Rapport

Projet FOSYMA

Encadrant :

Cedric Herpson

Equipe :

Jérémy Dufourmantelle
21104331
Ethan Abitbol 3804139
Groupe 1

Nicolas Maudet

Aurelie Beynier

sujet : Variante multi-agent du jeu « Hunt the Wumpus »

Table des matières

1 Introduction

En 1972, "hunt the wumpus" est l'un des premiers jeux informatiques, développé par Gregory Yob. Il consiste en une partie de cache-cache avec un monstre, le Wumpus caché dans une caverne tout en ramassant des trésors. Dans notre projet, nous ne nous consacrons que a la récolte de trésor, nos agents doivent explorer l'environnement et ramasser des trésors disséminés dans l'environnement. Pour cela, nous allons donc implémenter un Système Multi Agents dans lequel les agents peuvent communiquer afin de pouvoir explorer mais aussi communiquer pour coopérer afin de satisfaire la contrainte d'équité / efficacité de la récolte des trésors.

Différents types d'environnements sont considérés, les arbres, la grille ou encore les graphes. Chaque agent est doté de sacs à dos avec des capacités hétérogènes, un rayon de communication et des perceptions limitées. De plus, deux types de trésors, OR et diamant ainsi que deux golems sont présents dans l'environnement.

Nous utiliserons le langage de programmation Java et la plateforme multiagent JADE.

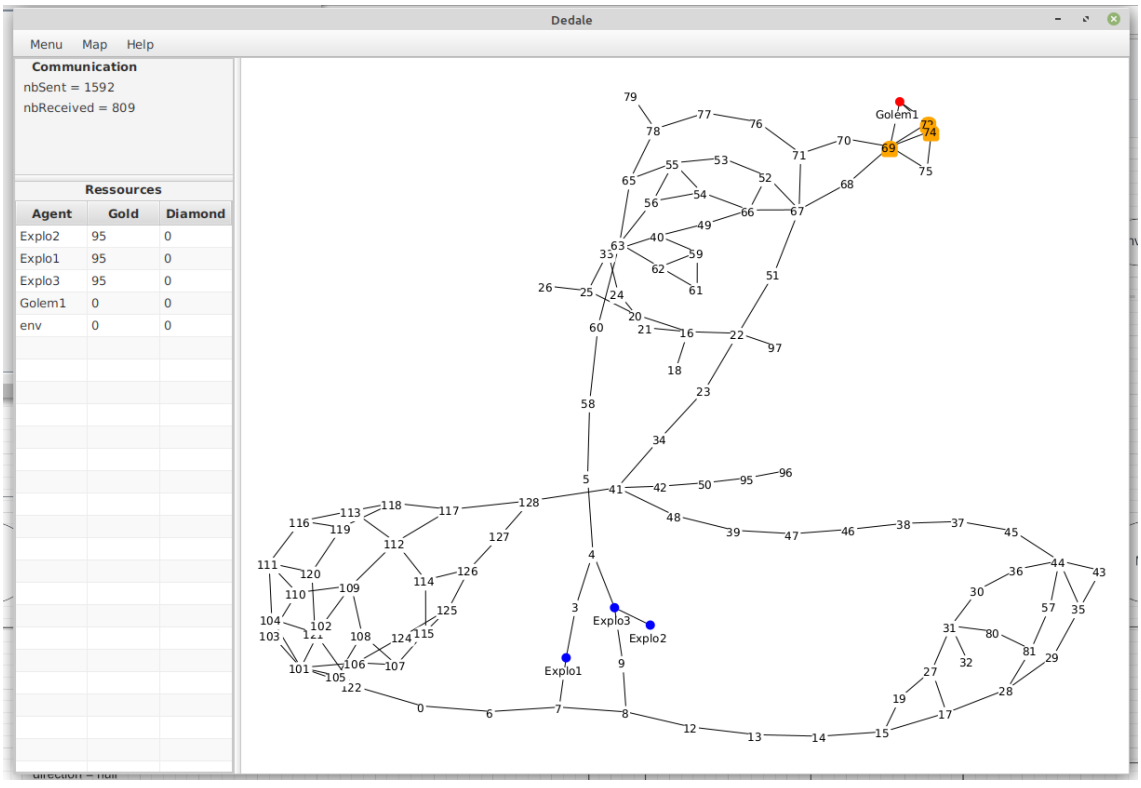

FIGURE 1 – Exemple d'exécution

2 Présentation des choix associés

Tout au long du projet, Nous avons décidé de nous baser sur une FSM (Final State Machine) pour pouvoir contrôler la séquentialité et de pouvoir passer d'un béhaviour à un autre de façon automatisé dès lors que celui-ci se termine.

