Projet Fondements des Systèmes Multi-Agents 2025

Master 1 ANDROIDE – Sorbonne Université

Membre du groupe :
MAZLUM Jules

Numéro de groupe : 19

Dépôt Git :
`https://github.com/julesmazlum/Projet-FoSyMa.git`

Table des matières

Date de rendu :
Jeudi 8 mai 2025 – 20h

1. Introduction

Le projet *FoSyMa* s'inscrit dans le cadre de l'UE du même nom du Master 1 ANDROIDE de Sorbonne Université. Il s'appuie sur l'environnement **Dedale** et la plateforme multi-agent **JADE** pour proposer une implémentation distribuée et coopérative du jeu classique *Hunt the Wumpus*. Ce jeu consiste à explorer un environnement inconnu pour y collecter des trésors, tout en évitant ou contrecarrant les actions d'un monstre – le Wumpus, ou Golem dans notre version.

Dans cette variante multi-agent, plusieurs types d'agents (explorateurs, collecteurs, silo) doivent coopérer pour explorer un environnement partiellement observable et dynamique, ramasser un maximum de trésors (or et diamants), et contrer les effets perturbateurs d'un agent adverse mobile, le **Golem**, qui peut déplacer les trésors et refermer les coffres.

Notre implémentation met l'accent sur plusieurs axes :

— une **exploration distribuée** et coordonnée de l'environnement ;
— un **partage optimisé de l'information** (topologie, positions des ressources) ;
— une **collecte collaborative** de ressources nécessitant des compétences spécifiques (serrurerie, force) ;
— une **gestion dynamique de l'environnement**, adaptative face aux actions du Golem.

L'objectif principal du projet est de maximiser la quantité de trésors collectés et stockés dans le silo, tout en respectant les contraintes de communication limitée, de capacité des agents, et de dynamique de l'environnement. Le présent rapport décrit l'architecture logicielle de notre solution, les choix algorithmiques réalisés ainsi que leur analyse critique en termes d'efficacité et de robustesse.

2. Architecture générale

Les diagrammes UML pour les **Behaviours** ne sont pas présentés, car ces classes ne contiennent ni méthodes ni attributs spécifiques. Leur représentation graphique n'apporterait donc pas d'information pertinente ou supplémentaire à la compréhension du système.

2.1 MyExploreAgent

2.1.1 Structure FSM & UML

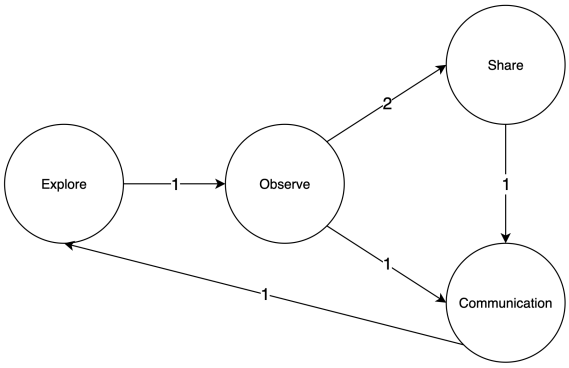

FIGURE 1 – FSM - MyExploreAgent

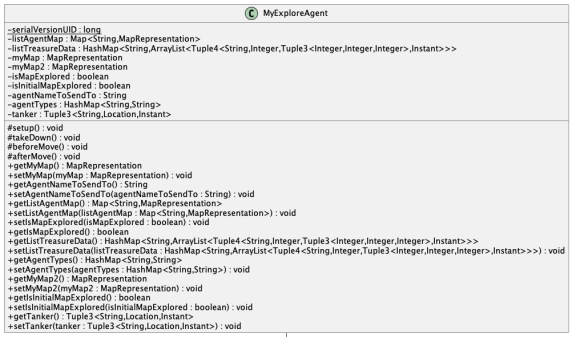

FIGURE 2 – UML - MyExploreAgent

2.1.2 Comportements de l'agent `Explore`

— **ExploreBehaviour** :
Ce comportement permet à l'agent d'explorer l'environnement selon la stratégie initiale. Il s'achève systématiquement en déclenchant le comportement `ObserveBehaviour`.

— **ObserveBehaviour** :
Ce comportement traite les observations locales faites par l'agent. Il parcourt les entités observées et agit comme suit :
— En présence d'un autre agent :
— Si son type est connu et différent de `agentTanker`, on active le comportement `ShareBehaviour`.
— Si son type est `agentTanker`, sa position est enregistrée.
— Si son type est inconnu, l'agent envoie une requête pour le connaître.
— En présence d'un trésor :
— Les informations sont enregistrées : position, quantité, type de coffre (`lockpicking`, `strength`, état), et horodatage.
Le comportement se termine par l'activation du `CommunicationBehaviour`.

