Projet de FOSYMA
Wumpus Multi-agent

B.Thanh Luong, Gualtiero Mottola

Groupe 2

Table des matières

1 Introduction **2**

2 Présentation des Agents **2**
2.1 Comportement des Agents 2

3 Communication et interblocage **3**
3.1 Processus de Communication 3
3.2 Les Outils 5
3.3 Interblocage 6

4 Conclusion **6**

1. Introduction

Ce projet consiste à développer une version multi-agent d'un jeu fortement inspiré de "Hunt the Wumpus" cette variante du jeu est définie de la façon suivante : un ensemble d'agents en coopération sont placés dans un environnement inconnu. Ils ont pour mission d'explorer et de récupérer un maximum de trésors qui sont disséminés dans cet environnement. Un agent Wumpus est également présent, il se déplace aléatoirement et a pour but de perturber l'exploration et la récupération des trésors.

2. Présentation des Agents

Les trois types d'agents utilisables pour récolter un maximum de trésors sur la carte sont les suivants : les Agents Explorateurs qui n'ont pas la possibilité de récupérer des ressources, leur seul but est d'explorer la carte. Des Agents Collecteurs qui ont un sac à dos correspondant chacun à un type de trésor (`TREASURE` ou `DIAMONDS`) et qui possèdent une méthode permettant de récupérer ce type de trésor et de le placer dans leur sac si celui-ci n'est pas plein. Lorsque cette action est exécutée, une partie du trésor est perdue. Enfin le dernier type d'agent, l'agent Tanker qui ne peut pas ramasser de trésor mais qui a un sac à dos de capacité illimitée. Tous les agents collecteurs ont la possibilité de donner leurs trésors à l'agent Tanker. Ce sont les quantités présentes dans l'agent Tanker qui seront comptabilisées à la fin de l'exécution.

2.1. Comportement des Agents

Les comportements de nos trois types d'agents sont tous implémentés sous la forme de `FSMBehaviours` qui sont des Automates finis. La classe offre des méthodes qui permettent d'enregistrer les états et les transitions, ce sont ces dernières qui définissent l'ordre dans lequel les états s'exécutent. Chaque état de l'automate fini est un object qui étend la classe `behaviour` et qui sera exécuté selon l'ordre définit par l'utilisateur.

Voici un exemple du code de l'automate fini de notre agent explorateur.

```
1  FSMBehaviour fsmBehaviour = new FSMBehaviour();
2  fsmBehaviour.registerFirstState(new MainBehavior(this),"Main");
3  fsmBehaviour.registerState(new CheckMailBehavior(this),"Ckm");
4  fsmBehaviour.registerState(new RequestConnectionBehaviour(this),"Com");
5  fsmBehaviour.registerState(new SendMapBehaviour(this),"Smp");
6  fsmBehaviour.registerState(new ReceiveMapBehaviour(this),"Rmp");
7
8  fsmBehaviour.registerTransition("Main","Ckm",1); //main to check mail
9
10 fsmBehaviour.registerTransition("Ckm","Com",1); //check mail to start com
11 fsmBehaviour.registerTransition("Ckm","Smp",2); //check mail to send map
12
13 fsmBehaviour.registerTransition("Com","Rmp",1); //com to receive
14
15 fsmBehaviour.registerTransition("Smp","Rmp",1); // send to receive
16 fsmBehaviour.registerTransition("Smp","Main",2); // send to main
17
18 fsmBehaviour.registerTransition("Rmp","Main",1); // receive to explore
19 fsmBehaviour.registerTransition("Rmp","Smp",2); // receive to send
20
21 addBehaviour(fsmBehaviour);
```

Représentation de la Carte : Pour pouvoir expliquer le comportement des nos agents, il est nécessaire de décrire la façon dont ceux-ci sauvegardent leur représentation de la carte de l'environnement. Nous utilisons une table de hachage dans laquelle chaque clef correspond à un nœud et chaque valeur est un structure de données, dans laquelle on trouve la date de découverte de ce nœud, ses voisins et les ressources qu'il contient.

Agent Explorateur : Sa mission principale est d'explorer la carte, afin d'établir une connaissance commune de celle-ci pour tous les autres agents. Il construit la carte au fur et à mesure de ses déplacements dans sa propre table de hachage. Il met à jour dans sa structure de données personnalisées, la disponibilité et la quantité de trésors. Lors de la phase de communication, tous les agents s'échangent leur carte. On note que tous les nœuds de la carte sont horodatés, cela nous permet lors de l'échange d cartes, de sélectionner les nœuds les plus récents, lorsqu'ils sont présents dans les deux cartes. Une fois la carte complète, l'agent sélectionne des nœuds aléatoires dans la carte, pour mettre à jour leur contenu.

