Compte Rendu

Fondements des Systèmes Multi-agents

# Recherche de trésors par Multi-agents ( Dédale )


*Binôme :*

Qingyuan Yao
Jules Cassan

*Encadrant :*

Aurélie Beynier
Cédric Herpson
Nicolas Maudet



Github : `https://github.com/MRVNY/Fosyma`

# Sommaire

# 1 Introduction

Le but de ce projet est d'explorer les possibilités et les mécanismes des systèmes multi-agents. On utilisera l'environnement Java, **Dédale**, implémentant déjà de nombreux outils pour le fonctionnement des agents tel que les communication, le déplacement ainsi que la récolte de ressources sur une carte. Notre objectif est de faire travailler nos agents ensembles et de coordonnée leurs action dans le but de récolté le maximum de ressources dans le labyrinthe tout en maintenant une équité entre les ressources récupéré par chaque agents. Tout cela en limitant leurs communications aux informations et avec la présence de Wumpus, des agents concurrents qui ne sont présents que pour perturber le travail de nos agents.

# 2 Déplacement

## 2.1 Modes

Il existe 3 modes de déplacement : **EXPLORE**, **LOCATE**, **SEARCH**, en chaque mode, les agents ont le but différent(cf.4.1).

1. **EXPLORE** : C'est le mode de déplacement à l'initialisation. Il permet de donner les directives d'explorations de la carte afin de complété leurs connaissance de leur carte ainsi que de rencontrer d'autres agents afin d'échanger des informations.
2. **LOCATE** : C'est le mode de déplacement de la recherche de trésors. Une fois les types des agents attribué, il doivent rejoindre le noeud correspondant à leur but qui peux évoluer en fonction des obstacle comme le wumpus et les rencontre avec d'autres agents ayant le même objectif.
3. **SEARCH** : C'est le mode de déplacement pour terminer l'exploration de la carte et de fin des agents. Dans ce mode de déplacement, les mouvements sont aléatoires mais donne la priorité aux autres agents souhaitant se déplacé. Le but est de continuer de répartir de l'information sans entraver les mouvement des autres agents alliés.

## 2.2 Transition entre les modes

L'exploration de la carte est la première étape du comportement de nos agents, ils sont tous en mode **EXPLORE**. Le but est de créer la meilleure représentation de toute la carte à tout nos agents le plus rapidement possible pour avoir l'idée la plus précise de la position de chaque trésors et quel est la quantité total de trésors que nos agents auront à récupérer. Ses informations sont primordiales pour le bon fonctionnement du partages des agents sur les ressources et pour le maintient de l'équité. Les agents ont donc pour consigne de navigué sur l'ensemble de la carte

et de complétée tout les noeuds ouvert de leur représentation mental ( L'objet *MapRepresentation* ). Pour accélérer ce processus, les agents peuvent communiquée entre eux l'état de leur carte. Une fois la phase d'exploration terminée, les agents passe en mode **LOCATE** et navigue sur la carte en fonction de leur But. Chaque agent à un but ( une simple string indiquant l'ID d'un noeud de la carte ) qui représente un noeud objectif. Les agents cherche le trajet le plus court vers leur but et continue jusqu'à passer en mode **SEARCH** lorsque leur sac est plein ou qu'ils ont récupéré la quantité de ressources qu'il leur à été attribué. Ils effectue des mouvement aléatoire sur la carte pour continuer l'exploration et mettre à jour les valeurs des trésors afin de venir en aide aux agents qui n'auraient pas fini leur travail en mode **LOCATE**. Si la carte est mise a jour et que l'agent ce vois attribuer le but d'un nouveau noeud de récolte alors il repasse en mode **LOCATE**.

# 3 Communication

Nos agents dispose de moyen de communication déjà implémenté avec **Dédale**. Cela permet aux agents d'envoyer des messages et d'en recevoir en fonction de la portée de communication. A chaque déplacement, les agent envoie un **Ping** pour établir un dialogue avec les agents voisins qui contient un objet de communication que nous avons crée pour contenir toutes les informations nécessaires ( l'objet *Message* différent de *ACLMessage* mais ne le remplace pas). Si il n'y a pas d'agents dans les environs donc pas de réponse sous la forme d'un **Pong**, l'agent reprend son chemin. Si il reçois bien un **Pong** comme réponse, il renvoie un message **End** afin de bien confirmer que le message **Pong** à été reçu et la communication est terminée. Les agents doivent toujours attendre un cours instant pour être certain que aucun agent ne leur répond.

