Projet de FOSYMA

Wumpus Multi-Agent

Alexandre Bontems, Hans Thirunavukarasu

Table des matières

1. Introduction

Une version multi-agent du jeu « Hunt the Wumpus » est implémentée dans ce projet. On y considère une carte de labyrinthe sous forme de graphe dans laquelle certains sommets présentent des trésors. Ceux-ci peuvent être de deux types, trésors ou diamants, et présents en une quantité prédéfinie. Les agents évoluent dans cette carte en s'y déplaçant : passer d'un sommet v_1 à un sommet v_2 n'est permis que si l'arête (v_1, v_2) existe dans le graphe. De plus, chaque agent occupe un sommet du graphe à tout moment et plusieurs agents ne peuvent occuper un même sommet. Les déplacements sont donc susceptibles d'être bloqués si le sommet de destination est déjà occupé.

Le but final du jeu est d'explorer entièrement le graphe et de récupérer tous les trésors qui s'y trouvent. Pour cela on dispose de plusieurs types d'agents : explorateurs, collecteurs et silo. Les explorateurs ont pour fonctions d'explorer la carte et de reporter la position des trésors. Les collecteurs possèdent un sac-à-dos pouvant contenir une certaine capacité d'un seul type de trésor et peuvent ramasser les trésors de ce type qui se trouvent à leur position. Le ramassage n'est pas parfait cependant puisqu'à chaque tentative un partie du trésor présent sur le sommet est perdue à jamais. Enfin le silo possède une capacité illimitée pour tous les types de trésors et les collecteurs peuvent donc lui donner le contenu de leur sac-à-dos lorsqu'il est plein.

Le *wumpus*, un agent adverse, est lui aussi déployé dans la carte. Son déplacement est aléatoire mais il est capable de déplacer des trésors sur la carte et d'en réduire ainsi la quantité totale définitivement. Il est lui aussi sujet aux restrictions décrites ci-dessus.

Dans ce rapport, l'exploration est d'abord abordée, comportement susceptible d'être adopté par tout agent quelque soit sont type. Tous les thèmes liés à la coordination tels

que les communications, la gestion des interblocages et le ramassage de trésor sont ensuite décrit.

2. Exploration

Le comportement d'exploration a été le premier à être implémenté en raison de son indispensabilité. En effet, tout autre comportement ne peut être fonctionnel que si l'environnement de l'agent est connu. C'est aussi le comportement par défaut de tous les agents lorsqu'ils n'ont pas d'objectif plus pressant. Au lancement de l'exécution par exemple, les collecteurs ne connaissent pas les positions des trésors ni du silo et pour les découvrir on passe en mode d'exploration. Pour la même raison, le silo ne connaissant pas la position des autres agents, se tourne d'abord vers une phase d'exploration.

Puisque l'environnement est modélisé sous forme de graphe, il est naturel pour le parcourir entièrement de se tourner vers un comportement inspiré de Breadth First Search (BFS). Ainsi trois structures de données sont utilisées : premièrement, la carte est sauvée en tant que listes de voisins (`HashMap<String, HashSet<String>>` en java) et associe à chaque sommet ses voisins dans le graphe. Un ensemble de sommets ouverts est aussi maintenu : ce sont les sommets encore non explorés. Enfin un ensemble de sommets déjà explorés est gardé en mémoire. Chaque agent en mode d'exploration construit donc une carte en choisissant le sommet ouvert le plus proche et en l'ajoutant à la carte (lui et ses voisins) lorsqu'il est atteint. Les chemins jusqu'à un sommet sont calculés par BFS depuis le sommet de départ et le sommet ouvert le plus proche est donc celui pour lequel le chemin est le plus court. À chaque choix de sommet de destination on a donc une complexité de $O(n+m)$ pour le BFS et une carte de n sommets et m arêtes.

Lorsque l'exploration est finie, i.e. un agent n'a plus de sommets ouverts, les agents explorant ont plusieurs options. Tout d'abord, lors des phases de communication (voir Section 3.1) les agents se mettent d'accord sur l'identité de l'explorateur dit de « mise à jour des points d'intérêts ». Cet agent aura pour mission, lorsqu'il a terminé son exploration, de se déplacer entre chaque sommet connu comme contenant un trésor afin d'avoir la représentation la plus récente possible de chaque trésor de la carte. Il lui sera ainsi possible lors des rencontres de partager cette information. Les autres agents quant à eux relancent une exploration complète de la carte en « oubliant » quels sommets ont déjà été visités. Cela est nécessaire puisque le *wumpus* est capable de déplacer des trésors et il est donc important de repasser par tous les sommets pour détecter les nouveaux points d'intérêt. L'agent de mise à jour choisit donc le point d'intérêt le plus ancien, planifie un chemin pour s'y rendre et lorsque ce point est atteint passe au prochain point le plus ancien. Un comportement sensiblement plus intelligent serait de concentrer l'exploration dans le voisinage des points d'intérêt déjà connus car le *wumpus* a seulement une faible portée de déplacement. Cependant, dans un soucis d'optimisation des interblocages et par manque de temps, ce comportement n'est pas implémenté.

