Sorbonne Université

Bruce Rose
Hugo Abreu
Groupe 12

M1 androide

Fondements des Systèmes Multi-Agents

Projet - Hunt the Wumpus

30 Avril 2021

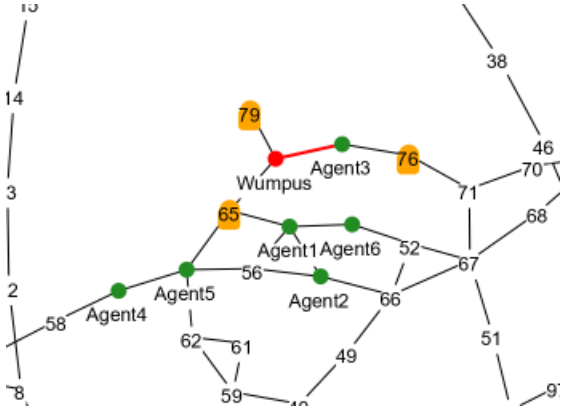

Table des matières

1 Introduction

Il ne fait nul doute qu'à l'avenir, l'utilisation d'agents intelligents dans nos villes se fera de plus en plus commune. Flottes de robots livreurs, nuées de drones de surveillance... difficile d'imaginer cependant quelle sera sa forme exacte, mais une chose est sûre : ces agents devront agir en coordination pour mener à bien leurs missions.

Dans le cadre de notre cours « Fondements des Systèmes Multi-Agents » de première année de master d'informatique de Sorbonne Université, nous avions la tâche de concevoir des agents intelligents, dont le but est de pièger un *Wumpus* (golem, agent adversaire) sur un graphe non-orienté. Nous nous sommes appuyés sur le projet *Dédale* https ://dedale.gitlab.io/. Ce rapport fait état de notre réalisation de ce projet.

2 Architecture

Le fonctionnement des agents s'appuie sur un certain nombre de *comportements*. Définis à l'avance, l'ordre de leurs exécutions en revanche peut varier et poser un problème, s'ils ne sont pas adaptés. Pour un contrôle total de nos agents, nous avons opté pour la mise en place d'un automate à états finis comme sur la figure 2.1, qui déterminera quel comportement s'exécutera à chaque moment donné. La conceptualisation d'un tel automate nous a permis en outre un découpage naturel des fonctionnalités en comportements cohérents, grâce au maintien en parallèle d'un graphe récapitulatif, mettant en exergue les relations entre ces derniers.

Après de nombreuses versions (que nous ne vous montrerons pas par souci de concision), nous sommes arrivés à la figure 2.2 à un automate permettant l'exploration et la chasse, l'une après l'autre.

3 Exploration

De base, notre version étudiante du projet Dédale incluait une classe MapRepresentation faisant guise de représentation mentale du graphe des agents. Fournie par nos encadrants, elle n'était pas satisfaisante car nous n'en avions pas la maîtrise ; nous nous heurtions donc parfois à des comportements difficiles à débugger. Elle permettait –entre autres– d'observer en temps réel l'évolution de la connaissance des agents sur la topologie du graphe, via une fenêtre externe. Nous apercevant que cette information était superflue, nous avons opté pour l'écriture d'une nouvelle classe MiniMap qui, certes, ne possède pas cette fonctionalité, mais remplit néanmoins toutes les fonctions utiles à