Agent Explorateur des nœuds plus vieux : Cet agent a exactement le même comportement que l'agent explorateur avant la complétion de la carte, cependant lorsque celle-ci est complète, son but est d'aller explorer les nœuds les plus anciens de la carte et non des nœuds sélectionnés au hasard.

Agent Collecteur : Avant la complétion de la carte, cet agent a exactement le même comportement que les agents explorateurs. Lorsque sa carte est complète, il se dirige vers les trésors de son type qui sont les plus proches de lui. Si son sac à dos est plein ou bien si il ne trouve plus de trésors de son type sur la carte, il va alors se diriger vers le Tanker pour y déposer son butin.

Agent Tanker : Avant la complétion de la carte, cet agent explore la carte de la même façon que les Agents Explorateurs pour accélérer le processus d'exploration. Quand la carte ne possède plus de nœuds inexplorés , Le Tanker va alors fixer sa position en calculant la Betweenness centrality de tous les nœuds du graphe, puis sélectionner le nœud ayant la valeur la plus grande. Cette méthode permet aux Collecteurs de calculer la position du Tanker pour qu'ils puissent y déposer leurs trésors. Le calcul de la Betweenness centrality sera expliqué plus en détails dans la section Outils.

3. Communication et interblocage

3.1. Processus de Communication

L'algorithme de communication que nous décrivons ci-dessous, a pour but de maximiser le nombre d'interactions entre agents. Nous ferons donc tout d'abord une description de l'algorithme, puis une analyse du nombre de messages échangés par un agent lors d'une discussion.

Ici le comportement `Main` est défini comme le comportement principal de chaque agent, par exemple le `ExploreBehaviour` pour l'agent explorateur.

- Après chaque action dans le Behavior (`Main`), l'agent passe au `CheckMailBehaviour` pour regarder sa boîte aux lettres. On distingue le cas où il reçoit des demandes de communication et le cas où il ne les reçoit pas.
- Si l'agent n'a rien dans sa boîte, il envoie une demande de communication à tous les autres Agents avec (`RequestConnectionBehavior`). Il passe ensuite au `ReceiveMapBehaviour` pour attendre les cartes des autres agents. Après 50 millisecondes, s'il n'en reçoit aucune,

