UPMC, M1 INFORMATIQUE

FOSYMA Projet

YOU Jiang

June 15, 2017

1 INTRODUCTION

Dans ce rapport nous allons discuter l'exploration d'un environnement par multi-agent. En réel, l'environnement est dynamique et partiellement observé, donc il est difficile de manipuler tous les cas par un centre de calcul. L'idée d'un système de multi-agent est de laisser les contrôles aux plusieurs agents tel qu'ils travaillent en coopérative.

Le contexte de notre projet est un environnement virtuel "Dedale" qui a été crée en base du plate-forme "Jade", un système écrit pour la communication de multi-agent. Dans ce cas, un agent connais que le nom de sa position courante et ses voisins avec quelque attribues. A fin de bien comprendre l'environnement qu'il se trouve, en fait, un agent doit construire ses propos connaissance. On utilise un graphe pour modéliser ce environnement. En détails, le noeud et l'arrête représentent la position et un chemin aux voisins. Ainsi, deux type de trésor de quantité différentes se trouvent sur différentes noeuds. Nous devons écrire un agent qui ramasse autant de trésor le plus vite possible.

Bien que la mission est simple, il y a pas mal de difficultés dans la réalisation. Par exemple, les agents se bloquent très souvent, les communications des agents se dérangent de temps en temps etc. A fin de bien résoudre ces problèmes, nous vous présentons nos solutions dans ces sections suivants.

2 ARCHITECTURE GÉNÉRALE

Nous vous présentons ici l'architecture générale du code et les diagrammes UML des communications. Nous donnons un par un les structure d'agent et des comportements, ensuite on parlera les communications entre les behaviours et l'échange des messages..