— **ShareBehaviour** :
Ce comportement construit et envoie un message à un autre agent contenant :
— la carte de l'environnement,
— la liste des trésors connus,
— les types d'agents rencontrés,
— la dernière position connue du `Tanker`.
Une fois le message envoyé, le comportement déclenche `CommunicationBehaviour`.

— **CommunicationBehaviour** :
Ce comportement permet à l'agent de traiter les messages entrants. Il s'assure de réceptionner tous les messages en attente et d'y répondre si nécessaire. Ce comportement boucle ensuite vers `ExploreBehaviour`. En forçant le passage par `CommunicationBehaviour` à chaque boucle, on évite les cas de famine où l'agent ne consulterait jamais ses messages reçus.

2.2 MyCollectAgent

2.2.1 Structures FSM & UML

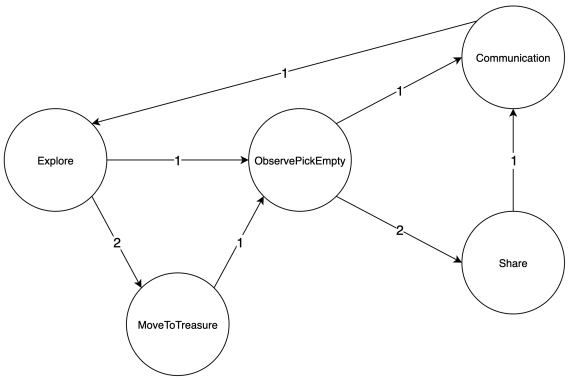

FIGURE 3 – FSM - MyCollectAgent

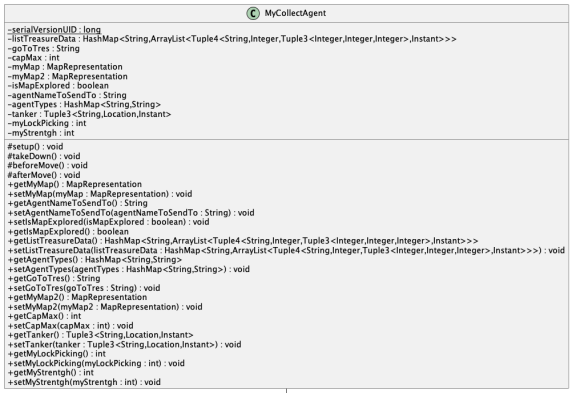

FIGURE 4 – UML - MyCollectAgent

2.2.2 Comportements de l'agent `Collect`

— **ExploreBehaviour** :
Ce comportement décide de l'action à entreprendre en fonction de l'état interne de l'agent :
— Si l'agent connaît des trésors exploitables :
— Si son sac n'est pas plein : transition vers `MoveToTreasureBehaviour`.
— Sinon, si la position du `Tanker` est connue : déplacement vers le `Tanker`, puis transition vers `ObservePickEmptyBehaviour`.
— Sinon : exploration continue.
— Sinon :
— Si son sac n'a pas sa capacité maximale et la position du `Tanker` est connue : déplacement vers le `Tanker`, puis transition vers `ObservePickEmptyBehaviour`.
— Sinon : exploration.
À l'issue de ce raisonnement, le comportement bascule vers `ObservePickEmptyBehaviour`.
— **MoveToTreasureBehaviour** :
L'agent sélectionne le trésor le plus proche parmi ceux qu'il peut récolter. Il calcule un chemin optimal vers celui-ci, puis s'y rend. À l'arrivée, il déclenche le comportement `ObservePickEmptyBehaviour`.
— **ObservePickEmptyBehaviour** :
Ce comportement traite les observations immédiates de l'agent :
— En présence d'un autre agent :
— Si son type est connu et différent de `agentTanker` : transition vers `ShareBehaviour`.
— Si c'est un `Tanker` : enregistrement de sa position et transfert des ressources depuis le sac de l'agent.
— Sinon : requête d'identification du type.
— En présence d'un trésor :
— L'agent tente d'ouvrir le coffre.
— Si succès :
— Si capacité suffisante : il ramasse le trésor.
— Sinon : il ne prend rien.
— Dans tous les cas, il met à jour les informations connues sur ce trésor.
Il termine ensuite par le `CommunicationBehaviour`.
— **ShareBehaviour** :
Ce comportement construit et envoie un message contenant :
— la carte de l'environnement,
— la liste des trésors connus,
— les types des agents rencontrés,
— la dernière position connue du `Tanker`.
Le message est transmis à l'agent rencontré, puis le comportement se termine par `CommunicationBehaviour`.
— **CommunicationBehaviour** :
Ce comportement permet à l'agent de traiter les messages entrants. Il s'assure de réceptionner tous les messages en attente et d'y répondre si nécessaire. Ce comportement boucle ensuite vers `ExploreBehaviour`. En forçant le passage par **CommunicationBehaviour** à chaque boucle, on évite les cas de famine où l'agent ne consulterait jamais ses messages reçus.

