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 le point d'intérêt le plus ancien (en terme de date d'observation), 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. Nous avons mis en place un système de requête d'arrêt et d'acquittement pour tenter de limiter le nombre de messages nécessaires au bon fonctionnement des agents. Les agents initient donc des communications lorsque l'un de leur mouvements est bloqué.

Pendant toute l'exécution des agents, des données transitent entre nos différents comportements grâce à un `DataStore` commun (ces données représentent la situation actuelle de l'agent). Ainsi, un certain nombre de flags y sont contenus et sont utilisés par la machine à état pour choisir la bonne transition lorsque l'on est dans CheckVoiceMail.

On peut distinguer plusieurs situations lorsqu'un agent est dans cet état. Tout d'abord si un message de performative *Request* est reçu, cela signifie qu'un autre agent lui a demandé de s'arrêter. Ce système de requête permet de s'assurer que les agents restent à portée l'un de l'autre et puissent exécuter l'entièreté du protocole de communication. Après avoir reçu une requête donc, l'agent envoie un message d'acquittement (performative *Confirm*) et passe dans l'état SendData grâce auquel il pourra partager les informations de sa carte (performative *Inform*). Dès que l'acquittement est reçu de l'autre côté de la transaction, l'autre agent fait de même. Si le protocole de communication se déroule sans accroc, ils passent tous deux par les états suivants : RcvData où les cartes sont reçues et mises à jour, SendGoal où les buts et plans respectifs sont envoyés (performative *Propose*), RcvGoal où ils sont reçus. Il est nécessaire d'exécuter l'envoi des buts après l'envoi de la carte car sa mise à jour peut affecter le calcul du meilleur plan. On note aussi que c'est dans la phase de réception des buts que les interblocages potentiels sont détectés.

Si dans l'état CheckVoiceMail aucun message n'a été reçu, on regarde la valeur que prend le flag « block_notification ». Si l'agent a effectivement été bloqué lors du dernier déplacement mais qu'aucune requête n'est visible, on en déduit qu'on est le premier à détecter le blocage et la procédure de communication est amorcée avec l'agent bloquant. Nous passons ainsi à l'état RequestStandby dans lequel l'agent enverra le message de performative *Request* en « broadcast » car il ne connait pas a priori l'agent qui le bloque. Après cet

envoi, il entre dans l'état WaitForStandby et a alors deux options : soit un acquittement est reçu et le protocole peut continuer, soit personne ne répond et on repasse à CheckVoiceMail pour vérifier que l'autre agent ne souffrait pas de désynchronisation.

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, l'agent est redirigé vers le bon comportement. Ici, TypeSpecificMvmt représente les comportements de mouvement spécifiques aux types des agents ; cela peut donc être la collecte ou le mouvement du silo par exemple.

