure 1* ci-dessus, fait référence à la communication où l'agent lancera des appels dans le but de recevoir une réponse à ceux ci et ainsi échanger sa carte avec les agents qui sont à proximité. Cet état lancera deux comportements en parallèle, d'un côté l'envoie de ping puis l'envoie de sa carte et de sa position si il reçoit une réponse positive. De l'autre côté la réception de ping afin d'y répondre et de réceptionner la carte et la position de l'autre agent. En plus de ces comportements il a un autre comportement parallèle qui réceptionne la carte des agents en état de chasse.

L'envoi de confirmation aux appels lancés permet de s'assurer que les deux agents s'arrêtent pour communiquer et qu'ils soient synchronisés pour pouvoir s'échanger des informations vraies à un temps donné. Cependant, il peut y avoir beaucoup d'échanges entre les agents lorsqu'ils se rencontrent plusieurs fois.

3.1.1 Complexité

Soit n le nombre d'agents, si un agent communique au moins une fois avec chaque agent donc $(n-1)$ fois, le nombre de pairs d'agents qu'on peut avoir est $\frac{n(n-1)}{2}$. Soit i le nombre d'interactions de chaque agent ($i = 1$ dans le meilleur des cas et peut avoir une grande valeur dans le pire des cas), et soit m le nombre de messages envoyés à chaque pas de temps ($m = 1$ dans le meilleur des cas ou $m = 3$ dans le pire des cas lors de l'exploration). Le nombre de messages est alors borné par :

$$\frac{n(n-1)}{2} * i * m$$

Les messages qui concernent l'envoi de ping ou l'envoi de réponse à celui ci ont une taille négligeable qui vaut un bit d'information, cependant l'envoi qui concerne la carte et la position a une valeur plus grande, soit N le nombre de noeuds et E le nombre d'arcs, ce qui vaudrait $(N + E + 1)$ bits d'information.

3.2 Chasse

Après l'exploration, les agents passent à la chasse. Il y a plusieurs états dans cette phase qui prennent en compte la communication : le **troisième état**, visible sur la *Figure 3* ci-dessous, est celui où l'agent peut recevoir un ping de la part d'un autre agent qui n'a pas encore fini son exploration et dans ce cas il lui enverra sa carte (et sa position) afin qu'il puisse lui aussi passer à la chasse.

Dans le **deuxième état**, l'agent envoie sa position, la position de l'adversaire et la liste d'odeurs aux agents à proximité afin de se coordonner pour bloquer l'adversaire, dans le **quatrième état** il va réceptionner ces informations. De même pour le **sixième état** où l'agent va également réceptionner ces informations afin de vérifier si l'adversaire a été bloqué et une fois assuré de cela, il enverra un message contenant sa position, la position de l'adversaire et une chaîne de caractère indiquant l'état de sa chasse, à savoir "FINISHED".

3.2.1 Complexité

Soit n le nombre d'agents, si un agent communique au moins une fois avec chaque agent donc $(n-1)$ fois, le nombre de pairs d'agents qu'on peut avoir est $\frac{n(n-1)}{2}$. Soit i le nombre d'interactions de chaque agent, et soit m le nombre de messages envoyés à chaque pas de temps ($m = 0$ dans le meilleur des cas ou $m = 2$ dans le pire des cas lors de la chasse). Le nombre de messages est alors borné par : $\frac{n(n-1)}{2} * i * m$

Soit n la taille de la liste de noeuds à odeur, la taille du message envoyé dans le deuxième état vaut $(n + 2)$ bits d'information. Dans le troisième état, si l'agent reçoit un ping d'exploration, il envoie un message de $(N + E + 1)$ bits d'information, pour N le nombre de noeuds de la carte et E le nombre d'arcs. Dans le sixième état, le message vaut 3 bits d'information.

4 Partie 3 - Chasse

Une fois l'exploration terminée, l'agent passe dans le dernier état d'exploration qui est un état de transition, qui lance le FSM Behaviour qui permettra à l'agent de chasser le golem qui se trouve sur le graphe.

