Faculté des Sciences et Ingénierie
Sorbonne Université

Projet - Wumpus Multi-agent

Michaël Trazzi, Michaël Aidan
Sous la supervision d'Aurélie Beynier, Nicolas Maudet et Cedric Herpson

FOSYMA, Groupe 1
Année 2017 – 2018, Semestre 2

Table des matières

1 Introduction

Pour ce projet nous avons été amenés a implémenter un Système Multi-Agent pour résoudre un problème en apparence simple : la collecte de trésors sur un graphe. Cependant, des contraintes sur la perception des agents et la présence d'un terrifiant « Wumpus » sur la carte ont grandement complexifié le travail algorithmique.

1.1 Ressources

Notre implémentation est disponible sur `github` a l'adresse suivante : `https://github.com/mtrazzi/Hunt-The-Wumpus`. De plus, nous avons rédigé un article, publié en ligne, expliquant en détail les difficultés liées aux Systèmes Multi-agents, et en particulier pourquoi implémenter des protocoles Multi-Agents pouvait s'avérer bien plus complexe que simplement faire de l'apprentissage machine supervisé. Cet article de vulgarisation est hébergé sur `Medium` : `https://hackernoon.com/why-coding-multi-agent-systems-is-hard-2064e93e29bb`.

1.2 Présentation du problème

Quatre types d'agents sont placés sur les noeuds d'un graphe : les Explorateurs, pouvant uniquement se déplacer, les Collecteurs, capables de ramasser des trésors de leur type jusqu'à remplir leur sac, les agents Silo, pouvant accumuler sans limite les trésors ramassés par les agents Collecteurs, et l'agent « Wumpus », se mouvant aléatoirement sur la carte en déplaçant les trésors.

Les seuls agents dont nous pouvons modifier le comportement sont les agents Explorateurs, Collecteurs et Silo. Ces agents vivent dans un environnement `JADE`, environnement de développement pour des systèmes multi-agents en `Java`. Ils exécutent des comportements (ou *behavior*) séquentiellement, dans un ordre pré-déterminé, a chaque fois qu'un processus les réveille.

En Intelligence Artificielle, un agent est n'importe quel objet percevant son environnement par des *senseurs*, et agissant sur le monde a travers des *effecteurs*. Le Wumpus n'ayant pas de comportement complexe (il ne prend pas de décision mais se contente de faire des mouvements aléatoires), nous désignerons par *agent* les trois autres agents (Explorateurs, Collecteurs et Silo). Ces trois agents (qu'on a été amené a implémenter) possédaient des senseurs et effecteurs relativement précaires.

- **Senseurs**
  - **Observation de l'environnement :** Chaque agent peut exécuter la méthode `observe`, qui retourne une liste d'attributs de noeuds adjacents au noeud où il se trouve. Il découvre alors quels noeuds lui sont adjacent, si le noeud sur lequel il se trouve contient des trésors, et leurs types dans le cas où il y en aurait. Un autre attribut perçu par les agents est l'odeur du Wumpus. Effectuer `observe` n'apporte pas *a priori* de connaissances sur la présence d'un autre agent sur un de ces noeuds (autre que l'odeur d'un Wumpus).
  - **Réception de message :** Les agent ont des boîtes au lettre, et peuvent recevoir des messages d'autres agents, sous la forme d'objets `Java` sérialisables.
- **Effecteurs**
  - **Mouvement :** Possibilité de lancer une action « bouger sur case X », qui résultera sur un déplacement sur la case X si celle-ci est libre et adjacente.
  - **Collecte de trésor (seulement Collecteur) :** Possibilité de lancer une action « ramasser », qui collectera effectivement un trésor sur la case actuelle si celle-ci contient un trésor, l'agent Collecteur n'a pas son sac rempli et si le type du trésor correspond au type de trésor que peut ramasser le Collecteur.

— **Transmission de trésor (seulement Collecteur) :** Possibilité de vider son sac a dos et transmettre son contenu a un agent Silo si celui-ci se trouve sur une case adjacente.