Lorsque deux agents se rencontrent les cartes peuvent être échangées et aucune vérification n'est nécessaire car la carte ne peut pas évoluer dans le temps. La liste des trésors cependant est changeante et c'est pourquoi à tout point d'intérêt est associée une date d'observation. Lors des communications les agents sont ainsi capables de comparer les

dates et de distinguer l'observation la plus récente. Ils utilisent pour cela l'horloge système de la machine sur laquelle ils sont lancés.

3. Coordination

Qui dit simulation multi-agent dit calcul distribué et donc un besoin primordial de communication entre processus. Chaque agent est ainsi capable d'échanger des messages avec d'autres agents aux travers d'un réseau (ces messages utilisent la norme FIPA). Ces communications forment la fondation des différents comportements de coordination qui sont décrit par la suite.

3.1. Communication

Pour pouvoir expliquer plus en avant les communications inter-agent il est nécessaire d'introduire la machine à états qui les régit. Visible en Figure 1, la machine et ses transitions sont expliquées ici. On y voit que le premier état atteint est bien celui de l'exploration mais il est surtout important de pointer du doigt l'état central CheckVoiceMail. C'est grâce à lui que les agents sont capables de s'écouter et se parler. En effet après chaque mouvement, chacun vérifie dans sa boîte aux lettres qu'aucun message n'a été reçu. En parallèle des données transitent entre nos differents comportements grâce a un DataStore pour pouvoir transmettre la situation actuelle de l'agent. Ainsi, des flags sont contenu dans ce DataStore et sont utilisés pour permettre le bon redirigement lorsque l'on est dans CheckVoiceMail. Tout d'abord si l'agent a reçu un message de type Request dans sa boite mail, cela signifie qu'un autre agent lui a demandé de s'arrêter, il envois donc un message ACK à cet agent et passe à l'état SendData dans lequel il va envoyer les informations de sa carte. Cette étape est nécessaire pour établir la communication entre nos deux agents car rappelons le, les deux doivent être immobile pour entamer une communication. En revanche si aucun message n'a été reçu, on regarde alors si le flag « block notification » de notre agent est a true pour enclencher une procédure de communication avec l'autre agent qui nous bloque. Nous passons ainsi à l'état RequestStandby dans lequel l'agent enverra un message de type Request en « Broadcast » car il ne connait pas forcément l'agent qui le bloque. Enfin toujours lorsque l'on est dans CheckVoiceMail, si aucun message n'a été reçu et que l'agent n'était pas bloqué, alors il retourne dans l'un des quatre comportements de déplacement suivant : Explore, AvoidConflict, RandomWalk ou TypeSpecificMvmt. En effet, toujours à l'aide des flags contenu dans notre DataStore, l'agent est redirigé vers le bon comportement. Par exemple il existe un flag « avoid conflict » qui est potentiellement mis à true lorsque l'on est dans l'état RcvGoal, or de RcvGoal nous retournons à CheckVoiceMail, qui redirigera donc notre agent vers le comportement AvoidConflict en adéquation avec la valeur booléenne true de notre flag.

3.2. Interblocage

Deux grosses situations se dégagent donc de notre machine à états lors d'un inter-blocage. La première, notre agent reçoit une requete de Standby d'un autre agent et la deuxieme, notre agent envoie une requete de Standby aux autres agents. L'éxécution étant