Avec cette structure, on peut utiliser le même comportement `ObservePickEmptyBehaviour`, que l'on rencontre un `Tanker` ou d'autres agents de manière aléatoire, ou que l'on s'y soit déplacé intentionnellement.

2.3 MyTankerAgent

2.3.1 Structures FSM & UML

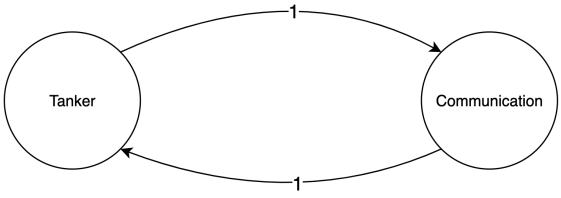

FIGURE 5 – FSM - MyTankerAgent

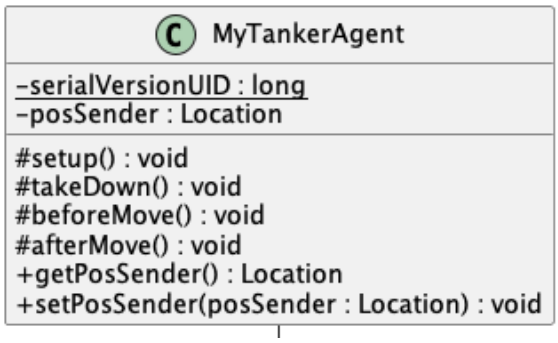

FIGURE 6 – UML - MyTankerAgent

2.3.2 Comportements de l'agent `Tanker`

— **TankerBehaviour** :
L'agent se déplace s'il bloque un autre agent. Le comportement se termine en déclenchant `CommunicationBehaviour`.

— **CommunicationBehaviour** :
Ce comportement permet à l'agent de réceptionner les messages entrants, et répond aux demandes éventuelles. Ce comportement boucle ensuite vers `TankerBehaviour`.

2.4 Global

FIGURE 7 – Classe Global

La classe `Global` contient des méthodes utilitaires communes utilisées par différents agents. Elle centralise certaines opérations récurrentes pour favoriser la réutilisabilité et la cohérence du code.

— `move` :
Cette méthode prend en paramètre une liste de nœuds représentant un chemin, et permet à un agent de le parcourir séquentiellement. Chaque étape du déplacement est gérée jusqu'à atteindre la destination finale.

— `moveNextNode` :
Cette méthode permet à l'agent de se déplacer vers un nœud spécifique donné en paramètre. Elle est utilisée pour des déplacements simples ou unitaires.

— `mergeRessources` :
Cette méthode prend en paramètre deux listes de ressources, et retourne une liste fusionnée. Lors de la fusion, seules les informations les plus récentes sont conservées pour chaque ressource, en se basant sur l'horodatage.

3. Stratégies

3.1 Stratégie de partage incrémental de la carte

Chaque agent `Explore` maintient une structure de données de type :

```
Map<String, MapRepresentation> listAgentMap
```

où :

— la `clé` (`String`) est le nom d'un autre agent rencontré,

— la `valeur` est une représentation de la carte (`MapRepresentation`) associée à cet agent.

Le comportement est le suivant :

1. Lorsqu'un agent rencontre un autre agent pour la première fois :
   — il lui envoie l'intégralité de sa carte locale ;
   — il initialise dans `listAgentMap` une carte vide associée à cet agent.
2. À chaque fois qu'un agent explore un nouveau nœud :
   — le nœud est ajouté à sa propre carte locale ;
   — le nœud est également ajouté à la carte associée à chaque agent dans `listAgentMap`, simulant les informations qu'il pourrait leur transmettre.
3. Lors d'une nouvelle rencontre avec un agent déjà connu :
   — l'agent lui envoie uniquement la carte contenue dans `listAgentMap.get(agentName)`, c'est-à-dire l'ensemble des nœuds qu'il n'a pas encore envoyés à cet agent ;
   — puis il réinitialise la carte de cet agent dans le dictionnaire (nouvelle carte vide).