— **Émission de message :** Transmission d'un message (objet `Java` sérialisable) a l'ensemble des destinataires étant a une distance $d < \rho_{max}$.

1.3 Problématiques

Au vu des limitations perceptives et effectives mentionnées ci-dessus, un certain nombre de problématiques se posent. Ces difficultés seront rapidement soulevées ci-dessous, puis nous reviendrons plus en détails sur ces problèmes et leurs résolutions dans la partie « Analyse des algorithmes ».

— **Exploration de l'environnement** : un agent n'observe que la présence/quantité de trésor relative au noeud où il se trouve. Afin d'avoir une vision globale ils doivent mettre en commun leurs graphes individuels. Se pose la question du partage d'information changeante (quantité de trésor diminuant, trésors se mouvant à cause du Wumpus) et d'horodatage de l'information.

— **Perception limitée** : les agents ne savent pas si un autre agent se trouve en face, ou pire, si le terrible Wumpus s'approche à grands pas. D'où le problème de l'interblocage.

— **Organisation et Coalitions** : les agents doivent planifier leurs actions de sorte à intégrer la complémentarité de leurs rôles. Ils doivent se coordonner de sorte à ce qu'un Collecteur puisse toujours donner ses trésors à un agent Silo, et que les Explorateurs informent en permanence les Collecteurs de l'état de la carte.

Voyons désormais les solutions que nous avons apporté à ces problématiques.

2 Analyse des algorithmes

Nous avons implémenté, par itérations successives, une série d'algorithmes permettant de résoudre en partie les problématiques mentionnées ci-dessous. Pour chaque domaine d'application d'algorithme (Exploration, Communication, Interblocages, Ramassage de Trésors/Coordination) nous présenterons leur principe général, étudierons leurs forces/limites, puis discuterons de leurs complexité.

2.1 Structure de donnée utilisée pour les algorithmes

Nous avons défini une classe `Graph<String>` (voir `mas.utils.Graph` dans le dossier `projet/src`) où chaque chaîne de caractère correspond à un noeud. Ce graphe contient :

— Une liste d'adjacence `adjacencyList` de type `HashMap<String, Set<String>>` définissant la structure du graphe.

— Une hashmap `isVisitedHashmap` de type `HashMap<String, String>` définissant si un noeud a été visité ou pas (on marque « discovered » un noeud visité, on ne le marque pas sinon).

— Une hashmap `treasureHashmap` de type `HashMap<String, MyCouple>` qui à chaque noeud associe un couple d'entiers (type « MyCouple ») définissant respectivement la quantité de trésors et de diamants sur le noeud.

— Une hashmap `timeStampHashmap` associant à chaque noeud la dernière date (en milliseconde, donnée par la machine) où le noeud a été mis à jour.

De plus, chaque agent possède une pile de chaîne de caractères (c'est-à-dire une `Stack<String>`) correspondant à une liste de mouvements à effectuer (c'est l'attribut `stack` de `GeneralAgent`, une classe agent générale contenant les méthodes utilisées par tout type d'agents). Si par exemple l'agent décide d'aller tout en haut à gauche, il calcule le chemin le plus court vers ce noeud, empile l'ensemble des noeuds où il va devoir aller sur sa pile (Stack), puis lors des prochains tours il va successivement dépiler sa pile pour obtenir le prochain mouvement à effectuer, jusqu'à arriver à destination.

2.2 Exploration

2.2.1 Principe

L'exploration de la carte est réalisée essentiellement par les agents Explorer, qui se déplacent en parallèle en cherchant à explorer des noeuds non visités à partir de ce qu'ils ont déjà découvert et des informations transmises par les autres agents. Chaque agent Explorer reçoit et transmet aux autres agents des classes de type `Graph<String>` contenant les 4 hashmaps (cf paragraphe `2.1` ci-dessus). Une fois que tous les noeuds ont été visités, le rôle des explorateurs va être de se diriger vers les trésors afin de communiquer des informations récentes concernant les trésors aux Collecteurs. De façon similaire, quand le Collecteur n'a plus de trésors en vue (dans sa représentation interne de l'environnement) il va adopter un comportement d'exploration afin de découvrir de nouveaux trésors (par exemple au début).