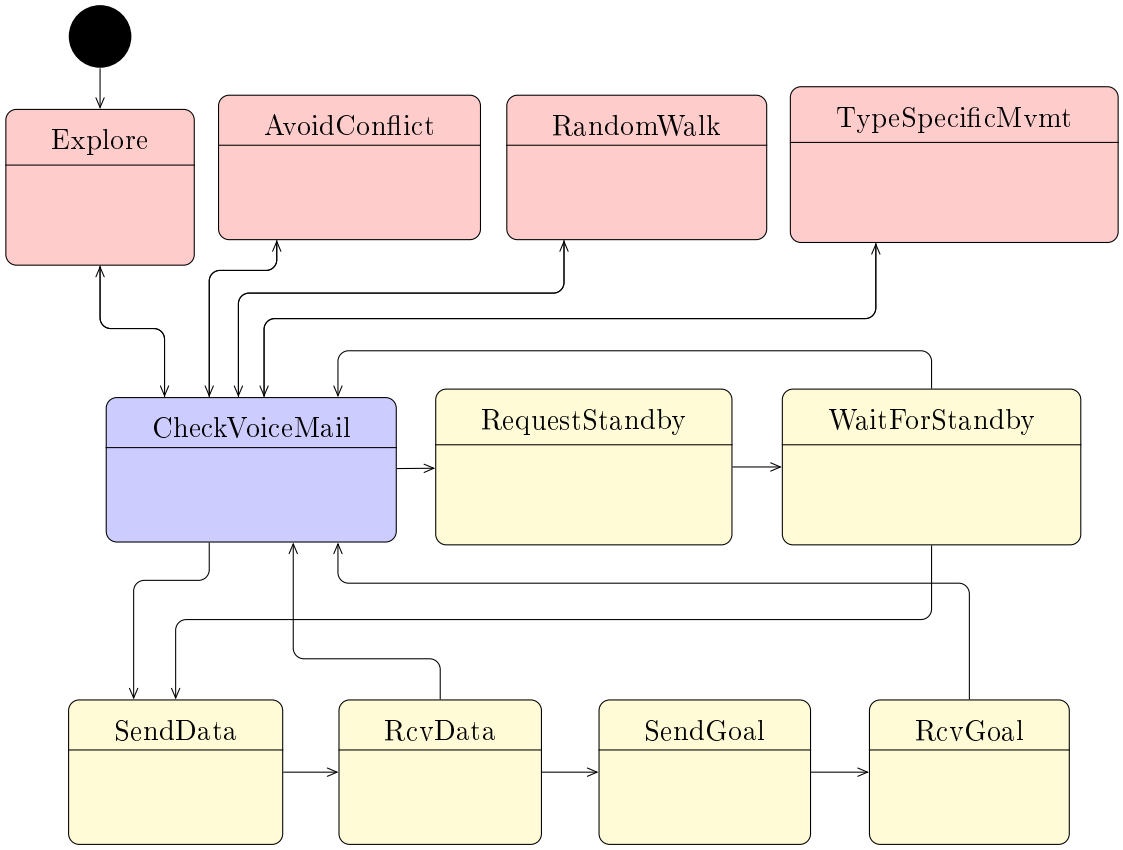

FIGURE 1 – Machine à états finie des agents.

Tous les états de réception sont bornés en temps et après toute attente de message, les agents ont la possibilité de revenir vers CheckVoiceMail si rien n'a été reçu dans le temps imparti.

Le maximum de messages échangés par conversation est donc au nombre de six. Si tout se déroule comme prévu, chaque agent envoie trois messages et reçoit trois messages (soit *Request* → *Inform* → *Propose* si on initie la conversation, soit *Confirm* → *Inform* → *Propose*). Nous nous sommes rendu compte après un certain temps des limites du système de requête. Il apparaît en effet que partager la carte en mode broadcast pourrait optimiser le

partage d'information et la phase d'exploration. Cela étant, cela augmenterait énormément le nombre de messages échangés et le nombre de conversations initiées. Comme les agents s'arrêtent à chaque conversation, cela pourrait également augmenter le nombre de blocages et/ou ralentir l'exécution globale (?). À vérifier, nous n'avons pas eu le temps de tester.

Il est important de noter qu'un certain nombre de conversations ne sont pas menées à complétion. Les raisons sont principalement dues à l'asynchronisme des agents qui peuvent vérifier leur boîte mail trop tard ou détecter un blocage trop tard. Le nombre de messages total à la fin de l'exécution n'est donc pas forcément un multiple de six.

3.2. Interblocage

Lors d'une conversation, les deux agents concernés s'échangent leur destination et leur chemin pour pouvoir résoudre un éventuel interblocage (via les états SendGoal et RcvGoal). Plus précisément, c'est dans l'état RcvGoal que se décide s'il y a interblocage et comment il sera géré. Les deux agents ayant connaissance de la même carte et du but de l'autre agent à ce stade de conversation, ils peuvent chacun de leur côté analyser la situation et décider de la prochaine étape. Le blocage d'un agent est détecté si la position de l'autre est la prochaine destination à atteindre. Si les agents estiment qu'un des deux agents n'est pas bloqué, alors le bloqué attend simplement que l'autre se déplace. Si les deux agents sont bloqués alors ils doivent prendre la même décision.

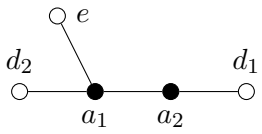

Figure 2 – Exemple de situation d'interblocage

Les deux agents calculent les « escape route » de chacun, c'est-à-dire qu'ils vont regarder dans le chemin de l'autre, le premier sommet où il est possible de s'échapper pour pouvoir laisser le passage (concrètement on regarde parmi les sommets voisins du chemin s'il existe un sommet qui n'en fait pas partie et qui est accessible). Ainsi si l'un des deux agents ne trouve pas d'« escape route », il aura la priorité et l'autre agent devra donc lui laisser le passage. Ce dernier aura donc le flag « avoid_conflict » à vrai ce qui lui permettra d'être redirigé vers le comportement AvoidConflict où il exécutera son « escape route » tandis que l'agent ne pouvant pas s'échapper attendra que l'autre agent se déplace. Si les deux agents trouvent des chemins pour s'échapper, alors celui avec le chemin le plus court devra se déplacer et laisser le passage à l'autre, tout cela dans le but de régler l'interblocage le plus rapidement possible.

Dans l'exemple visible en Figure 2, l'agent a_1 veut aller en d_1 et l'agent a_2 en d_2. En reculant l'agent a_2 ne peut pas laisser place à l'agent a_1. En revanche, l'agent a_1 peut reculer en e et laisser l'agent a_2 avancer. Les calculs des deux chemins sont exécutés par les deux agents pour réduire le temps de conversation.

