Sorbonne Université
Faculté des Sciences
Master Informatique - Parcours ANDROIDE

FOSYMA - Rapport Projet Dedale

Systèmes Multi-Agents pour la Collecte de Ressources

UE de Projet M1

Victor FLEISER – Thomas MARCHAND

2025

Table des matières

1 Introduction

1.1 Contexte

Ce projet vise à développer une solution pour la collecte des ressources avec un système multi-agents dans un environnement inconnu et dynamique, représenté par un graphe. Nous utiliserons un environnement Dedale, basé sur la plateforme Jade, pour implémenter les agents et leurs interactions.

1.2 Objectifs

L'objectif est de concevoir et d'implémenter un SMA capable de :

— Explorer collaborativement un environnement inconnu pour construire une carte partagée.

— Localiser les trésors.

— Coordonner les actions des agents pour collecter efficacement les trésors découverts, en tenant compte de leurs capacités et des contraintes.

— Robustesse pour faire face aux interblocages et perturbations du Golem.

1.3 Architecture générale

Nous commençons avec une architecture homogène pour l'exploration, qui évolue vers une architecture centralisée pour l'exploitation des trésors. Voici les principaux types d'agents :

Agent Collecteur [**1+**] Participe à l'exploration, puis reçoit des missions du Silo pour collecter ou ouvrir des trésors.

Agent Silo (Tanker) [**1+**] Participe à l'exploration, puis l'agent nommé "Silo" devient le coordinateur central pendant la phase d'exploitation. Il centralise l'information et détermine la stratégie de collecte puis donne des missions aux autres agents.

Agent Explorateur [**0+**] Participe à l'exploration, puis reçoit les missions d'exploration du Silo pendant la phase d'exploitation.

Agent Golem (Wumpus) [**0+**] Agent qui bloque le passage des agents et peut déplacer les trésors.

2 Structure du Code

2.1 Organisation

— `mas.agents.newAgents` : Classes d'agents
  — `ExploreCoopAgent.java`
  — `SiloAgent.java`