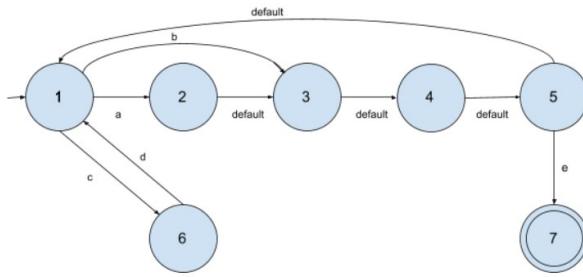


FIGURE 3 – FSM Chasse Behaviour

Le **premier état** est celui où l'agent observe ses alentours et en fait des constatations qui décideront de ce qu'il enverra comme message, ou non, durant la communication. Les **deuxième, troisième, quatrième et sixième états** sont des états de communication, qui sont expliqués plus en détail dans la *Partie 2 : Communication* ci-dessus. Le **cinquième état** est celui dans lequel l'agent choisit l'action à prendre selon ce qu'il a appris, ou non, durant la communication, et dans le premier état d'observation. Enfin le **septième état** est un état de clôture, qui ne pourrait être atteint par l'agent que dans des conditions spéciales dont nous parlerons plus tard.

C'est ici sur les états 1 et 5 que nous allons nous pencher.

C'est dans l'**état 1** que l'agent observe les noeuds qui l'entourent et les met dans une liste *obs-
Nodes*, une variable de l'agent, puis parmi ces noeuds observables il met ceux qui ont une odeur dans la liste *nodesStench*, une variable de l'agent. Si *nodesStench* est vide, cela remplira la condition *b* et l'agent ne passera que par les états 3 et 4 qui sont des états de réception de message. En revanche si *nodesStench* n'est pas vide, cela remplira la condition *a* et l'agent passera par l'état 2, qui est un état d'envoi de message, avant les 3 et 4.

L'agent vérifie également dans cet état s'il est à la position à laquelle il croyait être le golem, stockée dans la variable *positionGolem*. Si oui alors le golem ne peut pas y être et l'agent réinitialise

la variable à null et remet le compteur *compteurGolem*, qui compte le nombre d'itérations du FSM où il a gardé la même valeur de *positionGolem*, à zéro, ainsi que réinitialise la liste *finish*, variable de l'agent qui stocke le nom et la position des agents qui bloquent le golem avec lui. De même, si l'agent a reçu de la part d'un autre la position à laquelle il croit être le golem pendant la communication, cette position a été stockée dans la variable *givenPosGolem* et l'agent vérifie s'il est à cette position. Si oui alors le golem n'y est pas et il réinitialise la variable. Si non, mais que cette position fait partie des noeuds observables par l'agent et que ce noeud ne fait pas partie des noeuds à odeur, dans ce cas le golem ne peut pas y être et l'agent réinitialise la variable. De même, l'agent vérifie si la position stockée dans *givenPosGolem* est la même que la position d'un des agents qu'il a reçu durant la communication. Si oui, il réinitialise la variable.

Si l'agent remarque ne pas être à la position sur laquelle il se dirigeait, cela veut dire que quelqu'un le bloque. Il vérifie donc si la liste *nodesStench* est vide, et si non, il vérifie si la position sur laquelle il n'a pas réussi à aller fait partie de noeuds observables à odeur. Si oui, il vérifie si un agent qui lui a envoyé sa position durant la communication y était. Si non, l'agent suppose qu'un golem était à cette position et stocke le noeud dans la variable *positionGolem*.

Si le compteur *compteurGolem*, qui compte le nombre d'itérations du FSM où il a gardé la même valeur de *positionGolem*, a une valeur supérieure à 50, l'agent vérifie si le noeud contenu par *positionGolem* est une feuille à l'aide de la fonction *sendNodeEdges* qui rend le nombre d'edges d'un noeud, ainsi qu'une liste des noeuds qui sont reliés à lui par un edge. Si le noeud est bel et bien une feuille car la fonction ne retourne qu'un edge, alors l'agent a rempli la condition *c* et entre en état 6, qui est l'état entré lorsqu'un agent pense avoir confirmé sa capture d'un golem. Si la position du golem a plusieurs edges, l'agent vérifie donc si un autre agent est sur chaque noeud relié à celui du golem en dehors du sien. Ces positions sont stockées dans une liste *finish*, variable de l'agent, pendant la communication. S'il est assuré qu'un agent différent est sur chaque noeud relié à celui du golem en dehors du sien, dans ce cas l'agent a rempli la condition *c* et entre en état 6.

Après communication, l'agent se trouve dans l'**état 5**.

- Si l'agent n'a pas de noeud à odeur observable, n'a pas reçu de message durant la communication et n'a pas de valeur dans la variable *givenPosGolem* :

L'agent supprime de la liste de noeuds observables les feuilles, car il sait que le golem ne peut pas y être s'il n'y a pas d'odeur et cela le ferait se diriger dans une impasse sans raison. Puis, si la liste comporte plus d'un élément, il supprime de la liste le noeud auquel il était à sa position précédente (ce noeud est stocké dans la variable d'agent *lastPosition*). Enfin, il choisit son *nextNode* en faisant un random sur la liste de noeuds restants.