2.1 Exploration

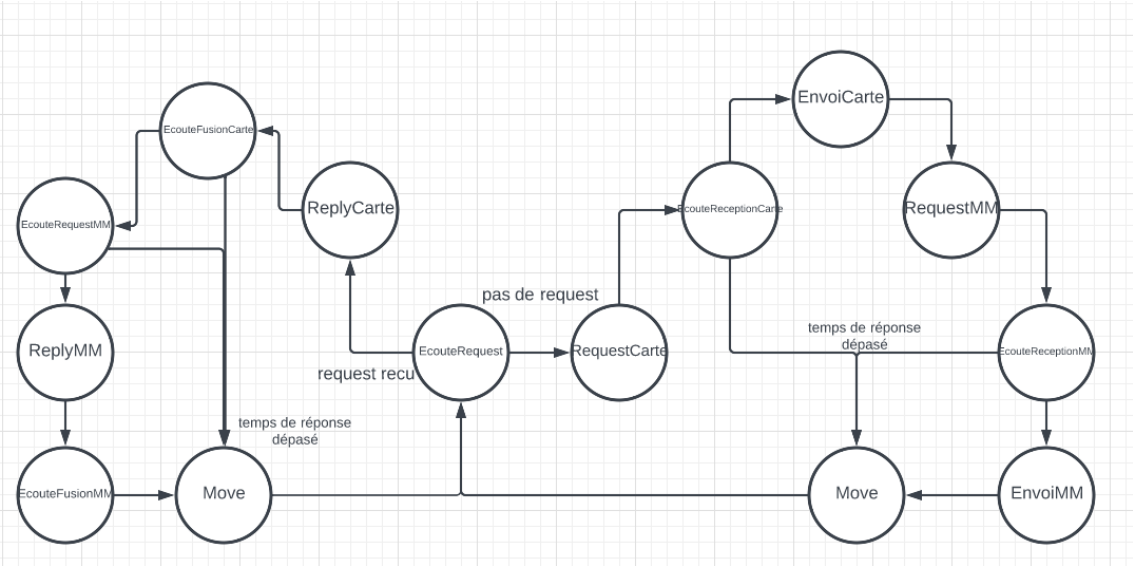

FIGURE 2 – FSM utilisé pour l'exploration

2.1.1 Principe, forces et limites

Pour ce qui est de l'exploration, nos agents utiliseront deux types de connaissances : La carte **Map** de l'ensemble des noeuds ouverts (à explorer) et fermés (déjà exploré) ainsi qu'un **MapManager** qui permet de faire des opérations et de stocker toutes les données propres à chacun des noeuds (que l'on nomme **Field**) comme la quantité de diamant, d'or ou encore le temps d'inactivité.

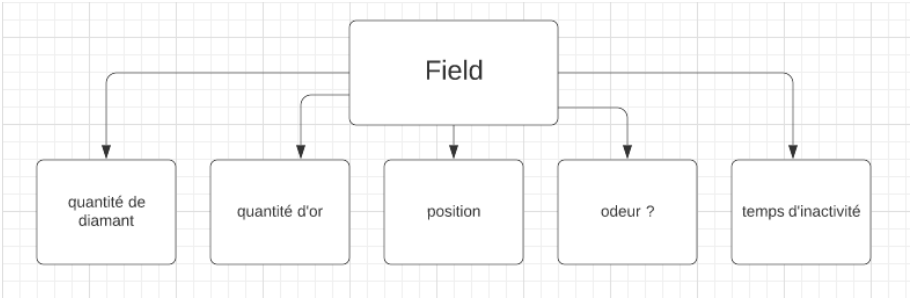

FIGURE 3 – MapManager

L'agent retient chaque noeud avec un type différent, soit or, soit diamant ou soit none. Cela nous permet de pouvoir mettre à jour les **Field** lorsque le wumpus déplace de l'or dans une case précédemment à none. De plus, le temps d'inactivité correspond a un compteur initialement a 0 et incrémenté de 1 a chaque déplacement, donc chaque case aura une valeur différente et reflétera le nombre de 'saut' depuis la case initial vers cette case. Cela nous permet lorsqu'il y a communication entre deux agent de comparer leur valeurs pour une même cases et prendre la valeur la plus élevé, celle qui correspond a la plus récente.

Nous avons aussi optimisé cette exploration pour que chaque agent envoi uniquement les **Field** contenant les informations pertinantes comme les **Field** contenant de l'or ou du diamant pour éviter d'allourdir les communications avec tous les **Field** de la carte.

De plus, lors de la phase d'exploration, notre agent peut rencontrer un wumpus lorsqu'il souhaite aller sur un noeud ouvert. Si un tel cas arrive, alors ce noeud est placé dans une liste de noeud ouvert à revisité plus tard (tous les x pas de temps). Avec cette méthode, cela résout le problème d'interblocage agent-wumpus et permet à l'agent de continuer l'exploration des autres noeuds ouvert. Si les noeuds ouverts ont été bloqué par le wumpus (exemple d'un couloir) alors l'agent passe en mode récolte pour éviter de se retrouver bloquer et de perdre du temps.

Nous avons utilisé l'algorithme DFS déjà implémenté qui possède de bonne propriété d'exploration dans un graphe car il ne va pas passer par des noeuds déjà explorés comme pourrait le faire l'algorithme BFS.

Enfin, les agents après avoir fini la phase de récolte retournerons dans la phase d'exploration, en explorant les noeuds avec le plus grand temps d'inactivité pour vérifier qu'il n'y a plus de ressource à récupérer qui aurait été déplacé par le wumpus.