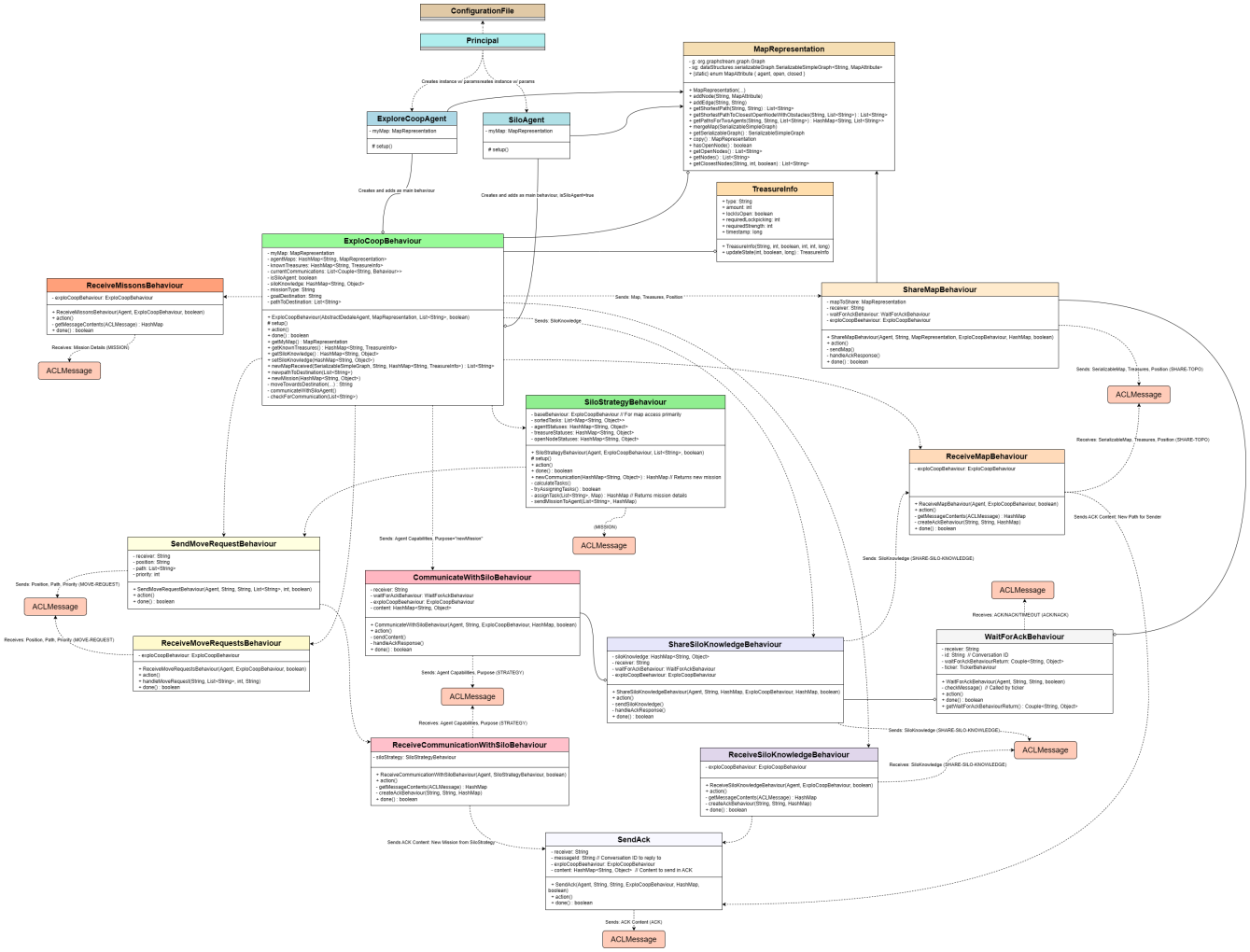
— `mas.behaviours.newBehaviours` : Behaviours pour la logique des agents
  — `CommunicateWithSiloBehaviour`
  — `ExploCoopBehaviour.java`
  — `ReceiveCommunicationWithSiloBehaviour.java`
  — `ReceiveMapBehaviour.java`
  — `ReceiveMissonsBehaviour.java`
  — `ReceiveMoveRequestBehaviour.java`
  — `ReceiveSiloKnowledgeBehaviour.java`
  — `SendAck.java`
  — `SendMoveRequestBehaviour.java`

  - ShareMapBehaviour.java
  - ShareSiloKnowledgeBehaviour.java
  - SiloStrategyBehaviour.java
  - WaitForAckBehaviour.java
- mas.knowledge : Structures de données pour les connaissances
  - MapRepresentation.java
  - TreasureInfo.java
- princ : Configuration et exécution du code
  - ConfigurationFile.java
  - Principal.java

Notes pour l'exécution :

- Nous utilisons une stratégie centralisée organisée par 1 agent Silo. Dans notre code, nous l'avons défini comme l'agent ayant le nom "Silo".
  Il faut donc que exactement 1 Agent Tanker ait comme agentName "Silo" lorsque les agents sont créés.
- Pour distinguer les différents types d'agents au sein du code nous utilisons leur capacités de stockage pour l'or et les diamants, il faut donc que les capacités des agents respectent ces règles lorsque les agents sont créés :
  - Agent Collecteur : une des 2 capacités est strictement supérieure à 0, l'autre est à -1
  - Agent Tanker : les 2 capacités sont strictement supérieures à 0
  - Agent Explorateur : les 2 capacités sont à -1
- Plusieurs parties utilisent des timer donc si on désactive le wait() ou change la durée d'attente qui ralentit les agents pour la visualisation (premières ligne de la classe ExploCoopBehaviour), le gain en efficacité sera limité par certain des timers.