On a choisi d'utiliser un random pour rendre l'agent imprévisible pour le golem, et parce que d'autres stratégies de recherche de noeuds à odeur nécessiteraient des communications complexes qui n'apporteraient pas assez d'intérêt à notre avis. On a cependant conscience que le choix du random signifie aussi que l'agent peut mettre plus ou moins longtemps à repérer un noeud observable.

Complexité temporelle : N le nombre de noeuds observables. Dans le pire cas, $O(8 + N*(K + 5) + C)$ pour K la constante représentant le nombre d'opérations élémentaires associées à la fonction *sendNodeEdges* et C au choix d'un noeud random.

- Sinon, si l'agent a une valeur dans la variable *positionGolem* :

Son *nextNode* sera la valeur de *positionGolem*, et il augmente le compteur *compteurGolem* de 1.

Complexité : Complexité temporelle $O(3)$ et spatiale $O(2)$.

- Sinon, si l'agent a reçu pendant la communication une position où un autre agent pense que

le golem est, et si ce noeud n'est pas une feuille :

Il utilise la fonction `shortestPathNewMap(myPosition, golem, others)` qui crée un nouveau graphe reproduisant le graphe de l'agent, dont tous les edges reliés aux noeuds `others`, qui est une liste des positions des autres agents, ont un attribut "length" fixé à 1000 tandis que pour les autres edges il est fixé à 1. Cette fonction utilise ensuite un algorithme Dijkstra qui veut minimiser l'attribut "length" pour trouver le plus court chemin de sa position, `myPosition`, à celle du golem, `golem`. Son `nextNode` prendra la valeur du premier élément du chemin, et la variable d'agent `givenPosGolem`, un couple de string et liste de string prendra la valeur de `golem` accompagnée de la liste de la position des autres agents `others`.

Complexité temporelle : N le nombre de noeuds et E le nombre d'arcs du graphe. Complexité bornée par $O((N + E) * (\log(N) + C) + K)$ pour C la constante associée à l'ajout d'un noeud ou d'un arc à un graphe, et K celle à l'affectation de `nextNode` et à celle de `givenPosGolem`.

- Sinon, si la variable `givenPosGolem` a une valeur :

L'agent utilise la fonction `shortestPathNewMap(myPosition, givenPosGolem.getRight(), givenPosGolem.getLeft())` et son `nextNode` prend comme valeur le premier élément du chemin.

Complexité temporelle : N le nombre de noeuds et E le nombre d'arcs du graphe. Complexité bornée par $O((N + E) * (\log(N) + C) + K)$ pour C la constante associée à l'ajout d'un noeud ou d'un arc à un graphe, et K celle à l'affectation de `nextNode`.

- Sinon, si l'agent n'a pas de noeud à odeur observable, mais a reçu un message durant la communication :

Soit `otherNodesStench` la liste des informations données pendant la communication, qui contient la position de l'agent qui a envoyé le message, la position du golem s'il l'a trouvée et null sinon, et une liste de noeuds à odeur qu'il a observé. L'agent calcule le plus court chemin menant de sa position à chacun des noeuds à odeur de la liste reçue. Enfin, il choisit comme `nextNode` le premier élément du chemin le plus court parmi tous ceux calculés.

Complexité temporelle : N le nombre de noeuds et E le nombre d'arcs du graphe, n le nombre de noeuds à odeur reçus par message. Dans le pire cas $O(n * (N + E) * \log(N) * K + 5n + 5)$ pour K la constante associée à l'ajout d'un chemin et de sa taille à une liste.

- Sinon, si l'agent a des noeuds à odeur observables :

Il compte le nombre de feuilles parmi les noeuds observables, puis compte le nombre de feuilles parmi les noeuds à odeur observables.

Si la liste de noeuds observables contient plus d'une feuille mais qu'il n'y a qu'une feuille à odeur observable, le golem est peut-être dans cette feuille à odeur observable. L'agent choisit donc cette feuille comme `nextNode`.

Sinon, si la liste de noeuds à odeur observables contient plus d'un élément, l'agent supprime de la liste sa position précédente, stockée dans `lastPosition`, si la liste la contient. L'agent choisit ensuite comme `nextNode` un noeud random parmi la liste.

Complexité temporelle : N le nombre de noeuds observables et n le nombre de noeuds à odeur observable. Dans le pire cas, $O((N + n) * (5 + C) + n + K)$ pour C représentant le nombre d'opérations élémentaires associées à la fonction `sendNodeEdges`, et K celle aux opérations élémentaires d'affectation et au coût associé au choix d'un noeud random.