Forces (avantages)

— **Limite le volume de données échangées** : seuls les nouveaux nœuds sont transmis à chaque rencontre.

Limites (inconvénients)

— Lorsqu'un agent **reçoit une carte** d'un autre agent, il fusionne directement les nouveaux nœuds dans sa carte locale, mais **ne les ajoute pas à son dictionnaire** `listAgentMap`. En conséquence, ces nœuds ne seront pas transmis à d'autres agents, ce qui peut ralentir la diffusion complète de l'information dans le système.

Complexité

— **Temps** : proportionnel au nombre de nœuds nouveaux à transmettre, soit $\mathcal{O}(n)$, avec n le nombre de nœuds découverts depuis la dernière transmission.

— **Mémoire** : chaque agent stocke une carte partielle pour chaque autre agent connu : $\mathcal{O}(k \times m)$, avec k le nombre d'agents et m le nombre de nœuds nouveaux non envoyés.

— **Communication** : très faible en charge : $\mathcal{O}(n)$ par message, avec n les nouveaux nœuds à transmettre. Aucun échange de redondance n'est généré.

Critère d'arrêt Actuellement, aucun critère d'arrêt n'est implémenté. L'agent envoie donc ses données à chaque rencontre, même lorsqu'il n'y a rien de nouveau à transmettre. Cela pourrait être optimisé par un test local : n'envoyer un message que si la carte à transmettre n'est pas vide.

Discussion sur l'optimalité Cette stratégie n'est pas optimale en termes de convergence rapide de la connaissance globale, car elle repose sur une diffusion pair-à-pair lente. Cependant, elle est **simple, robuste et peu coûteuse**, ce qui en fait une solution efficace dans des environnements à faible densité d'agents ou à communications limitées.

3.2 Stratégie de collecte coopérative basée sur les capacités

Tous les agents, qu'ils soient de type `Explore` ou `Collect`, maintiennent une base de connaissances partagée sur les trésors sous la forme d'un dictionnaire :

```
HashMap<String, ArrayList<Tuple4<String, Integer, Tuple3<Integer, Integer,
                    Integer>, Instant»> listTreasureData
```

— La **clé** correspond au type de trésor : `Gold` ou `Diamond`.
— La **valeur** est une liste de `Tuple4`, contenant :

1. la position du trésor (`String`) ;
2. la quantité disponible (`Integer`) ;
3. un `Tuple3` indiquant les compétences requises :
   — en `lockpicking` pour déverrouiller le coffre,
   — en `strength` pour déverrouiller le coffre,
   — et l'état du coffre (ouvert ou fermé).
4. un `Instant` indiquant la date d'enregistrement de l'information.

Ce dictionnaire est échangé à chaque rencontre entre agents afin de synchroniser les données sur l'état des ressources dans l'environnement.

Comportement des agents collecteurs : Chaque agent `Collect` parcourt les trésors connus dans sa base de données. Il sélectionne uniquement ceux qui remplissent les conditions suivantes :
— la quantité disponible est strictement positive ;
— et l'un des deux cas suivants est vérifié :
   — les compétences requises (`lockpicking` et `strength`) sont inférieures ou égales à celles de l'agent ;
   — ou bien le coffre est déjà ouvert.

L'agent sélectionne alors le trésor éligible le plus proche, calcule le chemin vers lui, et s'y rend. Si une fois sur place :
— le trésor a déjà été ramassé par un autre agent ;
— ou bien le coffre est de nouveau verrouillé (et l'agent ne peut pas l'ouvrir) ;
alors il met immédiatement à jour l'information correspondante dans son dictionnaire et passe au prochain trésor éligible de sa liste.

Cette stratégie permet aux agents de s'auto-réguler en fonction de leurs compétences et d'éviter les tentatives de collecte inutiles, tout en maintenant une vision partagée de l'état dynamique de l'environnement.

Forces (avantages)
— **Sélection intelligente des cibles** : les agents ne ciblent que les trésors qu'ils sont capables de collecter.
— **Synchronisation continue** entre agents via les échanges de `listTreasureData`.