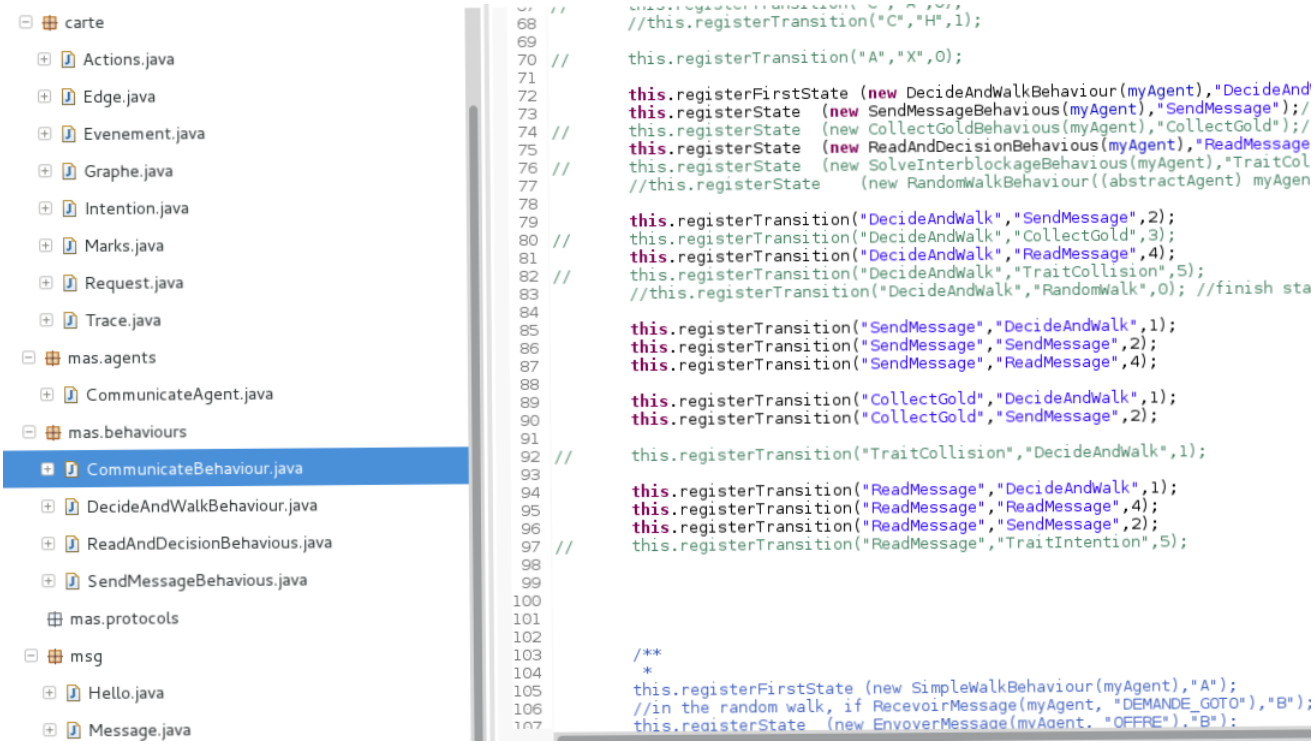

Figure 2.1: La conception d'architecture

2.1 LA CONCEPTION D'AGENT ET DES COMPOSANTS

L'agent contient des données à partage entre plusieurs behaviours tel que un **graphe** ,une liste de **message** à envoyer, une **intention**, une **trace** etc. Il propose des outils de manipuler ses données.

Le **nombre total** des agents est une variable statique, qui s'augmente de 1 quand la classe principale crée un agent. Ça permet d'avoir le nombre total de receveur quand on envoie un message au tous le monde. Le **graphe** de l'agent est mis à jours à chaque étape. Il enregistre les positions, les chemins et les trésors de l'environnement. l'objet **intention** contient une liste d'évènement à faire.
La **trace** de l'agent enregistre l'évènement passé qui est composé d'une action, une position et une description. La liste **messages** procède les messages à envoyer dans le SendMessage-

Behaviour. En fait on peut ajouter un message à n'importe quel part, mais on peut l'envoyer seulement par de ce behaviour. C'est à dire que ce behaviour gère tous les envoies de types différents.
Pour la partie de coordination et déblocage, on propose d'une liste des trésors observés **knownGoldsList** qui contient la position, le type et le quantité d'un trésor. La liste **coordination list** est pour les agents qui veulent coordonner, La liste **déblocage list** est pour es agents qui coordonner faire un déblocage. Les variables **on mission** et **on déblocage** sont deux états d'agent, pour qu'il ne soit pas découpé pendant une mission.

```
public class CommunicateAgent extends abstractAgent{
        private static final long serialVersionUID = -1784844593772918359L;
        private static int number_agent = 0;
        private Graphe gph;
        private Trace trace;
        private Intention intention;

        private ArrayList<Message> messages;

        private ArrayList<String[]> knownGoldsList;
        private ArrayList<Object[]> ignoreGoldsList;

        private ArrayList<String[]> coordination_list;
        private ArrayList<String> d blocage_list;

        private boolean isOnMission=false;
        private boolean isOnInterblockage=false;
...
}
```

Un **graphe** contient des noeuds, des arrêtes, et des marques. les marques sont mis jour à chaque étape. Il y a aussi quelque fonctions pour trouver un chemin, calculer une distance, ou trouver les information d'un noeud spécifié. Le marque d'un noeud est une ou plusieurs attribue dans le "MARK VALUE". On met la valeur "INSIGHT" quand on le voit, puis on met "EXPLORED" si un l'agent passe sur ce noeud. S'il y a du trésor sur ce noeud, on met sa type de dan.

```
public class Graphe  implements Serializable{
        private static final long serialVersionUID = 279309039702598912 9L;

        private String[] MARK_VALUE= {"EXPLORED","INSIGHT","TREASURE", "DIAMONDS"};
        private String racine;
        private HashSet<String> noeuds;
        private HashSet<Edge> edges;
        private Marks marks;
        ...}
```

L' **évènement** est composé d'une action, d'une position et d'une description. Un évènement peut être dans l'intention si l'agent veut exécuter ce action, ou bien dans la trace si l'agent a exécute ce action récemment.

```
public class Evenement implements Serializable{
        private static final long serialVersionUID = 5506875014373482727L;
                private Actions action;
                private String nod;
                private String description;
```

Le **message** permet d'échanger plusieurs type d'information tel que le graphe, le trésor et l'intention selon le cas. On peut spécifier un receveur, un petit message etc. Normalement, un message est bien crée avant d'entrer à SendMessageBehaviour.

```
public class Message implements Serializable {
        private static final long serialVersionUID = -8214953356962925299L;
        //
        private final String TYPES[]={"MAP","GOLDS","INTENTION"};
        private String type="";
        private Graphe graphe;
        private String[] golds=null;
        private ArrayList<String[]> golds_list;
        private ArrayList<String[]> intention;
        private String receiver="all";
        private String message="";
        private String[] agent_info;
        ...}
```

2.2 La construction de behaviours

Dans le CommunicateBehaviour, on définie les états des behavious(1 , 2 , et 4, les autres chiffres sont supprimés.). Les behaviours sont complètement connexes pour simplifier le système.
Tous les actions tel que "EXPLORE", "DECIDE", "GOTO", "COLLECT", "RANDOMWALK", "SEND", "RECEIVE" etc, sont dans le DecideAndWalkBehaviour. Tous les trois behaviours peuvent ajouter les évènements à la liste "intention" de l'agent. Puis l'agent exécute les événement de l'intention un par un. Ceci est le contrôle principal du système. le but global est de ramasser les trésors et d'explorer la carte. Finalement, l'agent commence à "RANDOM WALK" quand il n'a rien à faire.