2.2 Diagramme UML



2.3 Hypothèses

Nous faisont des hypothèses sur l'environnement dans notre projet :

— **Horloge système** : Nous supposons que la différence de temps entre les horloges des différents agents/machines sont négligeables. Cette supposition va nous permettre d'utiliser l'horloge du système et de comparer, par exemple, la récence de chaque trésor et de prendre le plus récent, même si ces 2 horodatages ont été calculés sur 2 agents différents.

— **Noms d'agents uniques** : Nous identifions les différents agents par leur noms, en comparant agentName dans l'observation et une liste de noms qu'on obtient au démarrage du code. Nous utilisont également le nom "Silo" pour identifier l'agent central.

3 Stratégie : Exploration puis Exploitation

3.1 Explication du choix

Notre stratégie consiste à diviser le code en 2 parties : L'exploration de la carte, puis l'exploitation des ressources.

— **Phase d'exploration :** Les agents essayent d'acquérir une connaissance suffisante sur l'environnement pour avoir une coordination efficace pendant l'exploitation.

— **Phase d'exploitation :** Nous avons une approche centralisée autour d'un Silo qui attribue les tâches aux autres agents.

3.2 Transition de phase

La transition [explorer -> exploiter] se fait au niveau de l'agent individuellement, tous les agents ne passent donc pas à la phase d'exploitation en même temps. Tous les agents explorent initialement. Un agent passe à la phase d'exploitation soit quand tous les noeuds sont fermés (indiquant qu'il n'y a plus d'autres nouveaux noeuds car la topographie est statique), soit quand il n'y a plus de noeuds ouverts accessibles (par exemple quand un agent bloque un chemin) Les agents autres que le Silo pourront toujours explorer les noeuds ouverts pendant l'exploitation pour découvrir les noeuds manquants.

4 Phase d'Exploration

Voici comment chaque agent explore l'environnement.

4.1 Préparation

Chaque agent commence par créer une représentation de l'environnement (`myMap`, `knownTreasures`). Il construit également des représentations pour les autres agents (`agentMaps`, `lastCommunicationTime`). Il crée aussi des behaviour pour le partage d'information (`ReceiveMapBehaviour`) et pour les demandes de mouvements (`ReceiveMoveRequestsBehaviours`).

4.2 Observation

A chaque mouvement, l'agent observe ses alentours et marque le noeud courant comme fermé, puis ajoute les noeuds voisins à sa carte locale. Il regarde également les trésors en mettant à jour l'information.

4.3 Collaboration

La collaboration est périodique et se passe quand un agent rencontre un autre agent dans son champ d'observation. Pour ce faire il vérifie les 3 aspects suivants :

— il vérifie qu'il n'est pas déjà en train de communiquer une carte à cet agent.
— il vérifie que la carte stockée associée à l'autre agent est moins à jour que la sienne (pour éviter d'envoyer une carte dont on sait que l'autre agent connaît déjà).
— il vérifie qu'il n'a pas déjà envoyé un message à cet agent durant les dernières `timeBetweenMessages` secondes (pour éviter le spam).

Dans ce cas, un behaviour de partage est crée (`ShareMapBehaviour`) en envoyant la carte, les trésors et la position de l'agent.

Quand un agent reçoit de l'information (`ReceiveMapBehaviour`), il merge l'information avec sa connaissance sur la topologie et les trésors, et recalcule un nouveau chemin.

Note : Pour les trésors, ils ont chacun un timestamp créé au niveau d'un agent individuel, et ils mettent à jour leur connaissances sur les trésors en comparant les timestamps.

Il calcule les noeuds ouverts pour les 2 agents en même temps, et renvoie le résultat à l'agent qui à envoyé les informations.

4.4 Mouvement

S'il n'est pas en communication, il suit son chemin calculé vers la destination. Si ce dernier n'est pas valide, Il effectue un BFS sur sa carte en prenant en compte les obstacles visibles. Il bouge ensuite vers son prochain noeud en suivant le chemin.

4.5 Terminaison

S'il n'y a plus de noeuds ouverts dans sa représentation de la carte ou s'il n'y a plus de noeud ouverts accessible, il passe à la phase d'exploitation.