Afin de déterminer quel noeud non-visité (i.e. noeud ouvert mais non fermé dans un parcours de graphe) explorer en priorité, nous avons décidé d'utiliser un algorithme glouton consistant à prendre le noeud ouvert le plus proche (on obtient ce « noeud le plus proche non-visité » par un parcours en largeur (BFS)). Au début, nous avions essayé de parcourir tout le graphe par un algorithme de parcours en profondeur. Cela fonctionnait bien... avec un seul agent. L'algorithme se généralisait très mal pour des déplacements d'un point vers un trésor, et n'était pas assez flexible pour les interblocages.

2.2.2 Avantages/Inconvénients

- **Forces :** Cette exploration est peu coûteuse en temps de calcul (algorithme glouton prenant le noeud non-visité le plus proche).
- **Limites :** Le principal inconvénient de cet algorithme est la faible coordination entre les différents explorateurs. En effet, si deux agents sont voisins, et sont proches d'un même noeud non-visité, ils vont se diriger vers ce même noeud, au lieu de se répartir le travail (par exemple en laissant un seul y aller tandis que l'autre va voir un autre noeud plus loin).

2.2.3 Complexité

- **Critère d'arrêt :** La première phase d'exploration (découvrir la structure du graphe) s'arrête quand tous les noeuds ont été visités (i.e. il ne reste aucun noeud ouvert quand les agents mettent en commun leurs cartes). La deuxième phase d'exploration (afin de vérifier que les trésors sont toujours présents et mettre à jour leurs valeurs) se termine quand les agents ne perçoivent plus de trésors sur la carte.
- **Temps :** les calculs de noeud non-visité le plus proche (ou trésor le plus proche) sont de même complexité qu'un calcul de plus court chemin dans un graphe $G = (V, E)$, c'est-à-dire $\mathcal{O}(|V| + |E|)$ (notre structure de graphe utilise une liste d'adjacence).
- **Mémoire :** En mémoire l'exploration a une complexité en $\mathcal{O}(4.|V| + |E|) = \mathcal{O}(|V| + |E|)$ également. En effet, dans notre classe `Graph<String>` on stoque une liste d'adjacence ($\mathcal{O}(|V| + |E|)$ en mémoire) et trois hashmaps qui ont au plus autant de clés que le nombre de noeuds du graphe $|V|$.

— **Communication :** A chaque itération (où toute les behaviors d'un agent sont exécutées), chaque agent explorateur broadcast à tous les autres agents sa structure `Graph<String>`. Ils communiquent toute leur carte (donc l'information totale) et non pas seulement l'information sur la carte que l'autre agent ne possède pas (comme par exemple pour le problème du gossip en théorie de l'information). Il y a donc un seul tour nécessaire en théorie (envoyer ou bien recevoir). En pratique, chacun des agents envoie et reçoit des graphes tant qu'il est dans la zone d'un autre agent, donc le nombre de tours est variable. La quantité maximale d'information transmise est la taille de la structure `Graph<String>`, soit $\mathcal{O}(|V| + |E|)$.

— **Optimalité :** Cet exploration n'est pas optimale. Il suffit pour cela de considérer le cas décrit dans la partie **Limites** où deux agents se dirigent vers un même noeud non-visité proche au lieu de se séparer.

2.3 Communication

2.3.1 Principe

Dans cette partie nous allons décrire plus particulièrement les protocoles de communication mentionnés précédemment dans la sous-partie **Exploration**. Lors du déroulement du programme, tous les agents effectuent des « broadcast » et reçoivent des messages en permanence : ils se partagent l'ensemble des informations concernant la carte afin que chaque agent ait toutes les cartes en main (c'est le cas de le dire) pour prendre la meilleure décision (ici, la meilleure direction par exemple). Les données envoyées (en plus de la structure (noeuds et arêtes) du graphe) sont la liste des noeuds visités, l'emplacement des trésors et l'horodatage (avec l'horloge système) relatif à la dernière mise-à-jour de chaque noeud.