Dans ce nouveau FSMBehaviour, le systmem entre le BDIBehaviour d'abord. Quand il a une intention de passer aux autres Behaviours, il se change l'état. Il utilise une Les détails de

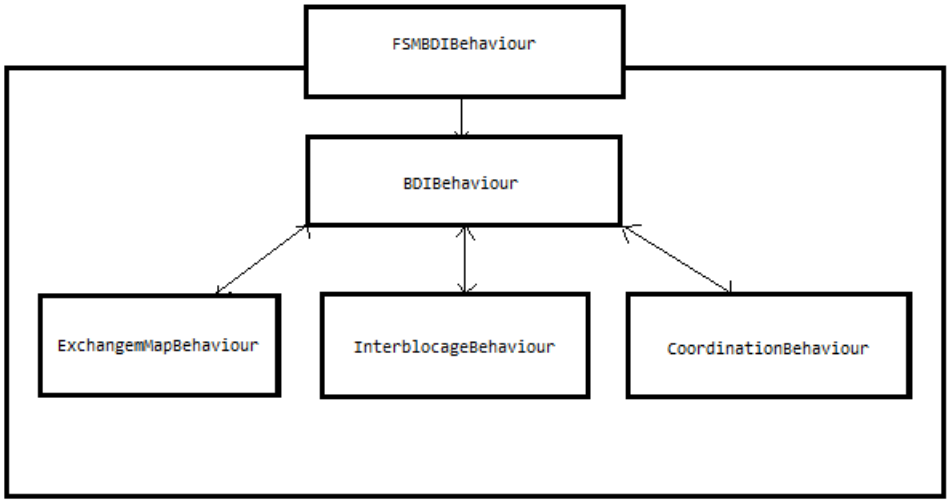

Figure 2.2: La machine d'état des Behaviours

protocoles sont écrits dans chaque behaviour. Les actions, la génération et l'exécution de plan sont écrit dans le BDIBehaviour.
Pour modéliser l'échange de carte, la coordination et l'inter-blocage, on utilise le protocole de fusionner. Ce conception est assez intuitive et simple.

Analyse des protocols

- **Echange de carte**
  Un agent envoie sa **carte** aux agents au tour de lui d'abord. Les agents qui reçoivent ce carte lui répond celle de soi même, puis ils en fusionnent. Ensuite le premier agent reçoit plusieurs cartes puis il en fait fusionner. Enfin, il envoie ce dernier carte aux toutes les agents concernés pour que les connaissances pour chacun sont pareilles. Donc ce protocole est symétrique.

  La complexité de ce protocole dépend le nombre d'agent N. Comme chaque fois la communication a 3 étapes, donc il y a 3*N de messages dans le pire cas.

  Ce protocole d'échange de carte permet d'accélérer l'exploration de la carte. Dans le meilleure de cas, chaque agent possède équitablement une partie de la carte et donc, un échange suffit d'avoir la carte entier. Alors c'est N fois plus vite pour l'exploration.