# 4 Coordination

## 4.1 Priorities & Goal

Pour bien coordonner les agents, il faut chercher le noeud intéressant sur la carte selon leurs modes (cf.2.1), On appelle ce noeud **"goal"**(but).

- **EXPLORE** : Aller sur le noeud ouvert le plus proche
- **LOCATE** : Aller sur le trésor qui correspond au rôle d'agent et qui satisfait l'équité entre les agents (cf.7)
- **SEARCH** : Aller sur le noeud aléatoire afin de trouver les informations ignorées

Idéalement, le goal est le choix optimal de l'agent. Cependant, en cas de conflits(cf.4.2), on doit laisser une possibilité de passer aux autres choix moins optimales. On appelle cet ensemble de choix (triés selon optimalité) **"priorities"** (liste de priorités). On utilise 3 fonctions différents dans MapRepresentation pour les 3 modes :

- **EXPLORE** : getClosestOpenNodes (triée par la distance)
- **LOCATE** : getClosestTreasuresOfClosestValue (triée par l'équité et la distance (cf.5))
- **SEARCH** : getAllNodes (randomisée)

## 4.2 Négociation

En suivant les mêmes règles de coordination, il est normal que parfois les agents ont le même goal ou se marchent sur les pieds. Un algorithme de négociation est mis en place lors d'un échange de message(cf. 3). La négociation se déclenche quand l'un des ces deux conflits est rencontré :

- **Conflit de goal** : les deux agents ont le même goal
- **Conflit de chemin** : les deux agents ont le même prochain pas ou les deux agents vont à la position déjà occupée par l'autre agent

### 4.2.1 Hiérarchie des modes

Quand les deux agents sont en modes différents, il existe une hiérarchie des modes pour prioriser les déplacements. Les modes sont en ordre :

**LOCATE** > **EXPLORE** > **SEARCH**

Cela veut dire que les agents en **LOCATE** sont priorisés que ceux en **EXPLORE**, et ceux en EXPLORE sont priorisés que ceux en **SEARCH**. Pendant la négociation, l'agent on mode inférieur compromise toujours son chemin ou son goal pour l'agent en mode supérieur. Cette hiérarchie garantit l'efficacité de la collection.

### 4.2.2 Compromis

Lors d'un compromis, l'agent d'abord abandonne son goal initial. Ensuite il sort sa liste de priorités(cf. 4.1) afin de la parcourir. S'il existe un noeud dans la liste qui ne pose pas de conflits précédents(cf. 4.2) et respecte le chemin de l'agent en mode supérieur, l'agent prendra ce goal. **L'algorithme arrête quand l'agent trouve un goal valide ou quand il finit parcourir sa liste de priorités.**

A cause de l'interdiction du contrôler les autres, l'agent ne peut faire compromis que à sa côté pendant la négociation. Cependant, grâce à l'échange (idéalement) bidirectionnel avec **Pong** et **End**, les deux agents peuvent faire une négociation chacun afin d'éviter le blocage ou le conflit de goal.

### 4.2.3 Contraintes

Cet algorithme est assez coûteux puisqu'il doit recalculer le longueur de chemin pour chaque noeud dans la liste de priorités, et la fonction pour le calcul de chemin (getShortestPath) qui utilise l'algorithme Djikstra devient lent quand la taille de la carte augmente.

Complexité = nb_de_priorites * tailles_de_la_carte

Manque d'une implémentation de communication tridirectionelle, on ne peut pas permettre aux 3 agents ou plus de faire la négociation ensemble. A cause de l'exécution asynchrone, les informations des agents peuvent tardif, qui amène à une négociation erronée. Ces contraintes peuvent (rarement) amener à un blocage, pour l'éviter, on a créer aussi un mécanisme de déblocage(cf.4.3).

## 4.3 Déblocage

Quand un agent reste sur un noeud pendant certains itérations, le mécanisme le déblocage se déclenchera. Pendant des itération donnée, l'agent bougent aléatoirement en excluant le noeud vers son chemin destiné qui lui bloque précédemment. Ce mécanisme est devenu de moins en moins utilisé au fur et à mesure qu'on optimise le calcul d'équité(cf. 7), de chemin(cf. 4.1) et l'algorithme de négociation(cf. 4.2).