4.1 Remarques

On remarque que, parce que les agents n'ont pas le moyen de savoir le nombre de golems présents sur le graphe, il n'y a pas de possibilité pour eux d'entrer l'état 7. En effet, parce qu'ils n'ont pas d'assurance d'avoir capturé tous les golems, les agents qui ne participent pas à une capture sont obligés de patrouiller indéfiniment à la recherche d'un golem potentiel. Le seul moyen pour tous les

agents d'être au repos dans l'état 6 ou 7 serait que tous les agents sans exception participent à la capture d'un ou plusieurs golems, ou que les agents aient un moyen d'être certain du nombre n de golems présents, et ainsi d'entrer en état 7 lorsqu'ils reçoivent n messages contenant un *positionGolem* différent accompagné d'une chaîne de caractère "FINISHED".

5 Partie 4 - Coordination

5.1 Exploration

Les agents partagent leur carte par le biais de la communication, puis l'algorithme d'exploration, expliqué dans la *Partie 1 : Exploration* ci-dessus, leur permet de se coordonner pour ne pas aller aux mêmes noeuds ouverts. On n'a pas mis en place de protocole particulier pour mettre tous les agents au courant de la fin de l'exploration, car lorsqu'un agent est en mode chasse, il peut toujours recevoir les messages d'exploration et envoie sa carte à celui qui lui a envoyé ce message. Cela signifie que, lorsqu'un agent qui a fini l'exploration rencontre un agent qui ne l'a pas finie, ce dernier est forcément au courant que l'exploration est terminée car lorsqu'il reçoit la carte de l'agent qui a fini l'exploration, il l'ajoute à la sienne et termine ainsi l'exploration lui aussi.

5.1.1 Complexité

Soit N le nombre de noeuds et E_i le nombre d'arcs d'un noeud i de la carte que l'agent veut fusionner avec la sienne.

La fusion des cartes a, dans le pire cas, pour complexité temporelle $O(N^*(16 + K) + N^*(R + E_i*C))$ avec K la constante représentant le coût de l'ajout d'un node au graphe, R la constante représentant le coût des opérations élémentaires associées à l'obtention des arcs d'un noeud i et C celle à l'ajout d'un arc à un graphe.

5.2 Chasse

Lors de la chasse, l'agent ne considère avoir capturé un golem que lorsqu'il sait, soit que le golem est sur une feuille, soit que chaque noeud attaché à celui du golem, autre que le sien, est occupé par un agent, et que cet agent occupe bel et bien ce noeud dans le but de capturer le golem. C'est dans le processus de communication que l'agent s'en assure.

Une fois que le compteur *compteurGolem*, qui compte le nombre d'itérations du FSM Behaviour où l'agent a gardé la même valeur de *positionGolem*, a atteint 50, l'agent se met à stocker les positions des agents qui ont la même valeur de *positionGolem* que lui, ce qui signifie qu'ils tentent de capturer le même golem. Il stocke leur position ainsi que leur nom dans une liste *finish*, si ces agents se révèlent être positionnés à un noeud adjacent à celui du golem et ainsi participent à sa capture. Si un agent déjà présent dans la liste envoie un nouveau message depuis une position différente qui ne participe pas à la capture du golem ou n'a plus la même valeur de *positionGolem*, il est retiré de la liste.

Ce n'est que lorsque la liste *finish* comporte un agent positionné à chaque noeud adjacent à celui du golem que l'agent remplit la condition c et entre en état 6, un Simple Behaviour de réception de message où il reçoit les messages de chasse et y répond par son propre message qui contient dans sa liste de noeuds à oïdeur le string "FINISHED". S'il reçoit un message provenant d'un des agents de la liste *finish* et que celui-ci est à une position différente de celle de la liste ou n'a plus la même valeur de *positionGolem*, il est retiré de la liste et l'agent remplit la condition d et retourne en état 1. S'il reçoit un message provenant d'un agent qui est à la position où il croyait être le golem, la valeur de *positionGolem* devient null, il réinitialise le *compteurGolem* ainsi que la liste *finish*, et retourne en état 1 selon la condition d . C'est de cette manière, ainsi que par le biais de la communication où les agents partagent leur valeur de *positionGolem* ainsi que celle de leur propre position, ce qui permet aux agents avec lesquels ils communiquent de participer à la capture du golem en se dirigeant vers lui en prenant un chemin n'incluant pas la position de l'agent et ainsi prendre le golem en "sandwich", que les agents peuvent chasser et capturer le golem en équipe.