4.6 Complexité

— **Messages :** Le partage de carte entre A agents visibles peut générer $O(A^2)$ messages de taille $O(N + M + T)$ sur une période donnée. La fusion et le calcul de chemins suggérés ajoutent une charge locale.

— **Temps (CPU) par agent :** Principalement dominé par les BFS pour la navigation et le calcul de chemins suggérés lors de la fusion, chacun en $O(N + M)$. La fusion de carte est en $O(N + M)$.

— **Espace (Mémoire) par agent :** Stockage de sa carte locale ($O(N+M)$), des informations sur les trésors ($O(T)$) et des chemins temporaires ($O(N)$).

5 Phase d'Exploitation

Une que les agents terminent leur phase d'exploration, ils passent à la phase d'exploitation.

5.1 L'agent central coordinateur

Un agent Silo se positionne sur un noeud optimal d'après certains critères (cf partie Agent Silo). C'est lui qui va gérer les informations des trésors et des autres agents. Il désigne des missions aux agents les plus adaptés pour collecter les trésors et les ramener au Silo.

5.2 Les agents exploitants

Tous les autres agents suivent des missions données par l'agent Silo. Ces missions peuvent être de l'exploration, l'ouverture ou la collecte de trésor. Les agents peuvent avoir besoin de collaborer entre eux pour ouvrir des trésors. Si ils perdent la position du Silo, ils explorent la carte à la recherche du Silo où d'autres agents qui connaissent sa position.

5.3 Terminaison

Nous calculons l'importance de chaque tâche, qui diminue au fil du temps quand on échoue plusieurs fois une même tâche. L'exploitation est considérée terminée quand il n'y a plus de tâche suffisamment importante.

5.4 Complexité

— **Messages :** La communication avec le Silo (demande/assignation/rapport de mission) est en $O(1)$ par mission. Le nombre total dépend du nombre de missions.

— **Temps (CPU) par agent collecteur :** Navigation en $O(N + M)$ par déplacement. Actions sur trésors supposées rapides.

— **Espace (Mémoire) par agent collecteur :** Similaire à l'exploration, plus les détails de la mission en cours.

6 Agent Silo

Nous avons 1 agent Silo principal. Il effectue l'exploration comme tous les autres agents initialement. Il sera responsable de la coordination des agents. En particulier, il devra :

— Déterminer sa position optimale dans le graphe pour la collecte de trésors et éviter les interblockages.

— Maintenir et centraliser les informations sur l'environnement, les trésors, les agents.

— Maximiser l'efficacité de la collecte.

— Donner des missions aux agents.

Il peut potentiellement exister d'autres agents Silo, qui ne seront pas le Silo principal. Ces Silos ne peuvent pas ramasser de trésors : ils seront donc utilisés pour l'ouverture des trésors ainsi qu l'exploration.

6.1 Positionnement stratégique

Le Silo calcule sa position optimale en donnant un score à chaque noeud :

$$\text{score} = (\text{distance à tous les noeuds du graphe}) \times (\text{multiplicateur basé sur le non-interblocage})$$

L'idée générale est de trouver un noeud le plus proche possible des trésors, qui a beaucoup d'arrêtes (pour faciliter la communication avec les agents), qui évite de bloquer les chemins, et qui évite de séparer le graphe en 2 autant que possible. (L'algorithme en détail (pseudo code) est dans l'annexe 1.)

Cette position est mise à jours toutes les 20 secondes tant qu'il y a des noeuds non découverts sur la carte (pseudo code dans l'annexe 2).

6.2 Déplacement du Silo

Pour éviter que le Silo se déplace et que les agents le perdent, on utilise les 3 principes suivants :

— Le Silo ne se déplace pas immédiatement lorsqu'il trouve une nouvelle position optimale, il attend que tous les agents soient au courant ou que 25 secondes soient passées.

— Tout les agents stockent et se partagent des informations sur la position du Silo ainsi que son status (cf annexe 3). Ce partage est fait de la même manière que le partage de carte mentionné précédemment (`ShareSiloKnowledgeBehaviour` et `ReceiveSiloKnowledgeBehaviour`).