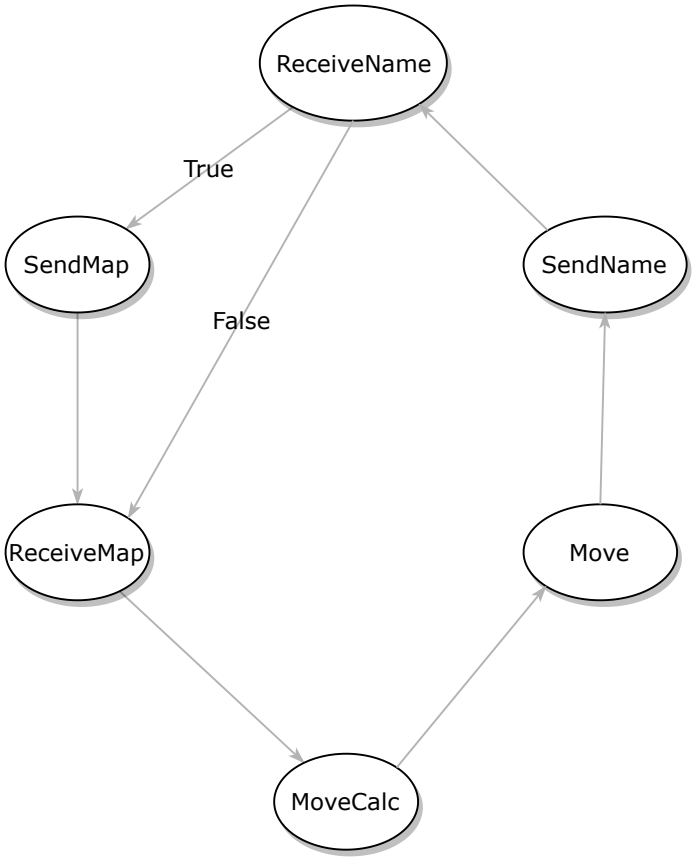

FIGURE 2.1 – Premier automate à états finis

l'exploration que MAPREPRESENTATION fournissait jusqu'ici, en plus de nous permettre d'ajouter nos propres algorithmes sans avoir à contourner le code original.

Pour explorer la carte, l'idée est donc d'enregistrer dans la MINIMAP les nœuds rencontrés, et ce, jusqu'à ce qu'il n'y ait plus de nœud ouvert (il s'agit de la phase *Explore* de l'automate de la figure 2.2). Pour accélérer l'exploration, nos agents envoient régulièrement leurs noms aux agents proches, dont ils ont connaissance grâce aux pages jaunes. Ensuite, ils écoutent pour recevoir d'éventuels noms envoyés par leurs collègues. S'ils en reçoivent, les agents s'envoient alors leurs cartes respectives. En recevant une carte, un agent fusionne celle-ci avec sa propre carte pour avoir le maximum d'information sur la topologie du graphe sur lequel ils évoluent.

L'utilisation des noms permet de réduire le nombre de messages volumineux envoyés. D'une part, c'est mieux que d'envoyer toute la carte à chaque itération, alors qu'il n'y a probablement personne pour la recevoir. D'autre part, en recevant un ou plusieurs noms d'agents alliés, ceux-ci ont alors la connaissance précise des destinataires de leurs cartes. Il ne faudrait pas envoyer la carte au(x) *Wumpus* !

Dans la phase *Explore*, c'est le comportement *MoveCalc* qui détermine le prochain nœud vers lequel se déplacer. Si la carte n'est pas complètement explorée, et qu'un nœud est ouvert et directement accessible depuis la position actuelle, le comportement envoie ce nœud au comportement *Move*, qui se charge de déplacer l'agent. Sinon, il demande à la MINIMAP de lui renvoyer *les deux nœuds ouverts les plus proches*. Trouvés par parcours de graphe en largeur, il en choisira un aléatoirement. C'est une façon simple de régler le problème des *interblocages*, mais également d'éviter aux agents de se diriger vers le même nœud après avoir échangé leurs cartes. Bien-sûr, cette technique n'est pas optimale,