Ce système fonctionne extrêmement bien avec deux agents mais devient rapidement inefficace si plus de deux se retrouvent dans un couloir. C'est pourquoi un comportement

de mouvement aléatoire (RandomWalk) a été mis en place. Il est activé à chaque fois qu'un agent se trouve bloqué sur un sommet au moins deux fois consécutives (on en déduit que le système d'échappée n'a pas fonctionné ou que l'on est en face du *wumpus*). Nous avons tout d'abord testé un mouvement dans lequel les agents choisissent uniformément aléatoirement un voisin de leur position et s'y déplacent. Dans les cas de grand groupement cependant il est peu efficace car souvent ils ne se déplacent pas assez loin pour régler l'interblocage. C'est pourquoi nous avons plus tard implémenté un comportement « walk to random » dans lequel les agents choisissent un sommet aléatoirement dans leur carte. Cette solution est loin d'être parfaite car elle peut ne pas régler l'interblocage et peut alors engendrer une nouvelle conversation. L'agent est capable néanmoins, avec probabilité non nulle, de tirer un sommet qui lui demandera de reculer. Un comportement plus intelligent permettrait de tirer un sommet le forçant à reculer mais par manque de temps cette solution n'a pu être testée.

3.3. Ramassage de trésor

Le ramassage des trésors est opéré par les agents de type collecteurs qui maintiennent comme tous les agents une liste de points d'intérêt. S'ils sont initialement en mode d'exploration, la collection commence dès que des trésors pouvant être ramassés sont connus.

Tous les agents, quelque soit leur type, propagent à chaque conversation la liste complète des agents collecteurs. Chacun est donc capable, après un certain nombre de conversations, de connaître le type et la capacité maximum de tous les collecteurs et cette connaissance influe grandement sur le choix des buts. En effet lorsqu'un trésor est considéré, si un agent collecteur de meilleure capacité pour ce trésor est connu alors la collecte ne sera pas tentée. De plus, si un agent décide de collecter un trésor mais que sa charge actuelle réduit la quantité récupérable en un trajet, alors le collecteur cherchera le silo pour se décharger. On tente ainsi de minimiser le nombre de collecte de trésor et donc minimiser la quantité perdue.

Si un agent ne connaît plus de trésor intéressant pour lui alors il repasse en exploration. On espère ainsi qu'il en découvre des nouveaux ou qu'il rencontre un autre agent lui apprenant des nouvelles informations. Si tous les trésors ont été ramassé c'est son état de terminaison. Il cherche également à se décharger à intervalles régulier pour que tous les trésors soient placés dans le silo à terminaison.

Comme la propagation de la liste complète des collecteurs peut prendre du temps, la collecte peut ne pas être optimale et ne pas minimiser les pertes de trésor. Nous pourrions délayer la collecte jusqu'à ce que la liste complète soit connue (on peut connaître le nombre total de collecteurs via le service DF de Jade). Il nous faut au minimum $2c - 3 + (n - c)$ messages pour propager, avec c le nombre de collecteurs et n le nombre total d'agents.

3.4. Placement du Silo

Le silo est statique pendant la majeure partie de l'exécution mais commence, comme tout le monde, en mode exploration. Lorsqu'il rencontre un autre agent et met à jour la carte il est capable de calculer la meilleure position sur laquelle se placer pour être

atteignable par les collecteurs. Si lors d'une autre conversation une meilleure position est apprise alors le silo est capable de s'y déplacer.

Nous avons choisi pour cette position, le sommet de plus grande « betweenness centrality » : c'est le sommet par lequel passent le plus de plus courts chemins au sein du graphe. Il a plusieurs avantages pour cette implémentation : il maximise la probabilité de rencontres et permet donc au silo de favoriser le partage d'information en prenant le rôle d'expert. Ce sommet a aussi la particularité d'avoir un grand degré et assure le bon fonctionnement de notre système d'interblocage. Lors de telles situations, il y a de fortes chances que le silo soit capable de se déplacer sur le sommet voisin et régler l'interblocage. Nous aurions pu planifier les chemins en évitant le silo mais le système d'interblocage est efficace dans la plupart des cas et on veut des conversations avec le silo pour qu'il apprenne le plus possible.

L'agent silo calcule la meilleure position dans l'état RcvData et la communique à l'autre agent dans SendGoal. Trouver le bon sommet implique le calcul de tous les plus courts chemins du graphe. Cela est fait naïvement et donc via n parcours BFS (n sommets).

Chaque agent garde ensuite en mémoire cette information et elle est propagée à chaque conversation. On a donc besoin de $n-1$ conversations au minimum pour que tous les agents puissent connaitre la position du silo (avec n le nombre total d'agents). Si la position change la propagation doit recommencer mais les agents se rencontrent assez souvent pour qu'une information manquante ou erronée ne soit pas problématique pendant longtemps.

4. Conclusion

Dans ce projet nous avons cherché à

Nous sommes conscients du potentiel d'amélioration de toutes les stratégies implémentées dans ce travail. Le *wumpus* et son odeur par exemple, ne sont pas pris en compte et les communications pourraient être améliorées. Cependant les comportements sont fonctionnels et donnent des résultats satisfaisants lors de nos tests. Sur une carte avec 400 trésors au départ, le silo contient en moyenne 320 trésors à la fin de l'exécution.