2.1.2 Complexité (temps, mémoire, communication)

Nous utilisons deux structures de données pour la phase d'exploration : un graphe $G = (V, E)$ qui représente la carte et une liste d'objets (6-uplets : 4 entiers et 2 chaines de caractères) contenant les informations des différentes zones. Ainsi nous obtenons une complexité spatiale par agent de :

$$O(|V| + |E| + |V| * (2 * size(chaine) + 4 * size(int))) \text{ en bytes en mémoire}$$

On peut estimer la complexité temporelle par agent en fonction des états franchis par l'agent dans la FSM avec le nombre d'état à franchir lorsqu'il ne communique pas $= 4$ (**EcouteRequest-RequestCarte-EcouteReceptionCarte-Move**), le nombre de noeud total du graphe à visiter N et une constante T pour le temps d'attente rajouté artificiellement afin d'éviter les problèmes de synchronisation avec l'API :

$$O(NT) \text{ en temps d'attente}$$

En effet, toute les opérations et mise à jour de connaissance ce font en $O(1)$. Si il y a communication entre deux agents avec un gain de K noeuds déjà exploré par l'autre agent et t_1 et t_2 les temps respectifs d'attente des réponse lors de l'échange, on obtient :

$$O((N - K)T + t_1 + t_2) \text{ en temps d'attente}$$

Comme nous explorons un nombre de noeud inférieur avec cette échange et que t_1 et t_2 sont très faible, alors cette complexité est meilleure.

Pendant la communication, chaque agent doit transmettre une fois sa carte et il y a un agent qui est désigné pour faire la fusion et transmettre le résultat, ce qui fait 2 échanges. De plus, en complément, une liste est envoyé contenant uniquement les m endroits d'intérêt (or ou diamant) pour éviter d'alourdir le réseau avec des informations inutiles. Cela nous donne les complexité spatiale suivante lors de M partage d'informations cartographique pendant la simulation et c la variable représentant les demandes request faite par chaque agent :

$O(2M(|V| + |E| + m * (2 * size(chaine) + 4 * size(int))) + c)$ en bytes sur le réseau

2.1.3 Optimalité et critère d'arrêt

Les agents terminent leur exploration quand ils n'ont plus de noeuds ouverts et cela est toujours atteint grâce aux différentes méthodes pour gérer les inter blocages mais aussi les processus de communication qui permettent aux agents de combler les noeuds ouverts sur leurs connaissances de la map.

2.2 Communication

Dans notre système, nous voulons que les agents s'échange 3 types d'information : les informations des autres agents (**AgentManager**), les informations des cases (**MapManager**) et la carte sous forme de graphe permettant aux agents de se déplacer **MapRepresentation**.

2.2.1 Principe, forces et limites

La communication est au coeur de la FSM, chaque agent est toujours en écoute et vérifie sans cesse sa boite aux lettres pour voir si aucune request ne lui a été faite. La communication a été pensé de tel sorte a ne pas devoir envoyer des messages dans le vide. Pour cela, les agents doivent arrêter de se déplacer quand ils communiquent, regarder régulièrement leur boite aux lettres et envoyer des accusés de réception quand ils reçoivent des messages :

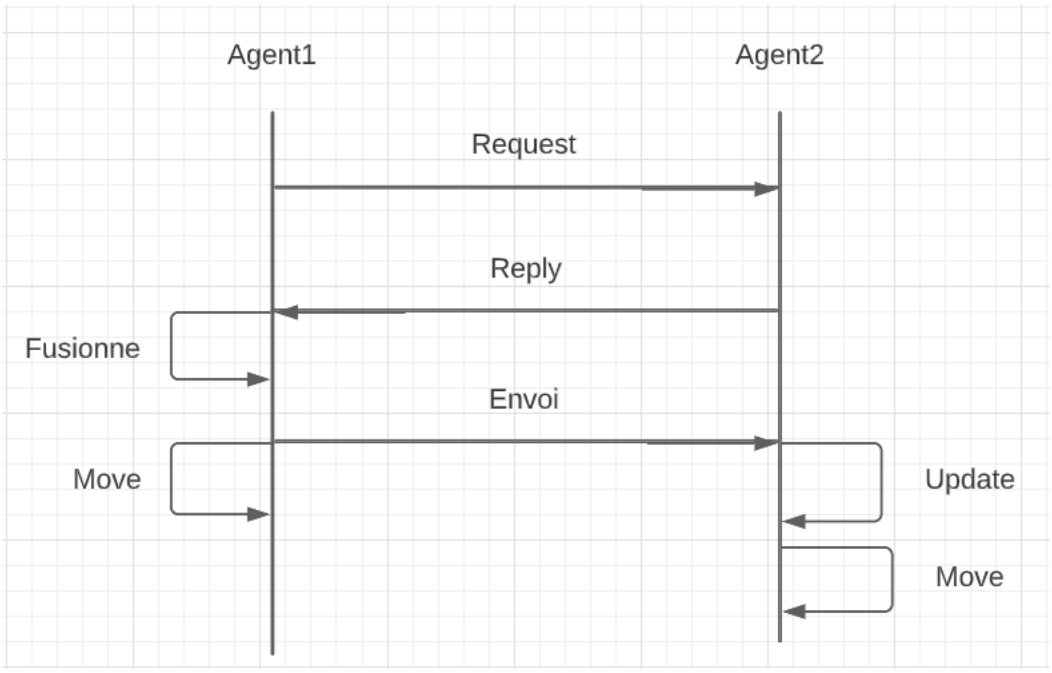

FIGURE 4 – Exemple typique d'une communication entre deux agents

Cette méthode nous permet d'éviter de propager sans cesse un paquet de message mais d'envoyer des messages que quand cela est utile.