# 5 Collecte

La collecte des agents ne peux être effectué que lorsqu'ils sont en mode **LOCATE**, dans le reste des modes, si ils croisent une ressources, ils mettent à jour leur carte en ajoutant un trésor ou en mettant la valeur de ce dernier à jour. Une des règle du projet était qu'une fois une ressources récupéré par un agent ( Or ou Diamant ), ce dernier ne peux plus récupérer un autre type de ressources. Donc, chaque agent ce vois attribuer un type qui nous permet de veiller à ce qu'aucun agents ne transgresse au protocole.

Chaque agent est assigné à un type au moment où les agents décide de la répartition des rôles et de l'équité. Un agent cherche à collecter une quantité spécifique de ressources pour ne pas tout prendre. Cette valeur est calculé au même moment de la répartition des rôles et mis à jour en fonction des changements de la carte. Les agents cherchent les ressources en fonction : de leur type, de leur proximité et de leur valeur ( cela est réalisé par la méthode *getClosestTreasuresOfClosestValue()* ).

Mais une des limite de ce système est que les agents risques de ce précipiter sur les même ressources, celle qui ont les valeur plus importante ( en effet, la méthode

cité précédemment favorise un trésors qui a une valeur identique ou proche de celle que l'on cherche, donc les agents vont vouloir les trésors avec le plus de ressources ). Donc, une solution pour remédier au problème et diminuer les chances de conflits pour un trésor, les agents ont une chance sur deux d'aller chercher un trésor d'une valeur inférieure à celle qu'il cherche en définitive.

# 6 FSM

Afin de bien gérer les transitions entre les actions, on utilise le machinisme de FSM. On commence par le State Move, qui nous permet le déplacer d'un noeud vers un autre suivant les règles de déplacement (cf. 2). Après avoir déplacé, l'agent essaye d'envoyer un **Ping** à tous les autres agents, et il entre dans le State Check.

## 6.1 Check

Dans le State check, l'agent vérifie sa boîte de lettre et son environnement pendant un nombre d'itération définie, et il réagira selon les informations obtenues.

- Si l'agent reçoit un Ping, il passera à Send_Pong pour envoyer un **Pong**, et il reviendra à Check. Il mettra aussi un Timer pour attendre l'envoie de End.
- Si l'agent reçoit un **Pong**, il lira le message pour fusionner la carte reçue avec sa propre carte et pour faire une négociation(cf. 4.2) s'il détecte un conflit, à la fin, il enverra un **End** et il reviendra à Check
- Si l'agent reçois un **End**, il lira le message pour fusionner et faire un négociation comme action précédente, et il reviendra à Check sans rien envoyé.
- Si l'agent trouve un trésor sur la carte et qu'il est en mode **LOCATE**(cf. 2). Il passera à Decide pour Decider s'il va collecter le trésor(cf. 6.2).
- Si l'agent ne trouve rien intéressant, il bouclera sur Check jusqu'où il atteint son itération maximum.
- Après avoir atteint son itération maximum et avoir épuisé son Timer pour attendre les End, l'agent revient au déplacement.

## 6.2 Decide & Collect

Quand agent arrive sur un trésor et qu'il est en mode **LOCATE**, il vérifiera :

- Si le trésor correspond bien à sa rôle
- S'il y a assez de place dans son sac pour collecter le trésor
- Si le montant de trésor est bien ce qu'il cherche ou s'il l'endroit de trésor est bien son goal(cf. 4.1).

Si tous critères sont bien valides, l'agent passera au State Collect où il collecte le trésor et passera au Send_Ping pour informer tous le monde aussitôt qu'il a collecté le trésor à sa position. Sinon il reviendra à Check.