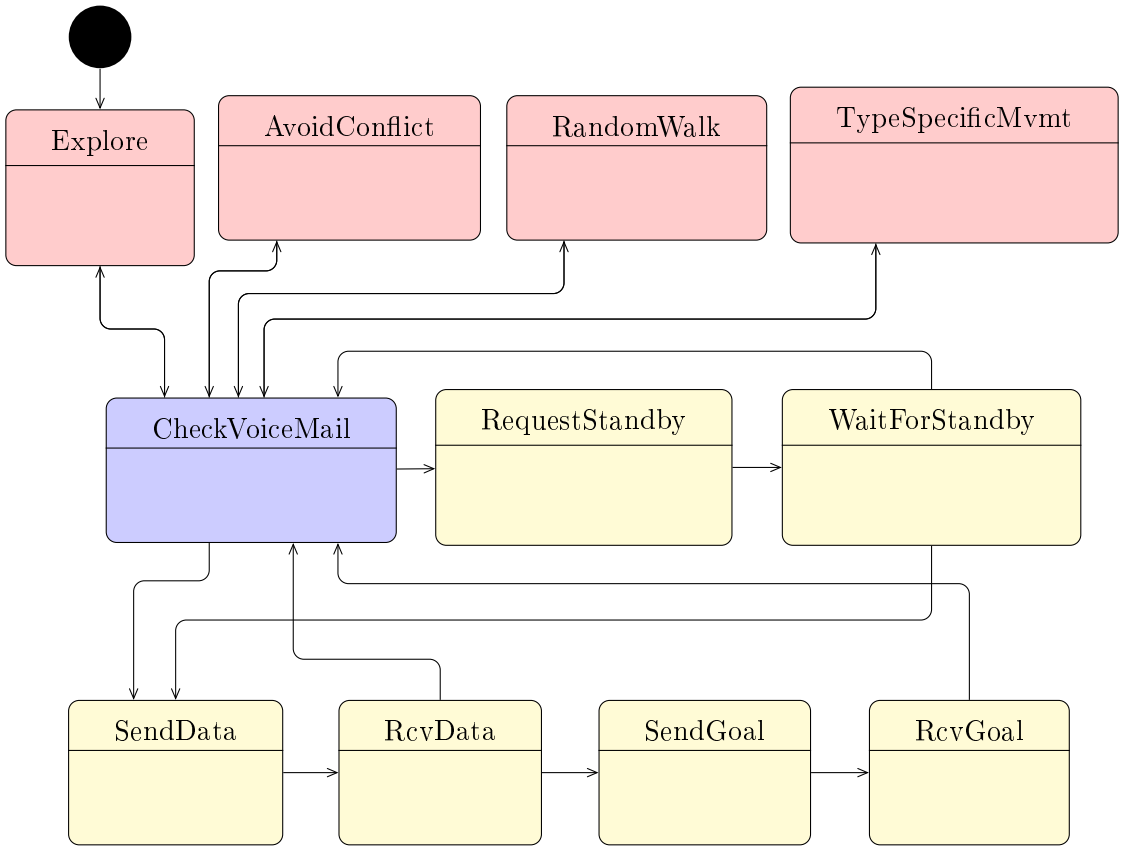

FIGURE 1 – Machine à états finie des agents.

concurrente et l'envois de messages étant asynchrone, tout dépends donc du premier agent à detecter l'interblocage. Si notre agent est le premier a le detecter (cet-à-dire que la fonction moveTo de Jade renvoie false) alors le flag « block notification » est mis à true et lorsque l'on arrive dans CheckVoiceMail, on est alors redirigé vers l'état RequestStandby. Après avoir envoyé sa requête en « Broadcast », il passe ensuite à l'état WaitStandby où il attend confirmation de l'autre agent. S'il reçoit le ACK (s'il ne le reçoit pas, il est redirigé vers l'état central) , il passe à l'état SendData qui est donc le début de la communication entre les deux agents. Il envoit donc ses données de la carte, et attends celles de l'autre agent à l'état RcvData. Enfin les deux agents s'échangent leur destination et leur chemin pour pouvoir résoudre l'interblocage via les états SendGoal et textsfRcvGoal , similaires aux deux états précèdents. Plus précisément, c'est dans l'état RcvGoal que se décide comment sera gerer l'interblocage. En effet, les deux agents ayant maintenant donc connaissance de la même carte, et le goal de l'autre agent, ils peuvent donc se mettre d'accord sur l'action à faire par la suite. L'interblocage est donc déroulé de cette façon : les deux agents calculent leur « escape route », cet à dire qu'ils vont regarder dans le chemin de l'autre, la premiere node où il est possible de s'échapper pour pouvoir laisser le passage (concrètement, on regarde parmis les noeuds voisins du chemin , s'il existe un noeud qui n'en fait pas partie et qui est accessible). Ainsi si l'un des deux agents ne trouve pas de

« escape route », il aura donc la priorité et l'autre agent devra donc lui laisser le passage. Cet agent aura donc le flag « avoid conflict » à true ce qui lui permettra d'être rediriger vers le comportement AvoidConflic où il executera son « escape route » tandis que l'agent ne pouvant pas s'échapper, attendra tout simplement que l'autre agent se déplace. Par ailleurs, si les deux agents trouvent des chemins pour s'échapper, alors celui avec le chemin le plus cours devra se deplacer et laisser le passage à l'autre, tout ça dans le but de régler l'interblocage le plus rapidement possible. En revanche si les deux agents ne trouvent pas de solutions, alors leur flag respectif « random walk » est mis à true : ils partent donc en CheckVoiceMail qui les redirige à l'état RandomWalk où ils executeront des mouvements aléatoires dans l'espoir de débloquer l'interblocage. Il est important de remarquer que dans la plupart des scénarios d'interblocages (hormis les situations de tunnel), un comportement de mouvements aléatoires règle très souvent l'interblocage.

3.3. Ramassage de trésor

3.4. Placement du Silo

4. Conclusion