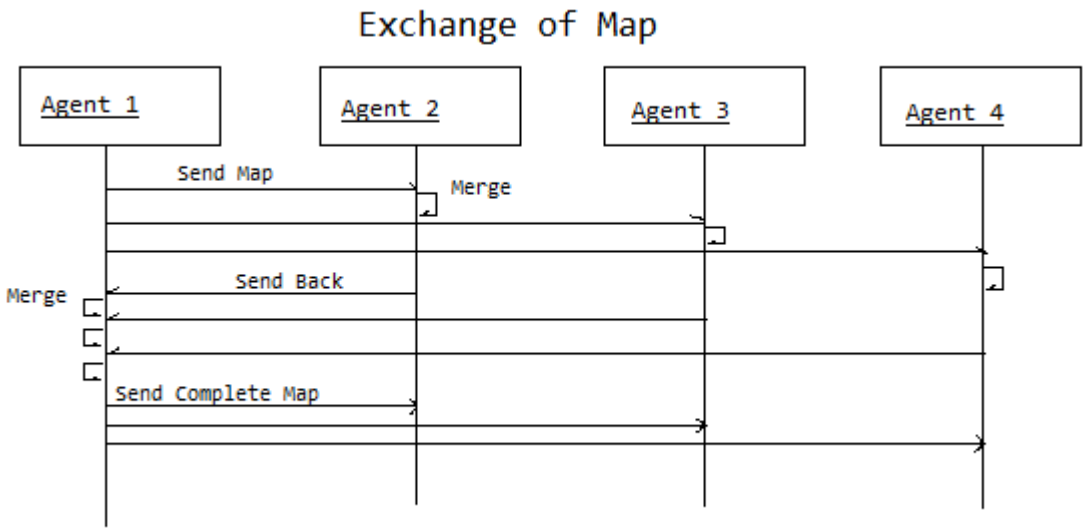

Figure 2.3: Le UML de l'échange de carte

- **Collecter les trésors**
  Un agent envoie une proposition de **coordination** aux agents au tour de lui d'abord. Les agents qui reçoivent ce proposition répondent lui un "Oui" si ils ont des capacités libres. En fin le premier agent reçoit plusieurs réponses puis il fait un calcul de préférence. Enfin, il réponds à ces agents avec une confirmation et une liste de coordination.

  Chaque agent connais une liste de trésors et une liste de candidates pour chaque trésor. Après un calcule, l'agent envoie justement ce nouvelle liste de coalition aux autres. Puis Les autres agents mettent à jour. Finalement, un agent collecte sa meilleure trésor dans la liste de candidature.

  En fait, la coordination de ramasser se commence quand un agent pense qu'il connais bien la carte. Dans la réalisation, l'agent commence à collecter les trésors quand il connais plus que 100 noeuds.

  La complexité de ce protocole dépend le nombre d'agent N, et donc il a besoin au pire 3*N échange de message.
  Dans le meilleure cas, ce protocol permet d'avoir une coalition parfait. C'est à dire, quand tous les agent connais tous les trésors, le calcul forment la meilleure coalition.

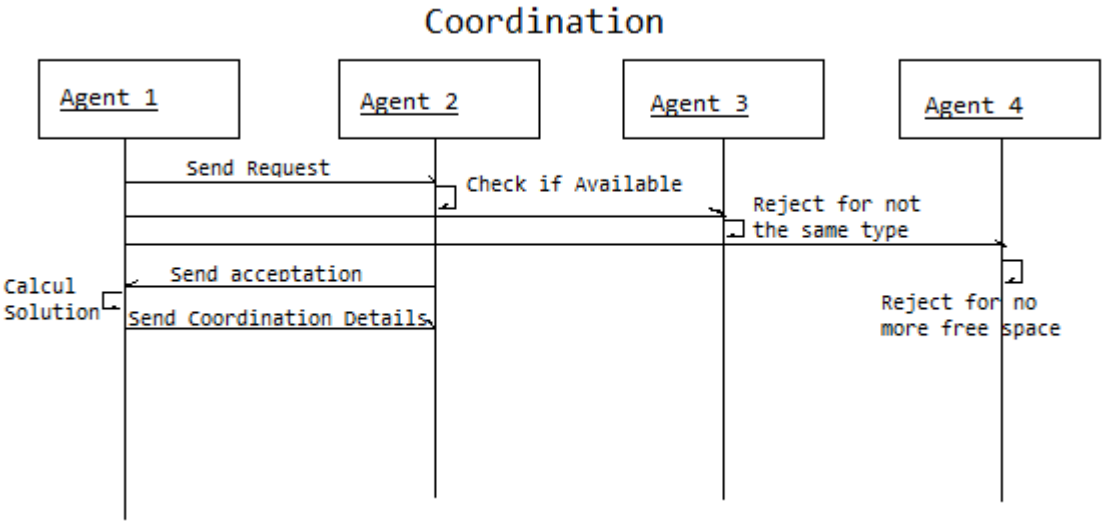

Figure 2.4: Le UML de la coordination

- **Interblocage**
  Un agent envoie une proposition d'**interblocage** aux agents au tour de lui d'abord. Les agents qui reçoivent ce proposition répondent lui un "Oui" et des informations tel que sa position, sa intention etc, si ils sont aussi bloqués. En fin le premier agent reçoit plusieurs réponses puis il réponds à ces agents avec une confirmation et une liste de stratégie.