Les agents envoient leur informations aux « pages jaunes » : le Directory Facilitator (DF) (spécifié par FIPA). Ce dernier contient les informations concernant les agents du graphe et peut donc rediriger les données reçues. Ces données sont envoyées sous forme d'objets `Java` sérialisables, et sont retransformées en objet `Java` à la réception. L'agent qui reçoit doit fusionner l'information qu'il avait (sa structure `Graph<String>`) avec le graphe reçu (de type `Graph<String>` également). Pour cela il va ajouter les noeuds et arêtes du graphe reçu à son propre graphe, et, concernant les trésors, il va prendre l'information la plus récente à chaque fois (à l'aide des hashmaps `timeStampHashmap` donnant les dates de dernière mise-à-jour pour chacun des noeuds (il y a une `timeStampHashmap` pour le graphe présent initialement, et une pour le graphe reçu)).

2.3.2 Avantages/Inconvénients

— **Forces :** Les agents s'échangent de nombreuses informations et peuvent donc avoir une vue précise de l'ensemble du graphe.

— **Limites :** Dans notre situation, les agents s'échangent beaucoup d'information, ce qui crée une redondance dans les transferts d'information. En outre, l'horodatage est géré avec l'utilisation de l'horloge du système, donc la solution proposée ne se généralise pas à un système distribué.

2.3.3 Complexité

— **Critère d'arrêt :** La communication entre les agents ne s'arrête jamais.

— **Temps :** La complexité temporelle correspond à l'envoi des données vers le DF, auquel s'ajoute la retransmission de celles-ci vers les différents agents et la traduction des informations reçues. En considérant qu'il faut au moins parcourir la taille du message pour envoyer et recevoir des messages, la complexité est en $\mathcal{O}(N_{agent}(|V| + |E|))$ pour chaque broadcast, où N_{agent} désigne le nombre d'agents sur la carte.

— **Mémoire :** La place en mémoire occupé par cet algorithme est faible. En effet, il y a juste la présence d'un Graphe en plus en mémoire lors du déroulement de l'algorithme. La taille du Graphe présent initialement est d'avantage lié à l'exploration qu'à la communication. Pour la communication il y a seulement un graphe supplémentaire, de même taille, en mémoire non permanente, qui n'existe que pendant le processus de fusion de graphe, et qui est supprimé ensuite (si on considère qu'il y a N_{agent} agents qui reçoivent le graphe, cela fait N_{agent} graphes supplémentaires).

— **Communication :** Les protocoles de communication sont encore ici en un seul tour (il n'y a pas d'attente de réception de message). La taille du message le plus important est en $\mathcal{O}(|V| + |E|)$ encore une fois.

— **Optimalité :** Cette méthode n'est pas optimale d'un point de vue échange d'information. En effet, les agents s'envoient l'ensemble des informations concernant la carte à chaque message. Un algorithme « optimal » devrait permettre à un agent de se rappeler avec qui il a déjà échangé, afin de ne lui envoyer que les informations récentes qu'il a découvert (par exemple que les trésors découverts depuis leur dernière rencontre). On pourrait imaginer ainsi un système de versionnage de fichier « carte ». De plus, si on considère que chaque position de trésor constitue un « secret », le nombre de messages optimal pour que chacun dispose de tous les secrets atomiques est de $2N_{tresor} - 4$, alors que dans notre cas les agents ne s'arrêtent jamais de s'envoyer des messages !