La communication intervient lors de deux étapes :

- Tout d'abord, lors de l'exploration, chaque agent est initialement sur le comportement EcouteRequest, si il ne reçoit pas de request alors il peut en émettre un selon un rayon de communication limité et si un agent reçoit ce request il Reply

en lui partagent sa map ainsi que sa MapManager, l'agent responsable du request réceptionne et fusionne avec ses connaissances et renvoi ainsi la nouvelle **MapRepresentation** et le nouveau **MapManager**, l'autre agent la réceptionne et les deux se déplace d'une case dans une direction.

- Ensuite, lors des inter blocages, toujours dans le même principe, chaque agent est initialement sur le comportement EcouteRequest, si il ne reçoit pas de request mais que cette fois il est bloqué, c'est a dire que sa last_position est égale a sa current_position alors il peut en émettre un selon un rayon de communication limité et si un agent reçoit ce request il Reply en lui partagent son agent manager, l'agent responsable du request réceptionne et fusionne avec ses connaissances et renvoi ainsi le nouveau **AgentManager** ainsi que les informations qui en découle pour résoudre l'inter blocage, l'autre agent la réceptionne et les deux se déplace d'une case dans une direction.

2.2.2 Complexité (temps, mémoire, communication)

Dans un processus de communication de partage d'information concernant la carte, nous utilisons deux structures de données : une liste d'information (**MapManager**) pouvant prendre la forme de 6-uplets et un graphe $G = (V, E)$ de la carte. Chaque case de la carte comporte 2 informations de type chaine de caractère et 4 entiers. Pendant la communication, chaque agent doit transmettre une fois sa carte et il y a un agent qui est désigné pour faire la fusion et transmettre le résultat, ce qui fait 2 échanges et c la variable représentant les demandes request faite par chaque agent. De plus, en complément, une liste est envoyé contenant uniquement les m endroits d'intérêt (or ou diamant) pour éviter d'alourdir le réseau avec des informations inutiles. Cela nous donne les complexité spatiale suivante lors d'un partage d'information cartographique :

$$O(|V| + |E| + |V| * (2 * size(chaine) + 4 * size(int)) + c) \text{ en bytes en mémoire}$$

$$O(2(|V| + |E| + m * (2 * size(chaine) + 4 * size(int)))) \text{ en bytes sur le réseau}$$

Toutes les opérations effectuées pendant un échange d'information cartographique s'effectuent en $O(1)$. Néanmois, on peut retenir les temps t_1 et t_2 qui apparaissent lors de l'attente de réception respectivement de la carte et de la liste d'information. On retient une complexité temporelle pendant cette échange de :

$$O(t_1 + t_2) \text{ en temps d'attente}$$

2.2.3 Optimalité et critère d'arrêt

Lorsqu'une communication est engagée, les deux agents ne bouge plus comme on peut le voir dans le schéma de la FSM. Cela nous permet d'assurer que les agents

soient toujours à portée et qu'ils puissent se faire des retours. Nous garantissons alors que grâce à la séquentialité de la FSM les deux agents ont connaissance commune des informations topographique.

L'échange prend fin lors du dernier retour du **MapManager** ou de **AgentManager** ou par time out dans le cas où un des agents meurt.

2.3 Coordination

2.3.1 Principe, forces et limites

La coordination est 'l'ensemble des activités supplémentaires qu'il est nécessaire d'accomplir dans un environnement multi-agents et qu'un seul agent poursuivant les mêmes buts n'accomplirait pas'. Ce principe est principalement concerné lors des inter-blocages. Comment les agents gèrent-ils les inter-blocage ? Ici, nous allons utiliser une méthode se basant sur le comptage d'intersections, c'est a dire que chaque agent détient une variable qui est incrémenté a chaque fois que l'agent arrive a un point ou il n'y a que 2 observations possibles, a droite ou a gauche (un couloir). Lorsque 2 agents sont l'un face a l'autre dans un couloir avec des directions opposées, c'est celui avec le compteur le plus élevé qui l'emporte et donc l'agent avec le compteur le moins élevé devra changer de direction et reculer de 'compteur + 1' cases pour laisser passer l'autre agent. En cas d'égalité de compteur, c'est l'agent faisant la demande en premier qui l'emporte.