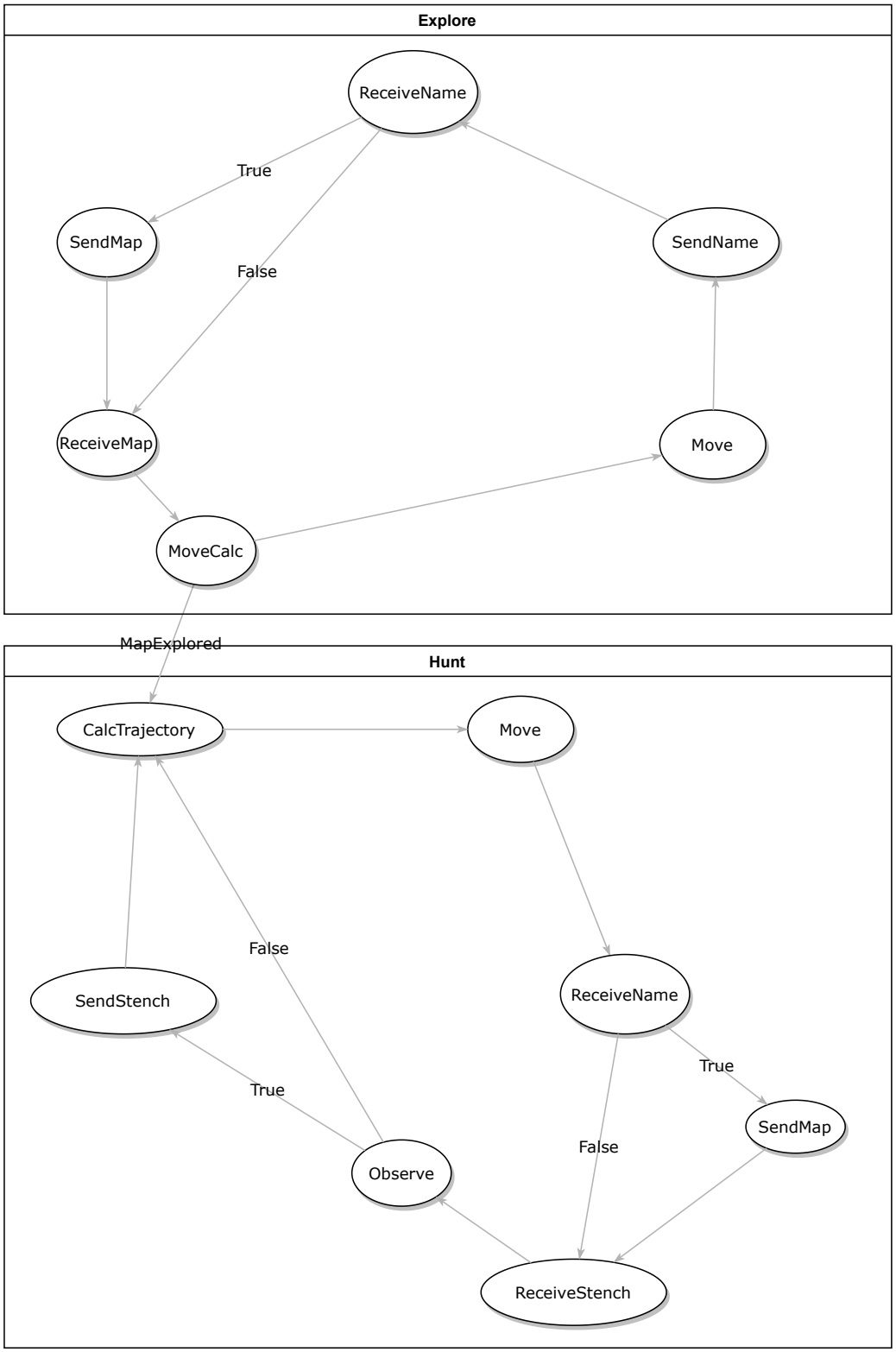

FIGURE 2.2 – Version finale de l'automate

mais a le mérite de fonctionner et d'être simple. Les inconvénients sont que, lors d'un interblocage, il faut attendre quelques itérations avant que l'agent choisisse aléatoirement un autre nœud vers lequel se diriger. Aussi, lorsque les deux nœuds ouverts les plus proches sont à équidistance de l'agent, on peut observer l'agent s'avancer vers l'un des deux nœuds, pour aller vers l'autre nœud l'instant d'après. Alternant ainsi entre deux positions pendant plusieurs itérations, celà donne cette impression que l'agent « hésite », ce qui, certes, ajoute au réalisme, mais n'a aucun intérêt pour la chasse au *Wumpus*.

4 Chasse

Une fois la carte explorée pour un agent, ce dernier passe en phase *Hunt* (cf. figure 2.2), sans attendre ses collègues. Dans l'éventualité qu'un agent allié soit encore en phase d'exploration, un agent en phase de chasse est toujours prêt à envoyer sa carte à quiconque lui demande, via l'envoi de son nom, le faisant passer directement en phase de chasse.

Notre stratégie d'encerclement coopératif se base sur les *Stenches* (odeurs dégagées par le *Wumpus*) et la coordination des agents autour du *Wumpus*. Dans cette optique, l'agent se tient prêt à recevoir des positions de *Stench* de la part de ses alliés, ainsi que les positions de ceux-ci. Les positions de *Stench* seront stockées dans une *Map* associant à chaque nom d'agent un ensemble de positions de Stenches reçues de sa part, et les positions des agents dans une *Map* associant à chaque nom d'agent allié sa dernière position connue (reçue).

Pour régler le problème de la fraîcheur des informations reçues, nous avons appliqué le principe selon lequel nos propres informations seront toujours plus à jour que celles reçues par des tiers. Ainsi, après avoir obtenu d'éventuels *Stenches* et positions des autres agents, le comportement Observe a pour mission de mettre à jour la connaissance propre de l'agent quant aux positions des *Stenches*, mais aussi celles rapportées par les autres agents. Autrement dit, si l'on observe que des nœuds possèdent des *Stenches* mais n'apparaissent pas dans les connaissances personnelles de l'agent, ils sont alors ajoutés. Au contraire, s'il est observé qu'un nœud ne possède aucune *Stench*, celui-ci est retiré de toute connaissance, qu'elle soit personnelle ou rapportée.