### 6.3 Graphe

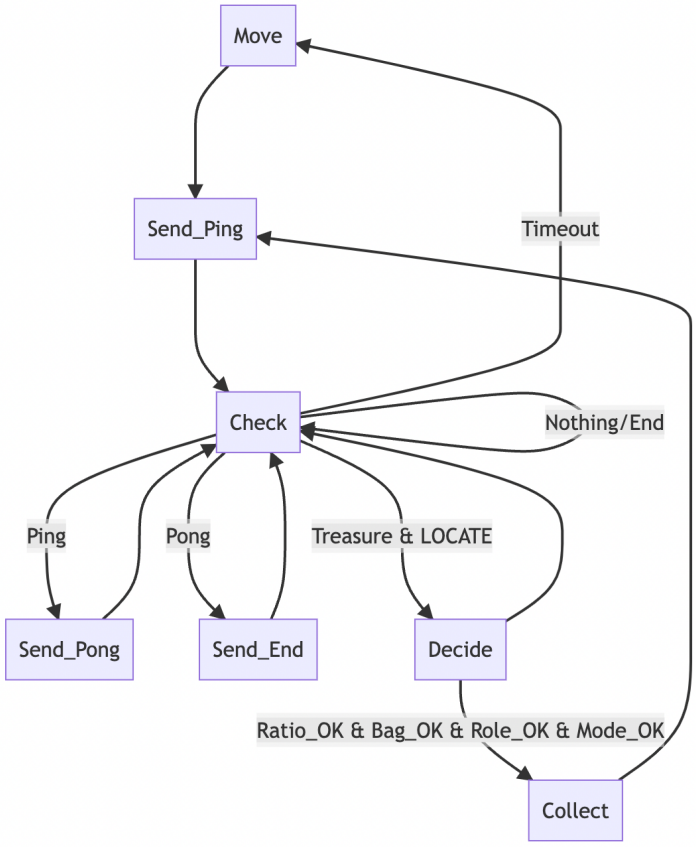


FIGURE 1 – Graphe FSM

## 7 Mécanisme d'équité

La répartition équitable des ressources peux être interprété de différentes façon. En effet, il est difficile de savoir quand on dois privilégier la quantité de ressources globale récupéré ou bien le fait que chaque agents possède à la fin une quantité équivalente de ressources. Pour nous, le but principal reste de récupérer un maximum de ressources à la fin. Dans le cas parfais, en répartissant nos agents correctement entre les deux ressources à collecter, il existe une valeur alpha qui est la valeur que tout les agents doivent avoir à la fin de la récolte et qui correspond à la quantité de cette ressource disponible sur toute la carte divisée par le nombre d'agents attribuée

à la récolte de cette ressource. Par exemple, si il y à 200 d'or répartis sur la carte, et que l'on attribue 5 agents à la ressource OR, alpha aura pour valeur $200/5 = 40$. Donc l'objectif de nos agents sera de récolter 40 d'or.

Mais on veux que cette valeur sois dynamique car si une nouvelle cache d'or est découverte plus tard dans l'exploration, la valeur alpha dois elle aussi changer. Il se peux également qu'un agent est une capacité de stockage plus faible que la valeur alpha. D'où l'émergence d'un autre problème de conception, faut-il limiter la collecte des autres agents afin de conserver une équivalence entre les agents ?

## 7.1 Problème de sous-capacité

Un problème récurent est la sous-capacité de nos agents. En effet, dans certain cas, nos agents n'auront pas forcément la capacité de tout récupérer car leur capacité total est inférieure à la quantité de ressource répartie sur la carte. Cela veux dire que pour récupérer le plus de trésors, tout les agents devrais avoir une quantité de ressource égale à leur capacité individuel qui n'est pas forcément équivalente. Donc si on souhaite privilégier l'équité, on dois limité la valeur de collecte de tout les agents liée a cette ressource.

## 7.2 Consensus Local

Cette partie est basé sur le code de la classe *EquityModule*. Pour résoudre le problème d'équité, notre représentation de la carte ainsi que toutes nos interactions avec les autres agents vont permettre à chaque individu de réalisée un consensus local, un consensus qui n'est donc pas dirigée par un proposeur et des accepteurs comme dans l'approche du *Paxos [Lamport,2001] Approche « Best Effort »*. Donc une des limites de notre approche est que notre environnement est asynchrone et que dès lors que un de nos agents ne possède pas toutes les bonnes informations il risque prendre une mauvaise décision. Donc pour obtenir le meilleur consensus possible, chaque agents dois favorisée le partages d'informations comme la capacité initial de chaque agents, la quantité de ressources sur toutes la cartes, etc.