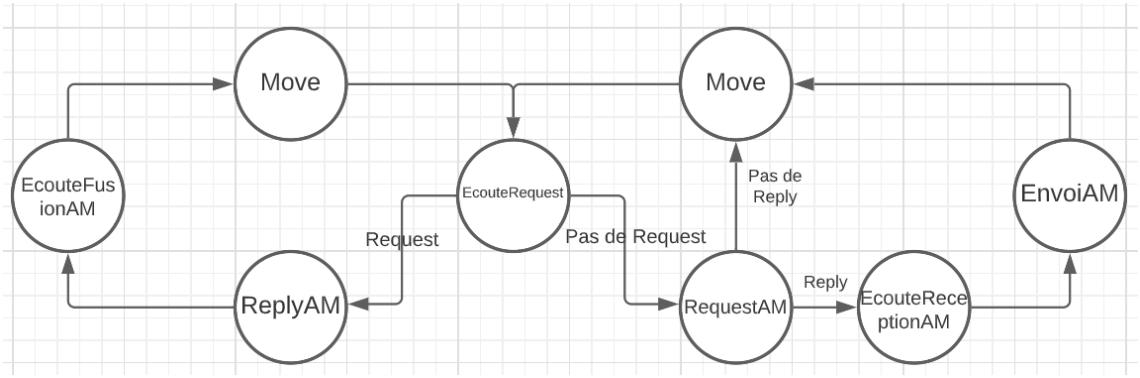

FIGURE 5 – FSM utilisé pour la coordination des inter-blocages

Ci-dessus la FSM (Final State Machine) correspondant a notre gestion des interblocages. Tout commence au comportement **EcouteRequest**, chaque agent va vérifier dans sa boite au lettre s'il na pas reçu de request. Sinon si l'agent se trouve dans un interblocage que nous avons définis par le fait que sa position courante est égale a sa derniere position, alors il émet une request de l'agent manager en passant au comportement **RequestAM**. Ci-dessous, la représentation de l'agent manager :

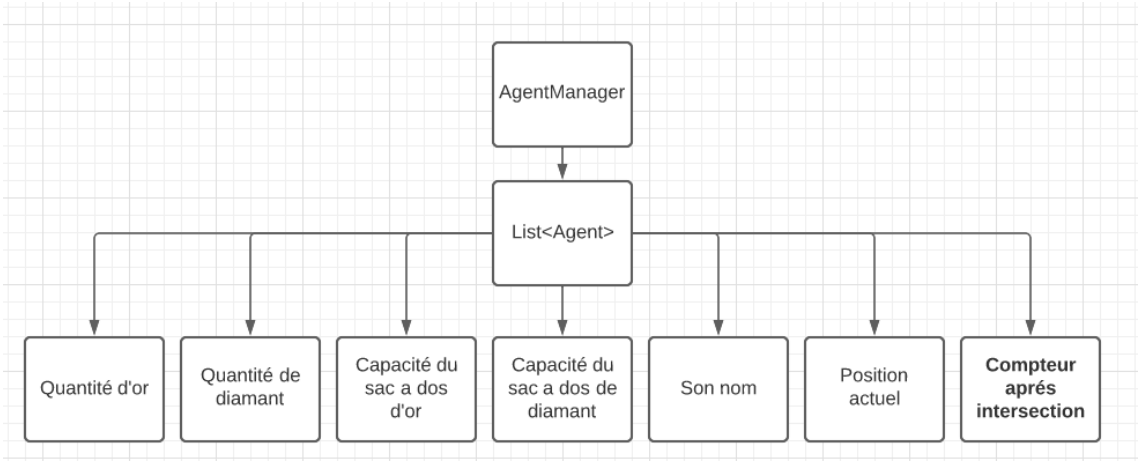

FIGURE 6 – AgentManager

L'agent qui reçoit le Request va donc passer au comportement **ReplyAM** et va envoyer son agent manager. L'agent a l'origine du request réceptionne ses informations et va surtout comparer leur **compteur après intersection** pour définir quel agent doit céder le passage. La question essentiel qui se pose est, comment incrémente on se compteur ?
Sur la figure ci-dessous on peut voir comment le compteur est incrémenté :

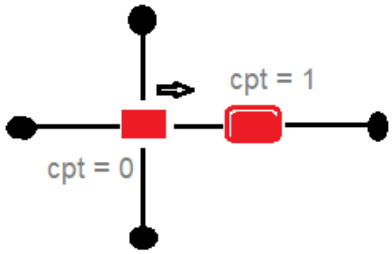

FIGURE 7 – Schématisation de l'incrémentation du compteur après intersection

Pour mieux comprendre la méthode, appliquons la a un exemple avec 2 agents, ici l'agent rouge et l'agent bleu. Comme on peut le voir, l'agent bleu a un compteur plus élevé donc l'agent rouge va recevoir l'information de reculer et de laisser passer l'agent bleu.

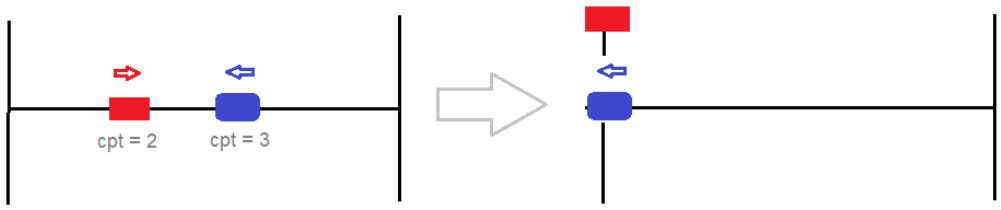

FIGURE 8 – Exemple avec 2 agents

Pour compliquer un peu les choses, prenons 3 agents, l'agent rouge avec un compteur de 2, l'agent orange avec un compteur de 1 et l'agent bleu avec un compteur de 3. L'agent orange et rouge ayant la même direction, opposé a celle de bleu. Tout d'abord, bleu et rouge vont être confronté et bleu va gagner donc rouge va changer sa direction pour le laisser passer, or orange les bloques, donc quand rouge va se confronter avec orange, il va gagner et ainsi orange va devoir changer sa direction. Enfin de compte, orange et rouge vont reculer afin de laisser passer bleu.