Limites (inconvénients)
— **Risque de redondance de trajectoires** : plusieurs agents peuvent viser le même trésor simultanément.

Complexité

- **Temps** : la sélection du trésor le plus proche a une complexité $O(n)$, où n est le nombre de trésors éligibles.
- **Mémoire** : chaque agent stocke tous les trésors connus, donc $O(m)$, avec m le nombre total de trésors.
- **Communication** : à chaque rencontre, les agents échangent leurs bases, soit un coût de $O(m)$ par message (en taille, non en fréquence).

Critère d'arrêt L'algorithme s'arrête localement lorsque :

- il n'existe plus de trésor atteignable par l'agent (soit tous sont déjà pris, soit non accessibles selon ses compétences) ;
- ou que l'agent a atteint sa capacité maximale et ne peut plus collecter sans déposer au `Tanker`.

Discussion sur l'optimalité Cette stratégie n'est pas globalement optimale :

- elle ne garantit pas une répartition optimale des ressources entre les agents ;
- elle ne minimise pas les déplacements au niveau du groupe ;

Cependant, elle offre une **bonne efficacité locale**, est simple à implémenter, fonctionne en environnement distribué, et respecte les contraintes de compétences individuelles. Des mécanismes de coordination explicite (ex. : réservation de trésor) pourraient être ajoutés pour améliorer son optimalité collective.

3.3 Stratégie d'exploration infinie

Une fois que la carte de l'environnement a été totalement explorée (c'est-à-dire que tous les nœuds et connexions ont été découverts), les agents – qu'ils soient de type `Explore` ou `Collect` – doivent néanmoins continuer à se déplacer. Cette stratégie vise à garantir une actualisation continue de la connaissance globale du système multi-agent.

Objectifs :

- Maintenir les informations à jour sur l'état des trésors (quantité, état du coffre).
- Croiser régulièrement d'autres agents afin d'échanger des informations.
- Réagir rapidement à des changements induits par l'environnement dynamique (Golem).

Principe de fonctionnement : Pour permettre une exploration continue même après avoir terminé la cartographie initiale, chaque agent utilise une **carte virtuelle temporaire** :

- Cette carte virtuelle est une copie vide de la topologie connue.
- L'agent réapplique dessus sa stratégie d'exploration initiale, comme s'il découvrait l'environnement à nouveau.
- Une fois que cette carte virtuelle est complètement explorée, elle est réinitialisée, et le cycle recommence.

Pendant ce processus, la carte réelle – représentant la connaissance complète et persistante de l'environnement – est conservée. Elle est utilisée pour :

- calculer des chemins optimaux ;
- mettre à jour les états des trésors ;

Forces (avantages)

- **Maintien d'une activité utile** après l'exploration initiale ;
- **Détection rapide des changements dynamiques** dans l'environnement ;

— **Favorise la communication passive** (rencontres fréquentes avec d'autres agents) ;

Limites (inconvénients)
— **Non ciblé** : les déplacements ne sont pas optimisés pour les zones critiques (ex. : proche d'un trésor connu) ;
— **Aucune gestion de priorité** entre différentes régions ou types de ressources.

Complexité
— **Temps** : équivalent à une exploration classique $O(|V| + |E|)$ par boucle, où $|V|$ est le nombre de nœuds et $|E|$ le nombre d'arêtes ;
— **Mémoire** : une carte virtuelle supplémentaire $O(|V|)$ est stockée par agent ;
— **Communication** : indirecte mais fréquente, car l'exploration provoque de nombreuses rencontres entre agents, entraînant des échanges (carte, trésors, types).

Critère d'arrêt Par nature, cette stratégie est **infinie** : elle ne s'arrête jamais tant que l'agent est actif. Cela garantit une surveillance constante de l'environnement, mais peut être ajusté avec des heuristiques (ex. : stopper temporairement si aucune nouvelle information pendant n cycles).

Discussion sur l'optimalité Cette stratégie est **non optimale** au sens algorithmique : elle ne vise pas à minimiser les déplacements, ni à maximiser une métrique particulière. Cependant, elle est **robuste**, **simple** et très **utile dans un environnement dynamique**, car elle permet une réactivité forte sans surcharger le système de règles complexes. Son efficacité peut être améliorée en combinant avec des priorisations locales ou des observations ciblées.

3.4 Stratégie de gestion des blocages

Comportement de l'agent bloqué : À chaque tentative de déplacement, l'agent vérifie si l'action échoue. Si le déplacement est impossible :

1. l'agent observe son environnement immédiat ;
2. s'il détecte un autre agent sur le nœud cible, il lui envoie un message demandant explicitement de libérer la case ;
3. l'agent répète ce processus (observation, demande) jusqu'à ce que le nœud devienne accessible.