2.4 Interblocages

2.4.1 Principe

Les algorithmes liés à l'interblocage implémentent des solutions à un problème majeur lors du développement logiciel : le blocage mutuel possible entre les agents lors de leurs déplacements dans le graphe. La perception des agents étant limitée, ils ne perçoivent même pas la présence d'un autre agent sur une case voisine (autre que l'odeur du Wumpus). Un agent remarque qu'il subit un interblocage en comparant le noeud où il devrait arriver au noeud où il se trouve, puis en remarquant que c'est le même, et donc qu'il n'a pas bougé car un autre agent était présent là où il voulait aller (nous avons implémenté un attribut `lastMove` afin que l'agent possède une mémoire de ses mouvements). Le protocole permettant de débloquer un agent d'un interblocage est le suivant :

— l'agent détermine le chemin le plus court vers un noeud de degré permettant un déblocage (nous avons pris « noeud de degré supérieur à 3 » comme condition). Pour déterminer ce chemin le plus court, un parcours en largeur est (encore une fois) effectué. A ce moment là, il empile sur sa pile de mouvements à effectuer les multiples noeuds le séparant de ce noeud de grand degré.

— si ce noeud est trop proche de lui (en face par exemple, ou bien c'est le noeud sur lequel il se trouve) ou s'il ne trouve pas de tel noeud (de degré supérieur à 3) alors il fait un mouvement aléatoire parmi les noeuds qui ne sont pas son dernier coup ni sa position actuelle.

— Finalement, au bout de 5 interblocages (chaque agent a un compteur d'interblocage) un agent se voit assigner 4 mouvements aléatoires. Ces paramètres sont empiriques et permettent de bien débloquer des situations tordues.

2.4.2 Avantages/Inconvénients

— **Forces :** La solution proposée pour débloquer les interblocages permet de débloquer un très grand nombre de cas, même tordus, avec des règles simples. Il utilise de l'aléatoire pour débloquer dans le cas général, mais c'est un aléatoire contrôlé (seulement 4 mouvements aléatoires) à des moments précis où les techniques précédentes ont échoué. Il n'y a pas d'attente d'une réponse d'autrui, de cas pathologiques où le récepteur sort de notre rayon de communication, etc. En outre, notre algorithme permet de gérer le Wumpus comme un agent comme

les autres, car il ne fait pas appel à de la communication (autre que l'échange de cartes entre agents non-Wumpus, ce qui se fait en continu de toute façon)

— **Limites :** Encore une fois, les agents ne peuvent pas se coordonner. Pour deux agents, des heuristiques de ce type dans un couloir fonctionnent toujours. Dans des cas pratiques, avec 5 agents dans un couloir, il faut réfléchir à d'autres protocoles où il y aurait un vrai consensus, de la communication/coordination autour de quel chemin emprunter, comment se séparer, etc. De plus, vu qu'il n'y a pas de communication à proprement parler, il n'y a pas de protocoles du type « pousse-toi j'ai beaucoup de trésors dans mon sac », ce qui pose problème.

2.4.3 Complexité

— **Critère d'arrêt :** L'interblocage se termine lorsque l'agent est arrivé à son noeud de degré supérieur à 3, c'est-à-dire son noeud de destination. S'il n'y parvient pas, le protocole se termine après 5 interblocages et 4 mouvements aléatoires, ou bien seulement un mouvement aléatoire.

— **Temps :** Encore une fois, on cherche un noeud vérifiant une certaine propriété avec un parcours en largeur. Ici, vu que la condition porte sur un degré, on sait que l'algorithme de parcours en largeur prendra $\mathcal{O}(|V|)$ itérations, car dans le pire des cas tous les noeuds sont de degré 2, jusqu'à arriver au noeud final de degré 3, ce qui correspond à un parcours linéaire en nombre de noeuds.

— **Mémoire :** La place en mémoire occupé est celle de l'algorithme de parcours en largeur, c'est-à-dire la mémoire nécessaire au processus de marquage, donc $\mathcal{O}(|V|)$.

— **Communication :** Ici, pas de communications qu'on n'aurait pas mentionné auparavant. Il est cependant important de remarquer que ce qui permet le déblocage dans des couloirs ou cul de sac, est le fait que les agents s'envoient leurs cartes en permanence, ce qui fait que quand ils se retrouvent face à face, les deux mettent automatiquement leurs cartes en commun, ce qui permet au deux de se diriger vers la sortie du tunnel.