Le but est donc que chaque agents choisisse la ressources où il pourra être le plus profitable et le moins limitant pour maintenir l'équité. Il a été décidée que l'équité primait sur la quantité de ressources globale récupérée mais cela peux être changée par un boolean ABSOLUTE_EQUTY valant true si l'équité est le plus important, false sinon. Le Principe pour l'attribution des types est le suivant : on parcours les deux listes des capacités de nos agents ordonnées celons leurs capacité d'or et de diamant ( donc le premier élément de chaque liste est le nom de l'agent avec la meilleur capacité de stockage d'une des ressources ) puis on retire le premier élément d'une des deux listes ( on retire également le nom de l'agent de l'autre liste ) et on

décide que cet agent est maintenant attribuée à la ressources qui lui correspond le mieux. On met à jour la valeur de capacité disponible pour la ressources.On répète l'opération jusqu'à ce que les liste sois vide et l'attribution des type est donc faites et on dois maintenant calculer la quantité de ressource que chaque type vas devoir collecter.

On calcul donc la répartition par le calcul suivant : quantitéAChercher = TotalDeRessourcesDuType / nombreAgentsAttribueDuType. Par exemple, si 3 agents ont été attribué à la collecte d'or et qu'une quantité de 120 d'or à été aperçu sur la carte, alors chacun des agents dois récupérer 40 d'or. Mais cette valeur peux être mise a jour avec la découverte d'un nouveau trésors et est donc recalculé à chaque mise à jour de la carte. De plus, On dois veiller a ce que cette valeur ne sois pas plus grande que la pire capacité de stockage d'un agent assigné à la ressource. Dans l'exemple précédent, si un des agents possède une capacité de seulement 30, il ne pourra pas atteindre la quantité demandé, donc pour que les objectifs sois les même, tout les agents assignée à la ressources en question doivent revoir leur valeur à la baisse. La valeur de recherche est donc dynamique et évolue en fonction des trésors trouvée et est robuste aux caractéristiques donné initialement aux agents.

La complexité de ce processus dépend du nombre total d'agents n, et du nombre de mise à jour du but/de la carte mental de l'agent. En effet on parcours deux liste simultanément n fois et le calcul de la valeur de recherche dépend des valeurs échangée entre les agents et les trésors trouvé lors de l'exploration. Une des limites de ce système est également qu'il n'est pas possible de choisir la quantité de ressources que l'on récupère au sol car la fonction pick() ne le permet pas.

## 7.3 Extension possible :Résolution par programmation linéaire

Nos agents possèdent des capacité de stockages différentes, et pour garantir que l'on récupère la quantité optimal de ressources on peux représenter le problème par un programme linéaire. On modélise le problème de la façon suivante :

Nos variables sont la capacitée de stockage de chaque agent pour nos deux ressources, AD et AG pour l'or et le diamant et suivit du numéro de l'agent n.
Max capD1*AD1 + capG1*AG1 + capD2*AD2 + capG2*AG2 + ... + capDn*ADn + capGn*AGn
AD1 + AG1 = 1
AD2 + AG2 = 1
...
ADn + AGn = 1

Les contraintes permettent savoir quelle capacité pour chaque agent permet de

maximisé la récupération de ressources. Avec ce programme linéaire, On peux déterminer quel agents doit récupérer quel ressources afin de maximiser la capacité générale de chaque ressources. La résolution par programme linéaire semble la meilleur approche pour le problème mais n'a pas été implémenté pour des raison d'implantation du solveur ( nous avons tenté avec Java mais des problèmes de licence sont survenue et nous préférions nous concentrer sur d'autres problématiques plus en lien avec les système multi-agents ). Nous avons donc opté pour une solution plus simple.

# 8 Conclusion

Pour conclure, Ce projet montre que les systèmes multi-agents sont très **modulable** en fonctions des attentes ( par exemple pour la répartition des ressources et comment on définis l'équité du système ). Notre approche semble fonctionner correctement et est suffisamment robuste aux changements de paramètres extérieurs comme les configurations des agents et la répartition des ressources.

Mais, le système n'est pas correctement optimisée sur la communication des agents ainsi que sur le parcours de la carte. En effet, malgré des effort sur la fusion des cartes et le partage des connaissance, **les messages sont lourds et envoyés très fréquents**. Le déplacement lui, repose sur la méthode de **Djikstra**, mais dont l'implémentions est présentée comme peu efficace dans le parcours de la carte. Pour améliorer ces aspects, il serais possible de mieux implémenter la recherche de plus cours chemin.

Il serais aussi possible d'améliorer les interactions de nos agents avec les Wumpus qui dans certains cas provoque un attroupement des agents et peu parfois aboutir à un blocage définitif de ces derniers. Mais un des ajout les plus intéressant serais l'ajout d'un solveur comme **Gurobi** afin de résoudre le problème d'équité par programme linéaire.