Par la suite, si de nouvelles *Stenches* ont été découvertes, elles sont envoyées aux agents à proximité (en plus de la position de l'agent). Notons que seule la connaissance propre de l'agent est envoyée. Envoyer les connaissances rapportées par les autres agents serait redondant, et en cas de cycle de communication, une connaissance pourrait circuler d'un agent à l'autre sans pourtant être vraie et sans jamais être remise en cause.

La décision quant à la trajectoire à suivre est prise par le comportement astucieusement nommé *CalcTrajectory*. L'agent inspecte en priorité son propre ensemble de *Stenches* observées (principe de fraîcheur), puis, s'il n'y en a pas, celles rapportées par ses pairs. Si une *stench* est connue, alors il s'y rend. Enfin, en l'absence de connaissance de la position des *stench*, ce comportement transmet au comportement de déplacement *Move* une « patrouille », calculée par MiniMap. Cette patrouille est la liste de tous les nœuds ayant un degré supérieur à 1, et constitue donc un recouvrement du graphe. En effet, les agents ayant un rayon de vision égal à 1, il est inutile de visiter un nœud s'il n'apporte pas de nouvelle information.

5 Coordination

Les agents ont donc pour objectif simple de se diriger vers et d'occuper les nœuds contenant des *stenches*. De cette manière, si toutes les *Stenches* – qui sont censées entourer le *Wumpus* – sont occupées par des agents, alors le *Wumpus* est piégé. Supposons à présent qu'un agent que nous nommerons Agent2 se trouve sur une *Stench* en position 4 (cf. figure 4.1). Il communique aux agents à proximité ses *Stenches* connues (en l'occurence, la position 5) et sa position (position 4). Imaginons alors qu'un agent nommé Agent1 passe à proximité de Agent2. Agent1 a à présent connaissance de

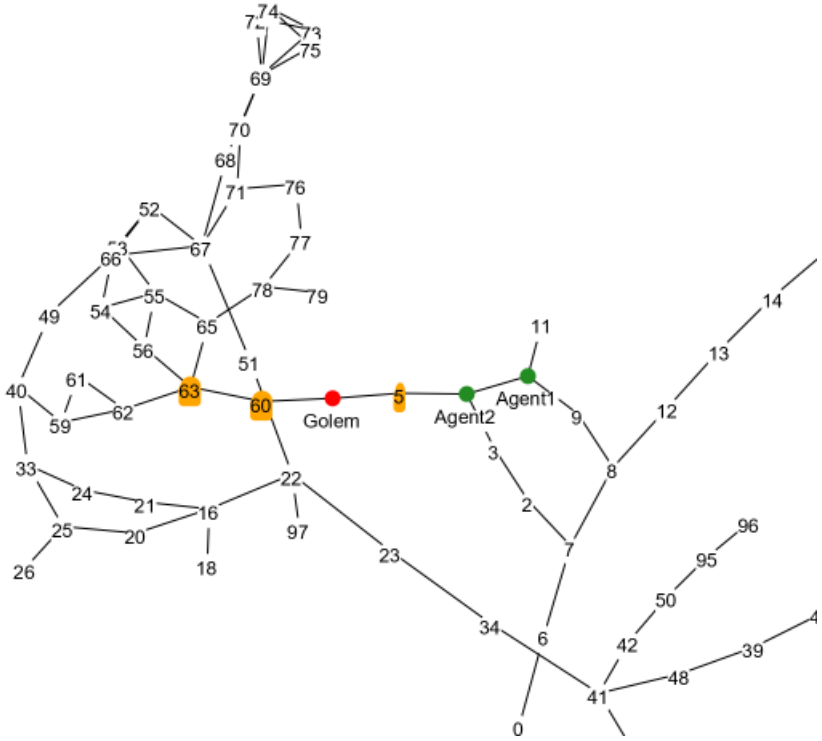

FIGURE 4.1 – Cas de bloquage d'un agent par un autre à proximité du *Wumpus*

Stenches envoyées par Agent2, mais comment se rendre en position 5 si Agent2 bloque le passage ? C'est ici que savoir quelle est la position des alliés intervient. Pour se rendre à une *Stench* (ou tout autre objectif), les agents font appel à MINIMAP en lui demandant explicitement de ne pas inclure dans la trajectoire les positions des alliés. Cela a pour effet de forcer l'agent à trouver des passages alternatifs (ça ne se voit pas sur la figure 4.1 car l'image est tronquée, mais il y a un passage en allant au delà du nœud 14 et qui revient en 39). Dans les cas similaires à la figure 4.1, ceci résulte en la prise du *Wumpus* en tenaille, s'il ne s'est pas enfui avant ! S'il tient la cadence, l'Agent2 va pour sa part suivre le *Wumpus* en se dirigeant toujours vers les *Stenches* à proximité.