— Si un agent fini tout de même par perdre le Silo, il va explorer la carte jusqu'à trouver le Silo ou un autre agent qui sait où il est.

6.3 Missions

Le Silo central donne des missions à tous les autres agents. Il essaye soit d'attribuer des missions aux agents disponibles autour à chaque seconde, soit lorsqu'il reçoit une demande de mission d'un agent. Pour ce faire il garde une liste de taches qui sont mises à jour tous les 5 secondes. Ces taches correspondent à des missions qu'il peut envoyer aux autres agents, ces taches ont toutes un score d'importance assigné en fonction de leur nature et du status du trésor/noeud ouvert associé.

Lorsque une tache associée à un trésor est assignée à un ou plusieurs agents, toutes les taches associées à ce trésor sont supprimées et un compteur du nombre d'essais pour ce trésor est augmenté (permet de diminuer progressivement l'importance d'un trésor si on rate constamment les missions associées).

Pour assurer que le Silo n'assigne pas que des taches solo, il regarde que un certain pourcentage des meilleurs taches, ainsi les taches solo tels que l'exploration de noeuds ouverts, qui ont tendance à avoir des importances plus basses que les taches collectives ne seront pas prises en comptes, ce qui correspond à demander à l'agent de rester disponible autour car on n'a pas de tache à lui assigner encore.

Le pourcentage des meilleures taches prises en compte augmente tout les secondes jusqu'à redémarrer lorsque on reset les taches. Exemple : Après un reset des taches on ne considère que les 20% meilleures taches, après 1 seconde les 40% meilleures taches, jusqu'à considérer toutes les taches durant la dernière seconde avant le reset.

Cette stratégie permet d'assurer que on puisse toujours demander à des agents de rester disponibles pour ensuite assigner des taches collectives. Mais également de garantir que on finira par assigner une tache solo à des agents après avoir attendu au maximum 4 secondes si on n'a pas réussi à avoir suffisamment d'agents pour une tache collective.

Notes : on a mis le seuil des taches considérées importantes à 1, et la tache d'exploration aléatoire à 1.1, ce qui assure que il y aura toujours une tache solo disponible possible pour tout les agents comme pire tache lorsque on atteint 4 secondes depuis le dernier reset des taches.

6.3.1 Trésors

Pour chaque coffre, l'importance est calculée par la quantité de ressources divisée par le nombre d'essais+1 et multipliée par une constante en fonction du type de tache.

Voici l'importance des tâches dans l'ordre croissant :

1. Collecte d'un trésor avec pertes (x0.75)
2. Ouverture d'un trésor (x1)
3. Collecte d'un trésor (x1.5)
4. Ouverture + collecte d'un trésor (x2.5)

6.3.2 Noeuds ouverts

Pour chaque noeud ouvert, l'importance est calculé comme une constante (20) divisée par le nombre d'essais+1.

Lorsque un agent recoit une tache d'exploration d'un noeud ouvert, si il finit la tache avant la deadline donné, il va continuer à explorer les autres noeuds ouvert jusqu'à ce que la deadline soit dépassée.

6.3.3 Exploration

Il y a également une tâches d'exploration aléatoire qui est assigné comme la moins importante tâche (cf Notes ci-dessus). Lorsque un agent est assigné cette tache il va continuer à éxplorer des noeuds aléatoires jusqu'à ce que la deadline soit atteinte. L'objectif de cette tache est d'éxplorer plus la carte pour trouver par exemple les coffer déplacés par les golems, ainsi que d'éviter que trop d'agents restent proche du Silo et bloquent le chemin.

6.4 Attribution d'une mission

Quand un agent arrive au Silo, ce dernier lui donne une mission parmi les missions les plus importantes. On vérifie s'il peut collaborer avec d'autres agents libres pour les attribuer ces missions. Sinon, on leur demande d'attendre.