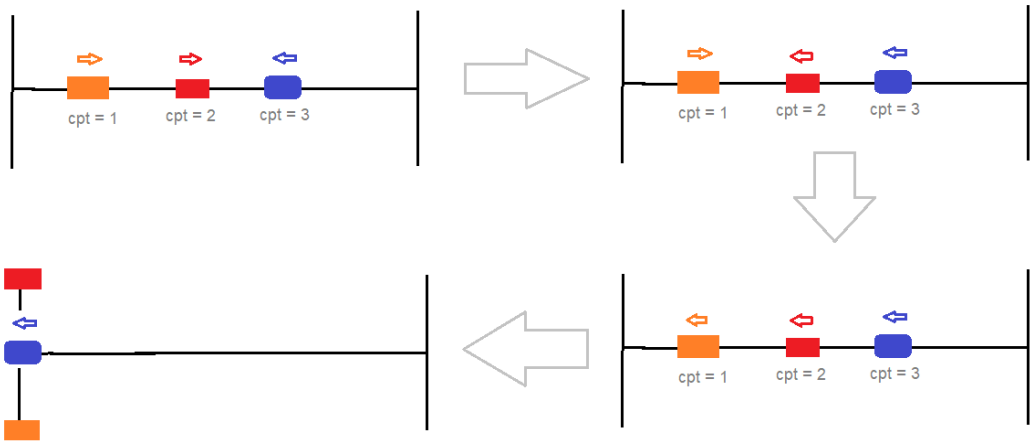

FIGURE 9 – Exemple avec 3 agents

2.3.2 Complexité (temps, mémoire, communication)

La décision des directions de chaque agents lors d'un interblocage se situe dans l'état **EcouteReceptionAM**, et tous les calculs et mise à jours de données se font en complexité temporelle $O(1)$. La résolution fait intervenir une répétition de 5 état pour l'agent qui fait une demande d'interblocage **(EcouteRequest-RequestAM-EcouteReceptionAM-EnvoiAM-Move)**. Cependant, un certain temps existe entre le passage de l'état **EcouteReceptionAM** à **EnvoiAM** car il s'agit de communication sur le réseau. On notera t le temps mis à recevoir le retour du message reçu à l'état **EcouteReceptionAM**. De plus, il y aura au plus $min(d_1, d_2)$ tour de

boucle de ces 5 état pour débloquer une situation d'interblocage, avec d_1 et d_2, les distances respectives des agents 1 et 2 depuis la dernière intersection. A noter que l'on utilise l'algorithme de Dijsktra en complexité O($a + nlogn$) avec a le nombre d'arête du graphe et n le nombre de noeud du graphe dans l'état Move. Cependant, nous considérons cette complexité négligeable car on souhaite juste atteindre une case de distance 1. On peut alors exprimer la complexité temporelle de la coordination de deux agents en situation d'interblocage :

$$O(t * min(d_1, d_2)) \text{ en temps d'attente}$$

Pour choisir l'agent qui va reculer et l'agent qui va avancer, chaque agent possède une liste d'information à propos des N autres agents. l'information essentielle ici est le nombre de pas depuis la dernière intersection ainsi qu'une liste d'identifiant des différentes positions de l'intersection. Cette liste prend la forme de n-uplets avec 1 entier et $n - 1$ chaines de caractères et c la variable représentant les demandes request faite par chaque agent. On obtient donc la complexité spatiale :

$$O(N(size(int) + (n-1)size(chaine) + c) \text{ en bytes sur le réseau}$$

$$O(N(size(int) + (n-1)size(chaine)) \text{ en bytes en mémoire}$$

Ceci serait l'approche idéal, cependant nous utilisons un format JSON pour le transfert des données sur le réseau ce qui augmente la complexité spatiale pour des raisons de facilité d'implémentation.

2.3.3 Optimalité et critère d'arrêt

Nous avons vu que lors d'un interblocage, les agents se coordonne afin de solutionner le problème. Pour cela, plusieurs échanges seront necessaire si les agents sont bloqué dans un couloir ou une impasse mais un échange résultera forcément avec un déplacement de l'agent plus proche de l'intersection. Cela nous garantit d'avoir un nombre de déplacements optimal.

L'échange est fini et la situation est débloquée lorsque l'agent faisant la demande d'interblocage n'a plus une position courante égale à sa dernière position.