— **Optimalité :** Ce protocole n'est pas optimal car il ne gère pas des structures trop complexes, par exemple des arbres binaires, où pour se débloquer plusieurs agents devraient se coordonner réellement en escaladant ensemble vers la racine de l'arbre.

2.5 Ramassage des Trésors / Coordination

2.5.1 Principe

Le ramassage des trésors repose sur un algorithme glouton consistant à toujours ramasser des trésors lorsqu'on est sur un trésor de son type et qu'on a de la place dans notre sac, mais surtout à se diriger vers les trésors de son type les plus proches en priorité, sans chercher à optimiser son sac à dos. Finalement, il s'agit de revenir vers l'agent Silo pour (littéralement) vider son sac dès qu'on ne trouve plus de trésors de taille suffisamment petite dans notre entourage, en supposant qu'on sache où il se trouve. Si tous les trésors sont récupérés, et que l'agent Collecteur n'a rien à donner à l'agent Silo (ou s'il ne sait pas où ce dernier se trouve), alors l'agent Collecteur se comporte comme un agent Explorateur

Les autres agents ont bien entendu leur rôle à jouer dans cette histoire, mais l'un d'entre eux est moins actif : l'agent Silo. En effet, nous avons décidé de le laisser immobile, condamné à rester sur sa position initiale. Tout ce que fait l'agent Silo est communiquer sa position aux autres agents par broadcast. Cette position est codée dans les messages par la présence d'un trésor négatif en diamant et trésor à la position du Silo. Ainsi, les agents se communiquent une position fixe du Silo, ce qui facilite grandement les trajets des Collecteurs qui savent précisément où aller pour vider leurs sacs-à-dos.

On finira par remarquer que par la spécification mentionnée jusqu'à présent, les agents (Explorateurs et Collecteurs) vont adopter un comportement complémentaire tout au long du programme. Au début, les Explorateurs découvrent rapidement la carte, ce qui permet aux Collecteurs de trouver le Silo. Quand le Silo est trouvé, et la carte entièrement découverte, les Explorateurs se mettent à se déplacer vers les trésors afin de pouvoir communiquer des informations récentes aux Collecteurs.

2.5.2 Avantages/Inconvénients

— **Forces :** Notre algorithme résout au mieux le problème de l'information imparfaite. En effet, vu que le Wumpus peut déplacer les trésors, on ne veut pas que l'agent Collecteur exécute des calculs savants d'optimisation de sac-à-dos en décidant d'en aller en chercher un à l'autre bout de la carte, alors que ces trésors lointains auraient une grande probabilité d'être déplacés entre-temps. L'agent Collecteur se contente de l'approche pragmatique consistant à prendre un trésor proche (sous réserve d'existence), ce qui s'avère efficace dans un contexte où les trésors peuvent être déplacés.

— **Limites :** Il n'y a pas de Coordination entre les agents, ni même d'optimisation du sac-à-dos d'un unique agent. Par exemple, s'il existe un trésor de valeur 10 proche de l'agent (qui a une capacité de 100) et un trésor plus loin de valeur 100, l'agent va récupérer le trésor le plus proche (mais avec la valeur la moins importante), et il ne pourra pas récupérer l'autre trésor sans rendre visite au préalable à l'agent Silo.

2.5.3 Complexité

— **Critère d'arrêt :** S'il n'y a pas de critère d'arrêt pour le processus de ramassage de trésors en général, l'agent Collecteur possède comme critère d'arrêt pour la collecte de ne plus trouver de trésors suffisamment petit à mettre dans son sac (il n'en voit plus sur sa carte), et il se dirige alors vers l'agent Silo. Il arrête également de se comporter en tant que Collecteur quand son sac est vide et qu'il ne voit plus de trésors. A ce moment là, il se comporte comme un simple Explorateur.

— **Temps :** Encore une fois, pour déterminer la position des trésors on effectue un parcours en largeur de complexité $\mathcal{O}(|V| + |E|)$.