6.5 Terminaison

Etant donné que nous filtrons les missions ayant une importance inférieure à un seuil (=1) et que elles sont diminuées à chaque essais, les importances vont diminuer jusqu'à ce qu'il ne reste que la tache d'exploration (importance = 1.1). Lorsque ce moment est atteint l'agent Silo va écrire dans le terminal "Finished : No tasks valuable enough left", on considère que ce message marque la terminaison de l'execution.

7 Protocoles de Communication

Les couples de behaviours suivants fonctionnent de la manière suivante : L'agent veut partager une information, alors il créé une nouvelle behaviour de partage, cette behaviour envoie un message à l'autre agent, puis elle créé une nouvelle behaviour `WaitForAckBehaviour` qui attend un ACK en retour. Lorsque la behaviour de reception (ajoutée lors du setup et reste indéfiniment) recoit le message, elle le traite puis crée une nouvelle behaviour `SendAck`, `SendAck` envoie le ACK avec potentiellement des données en plus puis se supprime (oneshot). Enfin la behaviour `WaitForAckBehaviour` recoit le ACK, le traite, puis passe les informations additionnelles à la behaviour de partage initiale, qui après traitement supprime `WaitForAckBehaviour` et soi même.

- `ShareMapBehaviour` - `ReceiveMapBehaviour`
- `ShareSiloKnowledgeBehaviour` - `ReceiveSiloKnowledgeBehaviour`
- `CommunicateWithSiloBehaviour` - `ReceiveCommunicationWithSiloBehaviour`

Les behaviour en lien avec l'interblockage n'utilisent jamais de ACK ou de réponses, l'agent ne s'arrête même pas. Lorsque un agent voit un autre agent qui le bloque il lui envoie un moveRequest en créant un nouveau `SendMoveRequestBehaviour`, l'autre agent le recoit et le traite via `ReceiveMoveRequestBehaviour`. La raison pour ce choix est que si on perd du temps à s'arréter de bouger ou envoyer et traiter des ACK, l'interbloquage va probablement juste s'empirer alors que il peut souvent être résolu très rapidement.

`ReceiveMissonsBehaviour.java` et `ReceiveCommunicationWithSiloBehaviour` fonctionnent de manière similaire sans ACK.

8 Coordination et Résolution d'interblocage

Pour gérer les interbloquages nous utilisons un système de priorités, tout les agents possèdent une destination principale (destination de la mission) et une destination temporaire (utilisée pour l'interbloquage). Les 2 destinations ont une priorité associée. Lorsque on reçoit une demande de se déplacer hors du chemin on regarde qui à la plus haute priorité, si c'est nous on ignore la requête, si c'est l'autre alors on calcule si on peut aller hors du chemin de l'autre agent (donné en paramètre) puis on change notre destination temporaire et adopte la priorité reçue. Si on voit que on va être bloqué (ex : cul de sac, golem, ou silo) alors on ignore le message et on calcule une nouvelle destination temporaire basée sur l'intersection la plus proche (pour laisser passer l'autre agent), on adopte une priorité de 900+ (les plus hautes priorités, basées sur le type de cul de sac).

Voir l'annexe 4 pour plus de détails (pseudo code de l'algorithme).

Dans le cas où on est stationnaire (attente d'un autre agent pour ouvrir un coffre par exemple) un autre algorithme est utilisé pour éviter de bloquer un chemin, généralement il consiste à attribuer une petite priorité mais gérer certain cas avec plus d'attention pour éviter de se bloquer dans une boucle où les 2 agents se poussent entre eux sur 2 noeuds.

En particulier on cherche les intersections les plus proches vers l'autre agent et vers le reste de la carte puis on choisi où aller en fonction de la priorité de notre destination et des résultats de ces 2 recherches.

Voir l'annexe 5 pour plus de détails (pseudo code de l'algorithme).

8.1 Priorités

Voici la liste des priorités et ce à quoi elles correspondent.