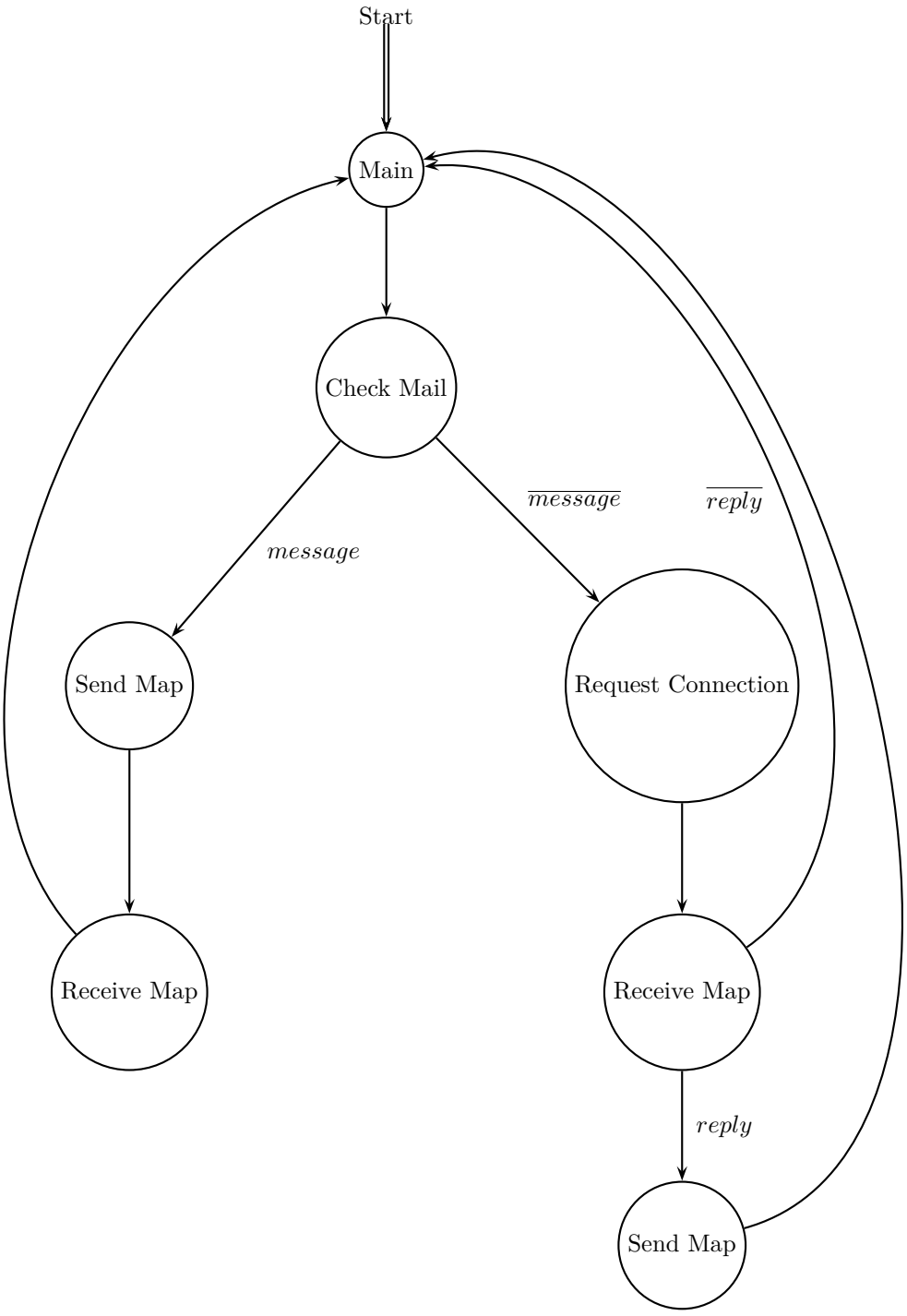
il revient au `Main`. Sinon il enverra sa carte à l'agent avec lequel il est en communication dans le `SendMapBehavior` puis fusionnera les cartes pour ensuite revenir au `Main`.

— Si l'agent a un message et que celui-ci est moins vieux que 50 millisecondes, il passe alors au `SendMapBehavior` pour envoyer sa carte à l'agent,dont il récupère le nom dans le message reçu précédemment. Après cette étape, il attend de recevoir la carte de son partenaire dans `ReceiveMapBehaviour`. S'il reçoit la carte dans la limite de temps prédéterminé, il pourra ensuite fusionner les deux cartes, et enfin revenir au `Main`.

Dans le cadre du projet, seulement les objets sérialisables et les chaînes de caractère sont autorisées dans la communication. C'est la raison pour laquelle nous utilisons une table de hachage et une structure de données sérialisables pour représenter la carte.
Nous terminons donc par l'analyse du nombre de messages échangés par un agent dans le pire des cas : nous considérons donc le cas où l'agent ne trouve pas de message dans sa boite aux lettres :

— Envoie d'un message à tous les autres agents sur la carte (**n** Agents)
— Envoie de la carte à l'agent avec lequel il est en communication

On note donc que le nombre de messages envoyés dans le pire des cas à chaque communication est **n+1**. De plus, après chaque mouvement, même si aucun agent ne se trouve dans le rayon de communication de l'agent en question, **n** messages seront envoyés. Il serait donc possible de grandement réduire le nombre de messages, en débutant la communication lors du blocage de l'agent. Cependant, compte tenu du fait que les agents ne peuvent communiquer que s'ils sont dans le rayon de communication (c'est-à-dire une distance de 2 nœuds du graphe), cette méthode réduirait grandement le nombre d'échange de cartes.



3.2. Les Outils

Calcul de chemin : Nous utilisions l'algorithme de Dijkstra, en considérant des arêtes ayant toutes le même poids, de ce fait le plus court chemin sera celui qui a le moins d'arêtes. L'implémentation utilisée de cet algorithme est celle fournie par `graphStream`. Quant à la recherche du

plus court chemin vers plusieurs cases, on applique cet algorithme vers chacune des cases et on sélectionne le chemin ayant la valeur la plus faible.

On note qu'après le calcul de la position du Tanker, nous enlevons le nœud des graphes pour le calcul de Dijkstra, cela permet d'empêcher le blocage des agents explorateurs et collecteurs sur le Tanker du fait que celui-ci ne se déplace pas. Les autres agents passent donc autour du Tanker.