— **Mémoire :** La place en mémoire correspond à celle occupée par la place des trésors en mémoire et aux processus de marquage lors du parcours en largeur, donc de complexité $\mathcal{O}(|V| + |E|)$.

— **Communication :** Tout les agents s'envoient leur graphe et réactualisent leur carte à chaque fois. La complexité ne change pas de ce point de vue par rapport aux autres parties. On remarquera cependant qu'on a incorporé la position du Tanker dans la carte des trésors, et que le fait que ce Tanker ne bouge pas permet de ne pas avoir à être incertain au niveau de l'information et de la communication.

— **Optimalité :** Cet algorithme n'est bien entendu pas optimal car la coordination entre les agents est limitée et le travail d'optimisation dans le remplissage du sac-à-dos n'est pas réalisé. On remarquera néanmoins que le problème du sac-à-dos étant NP-complet, une résolution optimale du problème du multi-sac-à-dos multi-agents en environnement incertain l'est aussi.

3 Conclusion

3.1 Synthèse

Nous avons résolu le problème du ramassage de trésors multi-agents en environnement incertain en faisant un certain nombre de choix. Le premier a été de garder un agent Silo fixe. Le second, d'échanger en continu par broadcast les données relatives à l'ensemble de la carte dans une grande structure de graphe contenant plusieurs hashmaps.

Les agents explorateurs commencent par se lancer à la découverte de la carte en choisissant les noeuds non-visités les plus proches. Quand tous les noeuds sont visités, ils se déplacent vers les trésors afin de garder des informations récentes sur les trésors (pour lutter contre le Wumpus déplaçant les trésors). Les collecteurs, quant à eux, ramassent les trésors les plus proches et les ramènent à l'agent Silo. Cette approche mêle plusieurs heuristiques et résout le problème de façon simple en n'utilisant pas d'algorithmes bien plus complexes que des parcours en largeur, mais permet néanmoins de trouver une solution au problème posé initialement.

3.2 Regard critique sur notre travail

Le problème majeur de notre implémentation est la complexité de communication qui est maximale. En effet, les agents communiquent en continu tout ce qu'ils savent en broadcast à tout le monde. Cela consiste en une structure `Java` contenant une liste d'adjacence (pour le graphe) et trois autres hashmaps de taille équivalente au nombre de noeuds du graphe. Cela ne constituerait pas un problème en soi si les agents ne lançaient pas de messages en continu avec toute la carte, mais faisaient plutôt des `ping` pour se mettre en interaction, puis commençaient à envoyer des messages lorsqu'ils auraient la certitude d'être à proximité d'un agent. D'autant plus que si deux agents restent dans un rayon de communication suffisant pendant un certain temps, ils vont s'envoyer des quantités immenses d'information tout à fait redondante. De plus, l'horodatage de l'information par l'horloge système n'est pas suffisante dans le cas où le système est distribué.

Le deuxième problème le plus important est celui de la coordination. Trop souvent, lors de l'exploration ou du ramassage des trésors, les agents ne vont pas se séparer le travail et peuvent se ruer vers un même noeud ou un même trésor.

Finalement, il reste des problèmes plus difficiles à résoudre, mais néanmoins importants, qui sont ceux de l'optimisation dans le ramassage des trésors et dans l'exploration. On pourra par exemple réfléchir à des calculs de chemins optimaux permettant de prendre en compte les capacités successives des agents collecteurs, afin de minimiser la distance parcourue en maximisant la quantité de trésor ramassée.

3.3 Extensions et améliorations possibles

Pour le premier problème énoncé, un protocole de mise en interaction (type `ping`) est à envisager. Pour limiter les échanges d'information, on pourra penser à un système de versionnage de fichier (type `git`) qui permettrait de savoir exactement quels ont été les noeuds ajoutés, ou les noeuds modifiés. Il faudra aussi remplacer l'horloge système par un système d'horloge vectorielle, afin de pouvoir déployer la solution sur un système distribué.

Pour la coordination, on pourrait placer le Tanker au « centre du Graphe », où cette centralité serait, par exemple, le noeud par lequel passe le plus de chemins géodésiques.