6 OPTIMISATIONS

La question d'une communication efficace est majeure pour ce projet de coordination. Aussi, nous fûmes exaspérés de voir nos agents s'envoyer un nombre excessif de copies du même message dès qu'ils se croisaient. En effet, sans l'utilisation de protocole élaboré de communication, il n'y a aucun moyen pour les agents de se rendre compte si un message envoyé est arrivé à bon port. Portés par le principe d'*Émergence*, mais aussi pressés par le temps, nous nous sommes dit qu'une solution simple mais efficace serait suffisante pour régler ce problème. Nous avons donc tout simplement fixé un délai entre deux messages envoyés que les agents doivent respecter. C'eut pour conséquence de réduire drastiquement le nombre de messages circulants sur le réseau.

7 Conclusion

Avec notre stratégie de chasse coopérative simple, nos agents arrivent dans la plupart des cas à capturer le *Wumpus*. Toutefois, celle-ci ne se fait que dans des conditions particulières. Par exemple, observons le cas de la figure 7.1. Si le *Wumpus* (nommé *ImHere* ici) a un rayon d'odeur de 2, les agents, eux, s'arrêtent à la première *Stench* la plus proche, laissant de la place au *Wumpus* pour circuler entre les agents. S'il existe une sortie pour le *Wumpus*, en passant par *target* puis le nœud 49 dans ce cas, il peut s'enfuir facilement. Il aurait pourtant suffit à l'*Agent2* de s'avancer en *target* pour l'en empêcher.

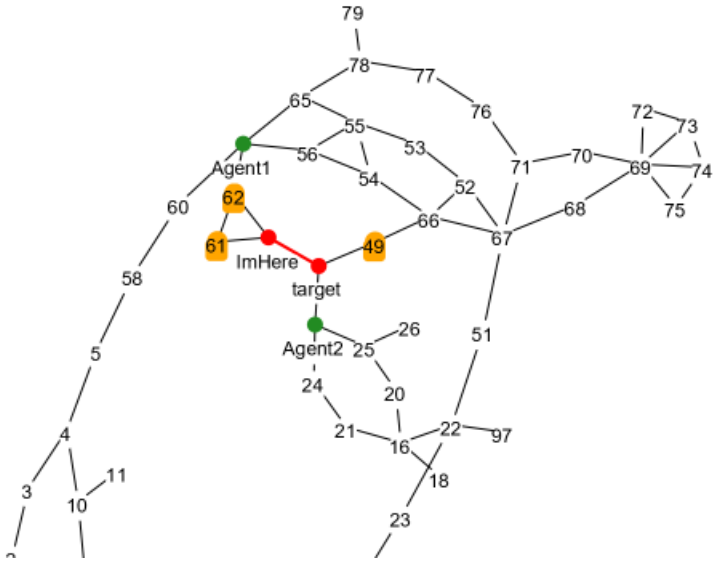

Figure 7.1 – Cas de fausse capture du *Wumpus*

Avec plus de temps, nous aurions voulu mettre en place une manière de détecter le *Wumpus* de façon plus explicite. Le *Wumpus* est toujours entouré de *Stenches*, ce qui signifie qu'il y a au moins un nœud adjacent aux *Stenches* dont tous les voisins sont des *Stenches*. Il est donc possible de déterminer la position du *Wumpus* en prenant l'intersection de l'ensemble des nœuds voisins aux *Stenches* connues (et en y soustrayant les positions de *Stench* bien-sûr). La détection explicite du *Wumpus* aurait réglé le problème de sa « marge de manœuvre ». Il aurait aussi fallu que les agents puissent comprendre si le *Wumpus* peut s'enfuir, en connaissance de la topologie de la carte et de la position supposée de ce dernier. Nous aurions également voulu remplacer notre façon de gérer les interblocages par une méthode moins aléatoire.

Enfin, on peut remettre en cause la structure exploration *puis* chasse, surtout dans des cas réels. Nous avons mis à l'épreuve une trentaine d'agents dans une reproduction des rues d'une ville. Nous avons alors constaté les limites de l'exploration préalable, car dans des graphes aussi grands et complexes, celle-ci peut se faire en beaucoup de temps, d'autant plus que les agents se croisent beaucoup moins. On peut se rassurer toutefois en supposant que dans un cadre réel, par exemple où une flotte de robots policiers essaient de piéger un malfaiteur, ceux-ci connaissent la carte de leur ville à l'avance.

8 Remerciements

Nous remercions Aurélie Beynier, Cédric Herpson et Nicolas Maudet pour leurs enseignements et ce projet enrichissant.