2.4 Collecte

2.4.1 Principe, forces et limites

Dans cette partie, on se concentre a l'allocation et la collecte des trésors. Afin de satisfaire la contrainte d'équité tout en maximisant la récolte, il devient nécessaire de préciser ce que l'on entend par le terme d'équité. Qu'est-ce qu'un partage équitable ? La définition de l'équité selon le Dictionnaire de la Langue Française d'Émile Littré (1863) est :

« Disposition à faire à chacun part égale, à reconnaître impartialement le droit de chacun. »

La première propriété nécessaire pour l'équité est donc l'impartialité : sa caractéristique est de ne pas favoriser l'un au détriment de l'autre. Dans notre cas, on a donc décidé de donner la même quantité de ressources a chaque agent en faisant une moyenne du nombre de quantité présente sur l'environnement en fonction du nombre d'agent. La difficulté sera donc de choisir le type de ressource a récolter. Le principe est le suivant, tout d'abord on calcul la moyenne d'or et de diamant qu'on marginalise pour obtenir des probabilités. Ensuite, pour chaque agent on marginalise indépendamment ses capacités de stockages. Suite a cela, on multiplie le ratio du stockage du sac a dos par le ratio de la moyenne du même type, on marginalise et on choisis la classe prioritaire. Une fois le type choisis on se dirige vers les trésors avec une quantité inférieure a celle qui est censé être ramassé, c'est a dire la moyenne du type choisis.

Voici un tableau illustrant le calcul du choix de l'affectation d'un agent à une ressource qui est soit or, soit diamant.

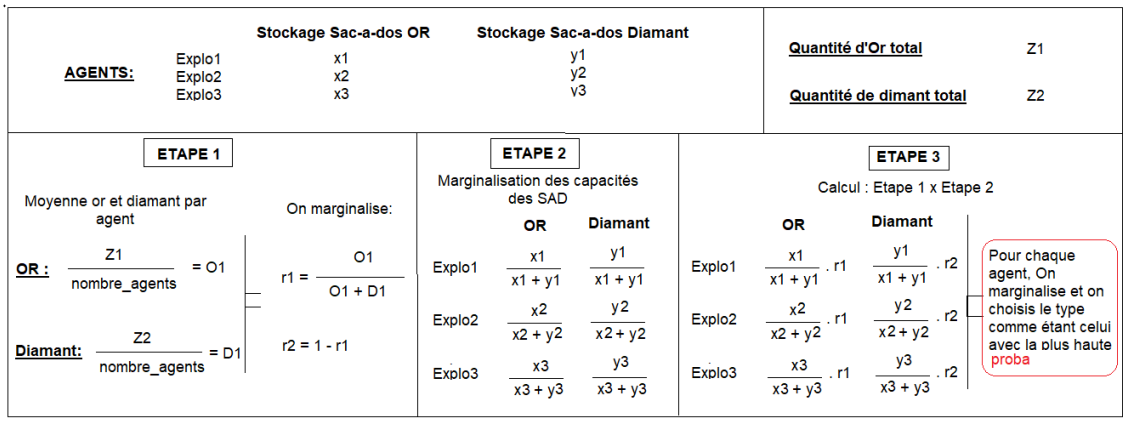

FIGURE 10 – Les différentes étapes du calcul pour la collecte

Pour déterminer si l'agent i doit être affecté aux ressources d'or ou de diamant, nous effectuons une comparaison grâce à nos ratios d'or et de diamant, ainsi l'affectation retenu correspond à un critère de classe majoritaire :

$$affectation = \begin{cases} or & \text{si } \frac{x_i}{x_i+y_i} * r_1 \leq \frac{x_i}{x_i+y_i} * r_2 \\ diamant & \text{sinon.} \end{cases}$$

Ensuite, lorsque l'agent i à reçu une affectation, il se rend dans tous les dépôts de ressource de la moins rempli à la plus rempli jusqu'à ce qu'il ne puisse plus ramasser à cause de sa capacité ou si il a atteint la moyenne.

Dans notre implémentation, nous prenons aussi en compte les mis a jour des cases, c'est a dire que si l'agent choisis diamant mais que toutes les cases de diamants ont été récupérés, l'agent va mettre a jour ses connaissances et recalculer son nouveau type et ainsi se diriger vers l'or. De même, pour les cases où les ressources ont été récupérés il va explorer du plus petit au plus grand jusqu'à obtenir la moyenne.

A noter que la phase de récolte débute lorsque l'agent connaît toute la représentation de la carte, c'est à dire qu'il n'y a plus aucun noeud ouvert ou qu'il y a encore des noeuds ouverts mais ces derniers sont bloqués par le wumpus (nous représenterons ces cases bloqués dans une liste) et tous les x pas de temps, l'agent effectuera une nouvelle visite.

2.4.2 Complexité (temps, mémoire, communication)

Cette méthode a été pensée pour éviter les communications, les zones parkings et que chacun agisses de manière dépendante par rapport aux autres afin de gagner du temps, d'éviter d'alourdir le réseau et par facilitée d'implémentation. Il n'y a donc pas de communication à faire et pas de complexité en mémoire sur le réseau. Tous les calculs se font en $O(1)$ en complexité temporelle dans l'état **CalculBestNode**. Ainsi, nous observons que nos agents convergent vers un type de ressource avec l'aide des informations qu'il reçoivent des autres agents et par déplacement.