Les agents sont également mis au courant lorsqu'une chasse est terminée. Lorsqu'ils reçoivent lors de la communication une liste de noeuds comportant le string "FINISHED", cela signifie que le golem a déjà été capturé et donc que l'agent devrait retourner patrouiller autre part sur le graphe à la recherche d'un potentiel autre golem. L'agent s'assure donc qu'il ne fait pas partie des noeuds adjacents au *positionGolem* associé au "FINISHED" envoyé par l'autre agent, ce qui signifierait qu'il participe à la capture du golem et que sa présence est essentielle, et sinon il compile le plus court chemin vers un noeud random jusqu'à ce que ce chemin ne comporte ni la position du golem capturé ni ses noeuds adjacents. Il continue de choisir son *nextNode* de cette manière jusqu'à ce qu'il ait passé trois itérations du FSM Behaviour sans avoir de noeud à odeur observable, ce qui assurerait qu'il n'est plus dans l'entourage du golem capturé et peut continuer sa patrouille, ou jusqu'à ce qu'il ait passé vingt itérations du FSM Behaviour en utilisant cette méthode.

5.2.1 Complexité

La complexité temporelle associée au comportement de l'agent lorsqu'il reçoit un message "FINISHED" est bornée par $O(N^*(N + E)^*\log(N) + C)$ pour N le nombre de noeuds et E le nombre d'arcs du graphe, et C une constante représentant les affectations, comparaisons et additions.

5.2.2 Remarques

La raison pour laquelle l'agent doit attendre que le *compteurGolem* atteigne 50 avant de se mettre à remplir la liste *finish* ou avant d'entrer en état 6, est afin de limiter les erreurs telles que l'agent était en fait en train de capturer un autre agent mais n'avait pas encore reçu son message à cause d'un problème de communication. Une grande valeur de *compteurGolem* permet ainsi une plus grande assurance, et nous a semblé avoir un coût assez bas sachant que du moment que la valeur de *positionGolem* ne change pas et donc que l'agent est bel et bien en train de chasser et capturer le golem, la complexité algorithmique est assez basse.

De même, la raison pour laquelle l'agent doit s'assurer que le noeud sur lequel est le golem est soit une feuille, soit encerclé par des agents dans le but de capturer le golem, est afin de s'assurer que la capture du golem est effectivement complète et pas seulement une impression parce que le golem n'a pas bougé pendant un long moment.

6 Conclusion

Les choix algorithmiques que nous avons fait nous ont mené à un taux de réussite de chasse de 100%. Cependant, parce que nous avons fait le choix de prioriser l'exploration lorsqu'un agent rencontre un golem durant l'exploration afin de s'assurer que tous les agents aient accès à une carte complète lors de la chasse, et parce que nous avons choisi de donner un caractère plus ou moins aléatoire aux patrouilles des agents en état de chasse, la capture d'un golem peut prendre plus ou moins longtemps.

Des idées d'amélioration porteraient tout d'abord sur les patrouilles des agents en état de chasse. Nous avons choisi leur caractère aléatoire pour permettre aux agents d'être imprévisibles, mais cela peut aussi avoir des défauts car le temps passé avant que l'agent ne trouve un noeud à odeur est également imprévisible. Nous avons également choisi cette option pour limiter les besoins de communication lors d'un état de patrouille. Cependant il pourrait être possible de trouver une alternative permettant une fouille de la carte plus méthodique.

Nous aimerais également trouver un moyen pour les agents de s'assurer du nombre de golems présents sur la carte, afin que les agents ne participant pas à la capture des golems n'aient pas besoin de patrouiller indéfiniment à la recherche d'un golem potentiel après avoir reçu un message de capture de chaque golem présent.

Enfin, la communication lors de l'exploration peut s'avérer coûteuse dans certains graphes tels que les arbres, où les agents sont constamment à proximité lorsqu'ils sont dans le même arbre, même lorsqu'ils ne se dirigent pas vers le même noeud. Une idée d'extension afin d'alléger le coût d'envoi des cartes aurait été de stocker la carte envoyée par un agent et de ne lui envoyer lors de la prochaine communication que les noeuds et arcs qu'on sait qu'elle ne contient pas déjà. Cependant, cela impliquerait également de fusionner deux cartes envoyées par le même agent afin de remplir les trous, et nous ne sommes pas sûres que cela réglerait

le problème de la communication coûteuse.

Un début d'implémentation de cette idée est mise en commentaire dans notre code.