La complexité de l'algorithme de Dijkstra implémenté dans graphstream est en **O(n log(n) + m)** avec n le nombre de nœuds et m le nombre d'arêtes.

Centralisation dans le graphe : Nous plaçons le Tanker sur le nœud de plus haute Betweenness centrality dans le graphe, c'est-à-dire le nœud qui est dans le plus grand nombre de plus courts chemins entre deux autres nœuds quelconques du graphe. Pour faire ce calcul, nous utilisons l'algorithme fourni par `graphStream`. Dans la plupart des cas, cette méthode est très efficace. Elle permet de minimiser le chemin que les agents collecteurs ont à faire entre leur trésor et le Tanker. Cependant, si le graphe se décompose en 2 grands sous-graphes de même taille qui se connectent par un nœud unique, une fois le Tanker placé, il est possible qu'il se trouve sur ce nœud unique. Les agents ne peuvent donc plus accéder à la partie du graphe dans laquelle ils ne sont pas.

Une autre méthode a été proposée : de prendre le nœud ayant le plus grand nombre de descendants, ce qui permet d'éviter le blocage du cas précédent. Cependant cela nécessiterait de communiquer la position du tanker à tous les agents, ce qui nous semblait être plus coûteux en terme de communication.

La complexité de l'algorithme implémenté dans graphstream qui calcule la Betweenness centrality est en **O(nm)**.

3.3. Interblocage

Nous voulions un système robuste pour la gestion de l'interblocage. Il nous semblait en effet très important de minimiser le risque que deux agents se dirigent vers une case occupée par un autre agent. C'est pour cette raison que nous avons conçu notre système de communication de façon à ce qu'un échange de carte se fasse le plus souvent possible. Bien que cette méthode soit peu économe en messages, elle minimise les interblocages dans la phase d'exploration car les agents ne se dirigent pas vers les nœuds qui se trouvent déjà dans leur carte. Nous exploitons cette fonctionnalité de la façon suivante : lorsque deux agents se trouvent à proximité de deux cases, ils échangent leurs cartes respectives et de ce fait ne se dirigent pas vers la position où se trouve l'autre agent en communication.

La méthode décrite ci-dessus ne permet cependant pas d'éviter les interblocages dans la phase de récupération des trésors : pour contrer cela lorsque l'un de nos agents n'arrive pas à faire un mouvement, il va sélectionner une case au hasard parmi ses voisins et essayer de s'y déplacer jusqu'à ce que ce soit possible.

4. Conclusion

Vous trouverez ci dessous les performances de notre algorithme sur l'instance de l'examen de 2017

Instance2017	Trésors	Diamants
Total	280	140
Ramassé	202	128

En conclusion notre méthode remplit le critère de récupération des trésors sur la carte, bien que nous remarquons un perte substantielle du nombre de trésors ramassés due à la méthode de

ramassage et au déplacement des trésors par le wumpus. Les communications entre agents, bien que couteuses en nombre de messages, permettent un partage rapide de l'information. La position de notre tanker nous semble optimale, bien que le déplacement de celui-ci nous permettrait sans doute de récupérer les trésors plus rapidement. De plus, le Tanker statique risque de créer des problèmes sur certaines cartes partagées en deux par un couloir. Notre approche vis-à-vis des interblocages pourrait se résumer de la façon suivante : nous essayons d'en minimiser le nombre avant que ceux ci se produisent, mais la gestion aléatoire des blocages est sous optimale. Nous pensons que la plus grande faiblesse de notre technique d'exploration est sa dépendance à la réussite de l'exploration totale de la carte : en effet s'il reste des nœuds inexplorés, le tanker ne sera jamais placé, et donc aucun des trésors comptabilisés.

Améliorations Nous pensons qu'il serait possible de grandement améliorer la gestion des interblocages : en effet la sélection d'une case au hasard lorsqu'un agent se bloque est sous optimale lorsque les agents se croisent dans une ligne.

Pour accélérer la récupération des trésors, il serait possible d'augmenter le nombre de Tankers, cela pourrait potentiellement diminuer le temps de trajet des agents Collecteurs vers les Tanker, on pourrait aussi imaginer un Tanker qui se déplace vers les agents collecteurs.

Nous avons aussi pensé à réorienter les agents explorateurs après la complétion de la carte pour bloquer l'agent Wumpus, cela nous permettrait de ne pas nous soucier de la mise à jour de la carte après l'exploration.