0 pas de but

100 en communication, ne bouge pas pour rester dans le champ

350 aller vers un noeud ouvert pendant l'exploitation

400 aller vers le SILO

600 aller vers un trésor pour confirmer sa location/contenu

625 aller vers un trésor pour le collecter

650 aller vers un trésor pour l'ouvrir

675 aller vers un trésor pour l'ouvrir et le collecter

800 pousse toi je suis le SILO et je vais vers ma nouvelle location

900 pousse toi c'est sans issue (SILO bloque)

925 pousse toi c'est sans issue (golem bloque)

950 pousse toi c'est sans issue (autre)

8.2 Complexité

- **Messages :** 1 message 'MoveRequest' par détection de blocage. En cas de forte congestion, peut être $O(A^2)$ sur une courte période.
- **Temps (CPU) par agent impliqué :**
  - Envoi de 'MoveRequest' : Négligeable.
  - Traitement de 'MoveRequest' et calcul de chemin alternatif : $O(N + M)$ (BFS).
- **Espace (Mémoire) par agent :** Stockage 'lastDeadlockMessage' (petit), chemin temporaire ($O(N)$), structures BFS ($O(N)$).

9 Conclusion

Nous sommes arrivé à court de temps et n'avons pas fini de dé-bugger le code, donc parfois les agents se bloquent à cause d'actions qu'ils ne devraient pas faire où actions qu'ils sont censé faire mais ne font pas. Nous n'avons pas eu le temps de trouver ces bugs, donc parfois l'exécution aboutit à la plupart des agents bloqués autour du Silo. La plupart des éxécutions ont tout de même tendance à finir correctement après au maximum quelques minutes, aux alentours de quelques centaines ou 1000 messages dans notre carte 2018, avec une bonne proportion des messages étant reçus par d'autres agents.

10 Annexes

Annexe 1 : position optimale du silo

```
bestNodeFound = null
bestScoreFound = +infinity

for node in myMap :
    multiplier = 1 // (*1 by default : so no penalty by default)

    // get a central node :
```

```
    distanceToAllNodes = // sum of the best distances to get to every node
                         // ALGO : BFS exploration of the graph and sum together
                         // all the distances found

    // reduce deadlocks (interblockage) :
    // get a node with a high degree (number of adjacent nodes)
    // to have more paths available for the agents
    degree = // number of neighbours of the node
    multiplier *= 1/(degree^3) // example : if there are 3 adjacent nodes,
                           // add a 1/27 multiplier to the final score

    // avoid blocking paths :
    // IDEA : test what the best path is between each adjacent node if the silo
    // occupies (and therefore blocks) the current node.
    // If the distance is highly increased that means we are blocking a
    // *probably* useful path
    if degree > 1 :
        distanceBetweenNeighbours = 0
        for neighbour in neighbours :
            for secondNeighbour in neighbours :
                // in practice we will iterate using indexes instead
                // to avoid testing twice the same couple
                // and the case where neighbour == secondNeighbour
                distance += // Dijkstra from "neighbour" to "secondNeighbour"
                            // with "node" blocked
                if distance == null // if the graph is separated in 2
                    multiplier *= 100 // Very high penalty for making
                                      // the graph discontinuous
                distanceBetweenNeighbours += distance
        distanceBetweenNeighbours /= nombre_couples // use the mean to not
                                                    // penalise nodes with
                                                    // a higher degree
        multiplier *= distanceBetweenNeighbours/10 // we add to the multiplier
                                                   // the resulting distance
                                                   // divided by 10
                                                   // (if we don't divide by 10 it
                                                   // will affect the score
                                                   // too much IMO)
    score = distanceToAllNodes * multiplier
    if score < bestScoreFound: // Corrected from > to < for minimum score
        bestNodeFound = node
        bestScoreFound = score
return bestNodeFound
```