  La stratégie concerne une suite d'action pour chaque agent. On essayer de simplifier ce protocole, donc la fonctionnement dans le tunnel n'est pas très idéal.

  La complexité de ce protocole dépend N.

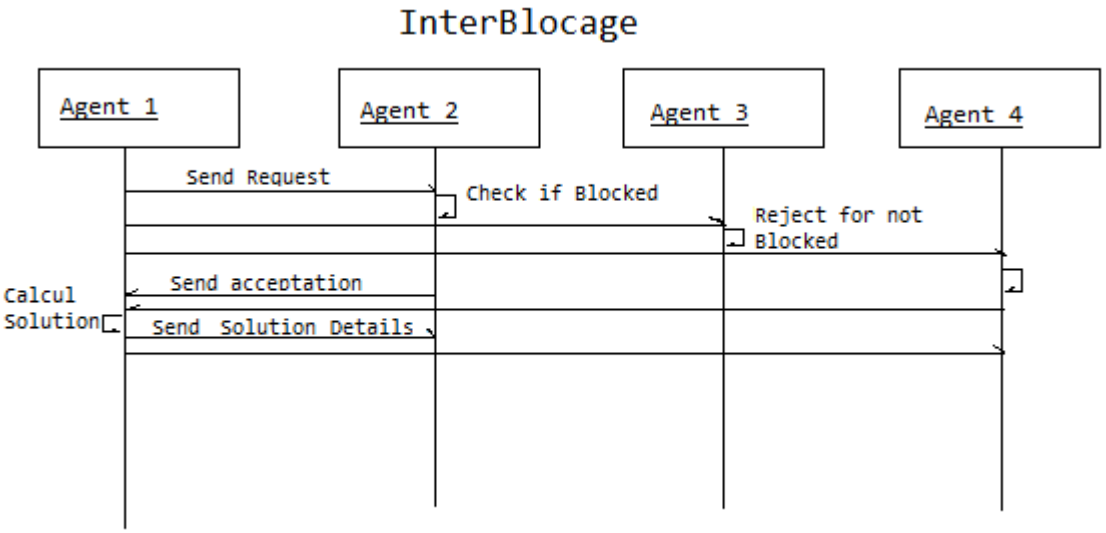

Figure 2.5: Le UML de la déblocage

3 Présentation des Algorithmes

- **l'exploration de la carte**
  Pour **l'exploration de la carte** on utilise la méthode A^*, avec une heuristique qui explore à priori des noeuds non explorés et le plus proches. Il retourne nulle quand il n'a plus de noeud à explorer. Ce méthode est assez stable, donc un agent peut facilement réussir à explorer tous les noeuds.

- **l'échange de la carte**
  Pour **l'échange de la carte**, on utilise le protocole "fusionner". Quand un agent reçoit une carte, il fusionne les noeuds et les arêtes. Le marque d'un noeud est mis à jours par le remplacement en ordre "EXPLORED > INSIGHT > inconnu ".

- **coordination de ramasser**
  La coalition des agents se calcul pendant l'échange des information. En détails, l'algorithme trie les groupes d'agents par leur valeur de coalition qui est la capacité reste. Finalement, la réponse aux agents à la coordination est une lise de coalition. Quand un agent pense qu'il est prêt de ramasser un trésor, il passe directement vers ce noeud. Si il est le premier, il collecte directement, si non, il attend un mis à jours de l'information de ses collègues.

- **Déblocage**
  L'algorithme de **déblocage** trouve une nouvelle chemin sans passé les noeuds occupés par les autres. Si ils sont dans le tunnel, alors l'agent ayant une priorité le plus grand

contrôle les autres agents bloqués sur sa chemin de l'intention, tel qu'ils reculent jusqu'aux noeuds non bloqués.

4 CONCLUSION

Puisque on a réalisé ce système avec les stratégies simples, alors la performance n'est pas très idéal. de temps en temps ils se bloquent. Donc nous avons pas mal d'amélioration à faire. Par exemple nous pouvons écrire un système de déduction de planification, pour qu'il manipule bien les inter blocages.

En conclusion, la planification est la partie le plus complexe dans ce projet. Nous devons respecter d'abord les homogénéité d'agent, c'est-à-dire qu'un agent est celui d'envoi et celui de reçoit.Alors il faut bien distinguer les différentes étapes pour que la communication ne se confonds pas. De plus, le moment d'accepter une proposition et de rejeter, c'est aussi un sujet à discuter dans les futurs améliorations.