Une fois l'affectation choisi, l'agent va se déplacer de dépôt en dépôt jusqu'à atteindre sa limite. On peut estimer cette complexité temporelle en fonction des états franchis par l'agent dans la FSM avec le nombre d'état à franchir dans cette boucle $= 3$ (**EcouteRequest-CalculBestNode-Move**) le nombre de dépôt à visiter N, le nombre de déplacement moyen m et une constante T pour le temps d'attente rajouté artificiellement afin d'éviter les problèmes de synchronisation avec l'API :

$$O(3 * NmT) = O(NmT) \text{ en nombre d'états à franchir}$$

On notera aussi que l'on utilise l'algorithme de Dijsktra pour se rendre de noeud en noeud en complexité $O(a + nlogn)$ avec a le nombre d'arête du graphe et n le nombre de noeud du graphe dans l'état **Move**.

On obtient donc une complexité temporelle polynomiale empirique finale pour la récolte de :

$$O(NmT * (a + nlogn)) \text{ en nombre d'opérations élémentaires}$$

Pour effectuer les calculs d'affectation et de direction, nous utilisons deux structures de données : un graphe et une liste de données. Cette liste de données comporte des objets (**Field**) qui peuvent s'apparenter à des 6-uplets : avec deux chaines de

caractères et 4 entiers.

Nous avons une complexité en mémoire de $O(a + n)$ pour le graphe. Ainsi qu'une complexité de $O(4(size(int)) + 2(size(chaine)))$ pour tous les K objets.

Nous obtenons donc une complexité en mémoire empirique totale de :

$$O(a + n + K(4(size(int) + 2(size(chaine)))) \text{ en bytes en mémoire}$$

2.4.3 Optimalité et critère d'arrêt

Étant donné que chaque agent cherche à récupérer une quantité équivalente à la moyenne de la quantité totale, on est assuré que le critère d'équité à la fin de la simulation est respecté, c'est à dire que chaque agent aura pour une même ressource une part à peu près égale de la quantité totale de cette ressource. Cela nous permet de respecter l'équité en prenant en compte la contrainte de ressource unique. En effet, on parcours chaque dépôt de la plus petite quantité à la plus grande quantité pour éviter qu'un agent ayant une très grande capacité de stockage récupère tous le dépôt d'un coup au lieu d'être limité par sa moyenne. De cette manière, nous prenons aussi en compte la contrainte que les agents récupèrent tout ce qu'il peuvent sur un dépôt.

De plus, les agents vont de dépôt en dépôt pour atteindre cette moyenne par agent. Donc avec cette méthode, on cherche aussi à maximisé le gain total obtenu. Enfin, les agents ayant récupéré leur moyenne ou si il n'y a plus de dépôt disponible repassent en mode exploration pour vérifier si le wumpus aurait déplacer des ressources, toujours dans un esprit de maximisation.

Le critère d'arrêt est atteint lorsque chaque agent à récupéré sa quantité moyenne de ressource, ou si il n'y a plus de dépôt disponible sur la carte.

3 Conclusion

La principale difficulté de ce projet a été de trouvé un calcul judicieux pour satisfaire la contrainte efficacité / équité et grâce a notre méthode basé sur la moyenne nos résultats sont assez robustes. De plus, nous avons essayé d'optimiser les différentes parties en commençant par la communication qui est au coeur de notre FSM en réduisant considérablement le nombre et la quantité de messages échangés entre les agents. Cette communication est très utile notamment pour l'exploration car par un rayon de communication limité certains agents peuvent s'échanger plusieurs informations concernant la map ou encore les informations propre a chacun et cela leur permet d'éviter de parcourir des noeuds et d'accélérer l'exploration. Mais encore, la communication est indispensable lors d'un inter blocage entre agents afin de coordonner suite a la méthode du compteur après intersection les directions de chacun.

Quelques améliorations auraient pu être faites, comme par exemple dans l'exploration il peut arriver qu'après communication entre deux agents, ils se suivent pour aller vers le même noeud ouvert ou cela, par manque de temps aurait pu être changées en indiquant à chacun un noeud différent. De plus, en ce qui concerne la communication, nous avons utilisé JSON pour faciliter le débuggage mais aussi la compréhension de ce qu'on obtenait ou cela peut avoir des impacts importants sur la complexité spatiale du réseau et donc cela aurait pu être amélioré en le changeant par des objets sérialisables. Toujours dans la communication, l'idéal aurait été de ne pas vérifier sa boite aux lettres ni d'envoyer de request trop souvent pour limité le nombre de messages échangés. Enfin pour la collecte, différentes méthodes ont été réfléchis comme par exemple l'utilisation d'une zone parking pour établir une décision collective en réunissant les agents a un même endroit et d'utiliser un agent comme leader qui réceptionne toutes les informations sur les autres agents et coordonne le ramassage des ressources par une résolution de problème d'affectation.

4 Remerciements

Nous tenons a remercier et exprimer notre profonde gratitude au différents professeurs qui nous ont aidé et guidé pour la réalisation de ce projet et notamment Monsieur Cédric Herpson pour le temps qu'il nous a consacré et pour les précieuses informations qu'il nous as prodiguées avec intérêts et compréhension.