Listing 1 – Algorithme de positionnement du Silo

Annexe 2 : position optimale du silo 2

```
// In the initialisation of SILO :
finalPositionFoundFlag = false

every 20 seconds :
  // if the map is already completely explored
  // we don't need to change position
  if finalPositionFoundFlag == true:
    return

  // run the above algorithm to find the best silo position
  new_position = // SILO BEST POSITION ALGORITHM (See above)

  // if all nodes are closed we can turn the finalPositionFoundFlag
  // to true so we don't have to execute the algorithm anymore
  if myMap.openNodes == null:
    finalPositionFoundFlag = true
```

```
17
18    // test if the new position is already the one we are on
19    if new_position == this.position :
20      return
21
22    // the silo needs to move
23    positionStatus = 3
```

Listing 2 – Mise à jour de la position du Silo

Annexe 3 : status du Silo

`positionStatus` (utilisé pour stocker l'état actuel du silo concernant sa position)

- 0 : pas de position définie (Phase d'Exploration)
- 1 : en transit vers une nouvelle position (ne pas s'arrêter pour communiquer, la destination est prioritaire)
- 2 : actuellement positionné correctement
- 3 : sur le point de changer de position (attend que tous les agents soient informés avant de bouger)

Annexe 4 : Algo interblockage

```
1  - If an agent is blocking our path :
2    - if the last message we sent to that agent is from less than X milliseconds,
         then we wait to not spam their inbox
3    - otherwise we send them a moveRequest message containing :
4      - the priority of our current destination
5      - the path we intend to follow
6      - our current position
7    - either way we can't move so we stay on the same node
8
9  - If we receive a moveRequest message :
10   - We check the priority from that message :
11     - if it's BELOW the priority of our current destination : we ignore it
12     - if it's ABOVE the priority of our current destination :
13             - we need to temporarily change our destination (new
                   temporaryDestination) :
14               - we use a BFS search to find the closest node that is not on
                       the sender's path :
15                 - if we can find such a node :
16                     - the destination is set to that node
17                     - the priority is set to the one we received in the
                           moveRequest message
18                 - if we can't find such a node :
19                     - the priority is set to 900/925/950 depending on the
                           type of dead end
20                     - we use a BFS search in the direction of the sender to
                           find the first intersection,
21                       then we pick a random direction from the intersection
                             and choose a node 1 further for the destination
22                       (edge case : if we can't find an intersection (eg :
                             blocked by a golem in a dead end) the whole thing
23                       is stuck, in which case the expected behaviour needs
                             to be defined (probably a random node))
```

Listing 3 – algo de résolution d'interbloquage

Annexe 5 : Algo interblockage (stationnaire)

```
1  The solution we consider is to give unmoving agents a low priority, but to
       handle smartly certain cases of where to move out of the way
2
3  ALGORITHM for an unmoving agent receiving a moveRequest :
```

```
- if we are not moving because we are communicating (priority 100), then handle the interaction the same way as the algorithm above "4.4) DEADLOCK RESOLUTION"
  ie : consider the temporaryDestination to be the one we are on, and the priority 100
- Otherwise (we are not moving because we are waiting for something (like another agent to open a treasure)), then do the following :
  - we use 2 BFS to find the closest intersection :
    - BFS1 : in the direction of the sender, we choose a node 1 further.
    - BFS2 : in the other directions, we choose a node 1 further
      - if the intersection is further than the sender's destination : write it down for later
  - we compare the priority of our main destination (goal) with the priority of the other agent :
    - if our priority is higher : we create a temporaryDestination with :
      - priority = the priority of our main destination (Goal)
      - destination = the closest intersection between BFS1 and BFS2
      - (if both BFS return NULL, then we are stuck and can't let the other pass.
        Since our priority is higher we should send them a message cantLetYouPass)
    - if his priority is higher : we create a temporaryDestination with :
      - priority = the priority of the sender
      - destination =
        - if the BFS in the other directions found an intersection : we use that one
        - Otherwise (only the BFS in the sender's direction found a valid intersection), then we use that one
        - (if both BFS return NULL, then we are stuck. Since their priority is higher we have no choice but to choose a destination
          further than their destination (if no such node exists (end of a dead way) ideally we should send them a message cantLetYouPass))
```

Listing 4 – algo de résolution d'interbloquage