Ce protocole repose sur une coopération implicite : l'agent bloqué compte sur la bonne volonté du bloqueur pour libérer la voie.

Comportement de l'agent bloqueur : Lorsqu'un agent reçoit un message de requête de déplacement :

1. il observe ses nœuds adjacents ;
2. il retire de la liste celui où se situe l'agent demandeur (afin de ne pas s'y déplacer accidentellement) ;
3. deux cas sont alors envisagés :
   — **S'il était en déplacement vers une destination** : il choisit aléatoirement un nœud libre parmi les nœuds observés, s'y rend, puis recalcule un nouveau chemin vers sa destination à partir de cette position ;
   — **S'il explorait simplement** : il se déplace sur l'un des nœuds adjacents disponibles et reprend son exploration normalement.

Forces (avantages)

- **Favorise la fluidité des déplacements** : permet de débloquer dynamiquement des situations de conflit de position.
- **Simple à implémenter** : fonctionne à partir de messages directs sans besoin de négociation complexe.
- **Réactif** : la gestion se déclenche uniquement en cas d'échec de déplacement.

Limites (inconvénients)

- **Suppose la coopération des agents** : si un agent refuse ou ignore la requête, le blocage persiste comme dans le cas d'un Golem.
- **Nécessite que le bloqueur ait un nœud libre** à proximité, ce qui n'est pas toujours garanti.

Complexité

- **Temps** : en moyenne $O(1)$ pour détecter un blocage, mais résolution dépend du contexte (déplacement + recalcul de chemin éventuel).
- **Mémoire** : négligeable (état du voisinage, messages reçus).
- **Communication** : faible, uniquement en cas de blocage. Coût de chaque résolution = un message de requête et une réponse implicite (ou mouvement).

Critère d'arrêt Le protocole s'arrête lorsque l'agent bloqué réussit à accéder à son nœud cible

Discussion sur l'optimalité Cette stratégie est **non optimale globalement**, mais **efficace localement**. Elle vise à résoudre les blocages rapidement sans planification globale. Dans des environnements à faible densité, elle fonctionne très bien. En revanche, dans des situations avec congestion fréquente, des stratégies collectives de gestion des priorités seraient plus adaptées.

3.5 Stratégie de gestion d'impasse pour l'agent `Tanker`

Elle s'applique lorsqu'un agent de type `Tanker` bloque un accès à un nœud cible que souhaite explorer un autre agent, mais se trouve lui-même dans une impasse et est donc incapable de se déplacer pour libérer la voie.

Principe : Lorsqu'un agent `Tanker` reçoit une requête de déplacement mais qu'il détecte qu'il est dans une **impasse** — c'est-à-dire qu'il ne dispose d'aucun nœud adjacent libre — il adopte le comportement suivant :

- il envoie un message à l'agent demandeur, contenant l'identifiant de son nœud actuel ;
- l'agent demandeur intègre alors ce nœud dans sa propre carte locale, en le marquant comme **fermé** ;
- l'agent explorateur peut ainsi ignorer ce nœud dans la suite de sa stratégie d'exploration et passer au nœud suivant.

Conditions de validité : Cette stratégie est valide uniquement si le `Tanker` est dans une impasse structurelle, c'est-à-dire qu'aucun de ses nœuds adjacents n'est libre. Si, en revanche, il est temporairement bloqué par un autre agent, il ne considère pas cela comme une impasse et peut tenter de résoudre la situation en envoyant une requête de déplacement. Cette distinction évite que des nœuds encore explorables soient marqués à tort comme fermés.

Forces (avantages)

— **Permet à l'agent explorateur de ne pas rester bloqué inutilement** sur un nœud inaccessible.

Complexité

— **Temps** : très faible. Envoi d'un seul message et mise à jour locale d'un nœud.
— **Mémoire** : négligeable (mise à jour d'un nœud comme "fermé" dans la carte locale).
— **Communication** : un message unidirectionnel (le Tanker informe l'agent bloqueur de sa position).

Critère d'arrêt L'échange s'arrête dès que l'agent Tanker a informé l'agent explorateur de son incapacité à bouger, et que ce dernier a retiré le nœud de sa stratégie d'exploration.

Discussion sur l'optimalité La stratégie est **efficace pour éviter les blocages structurels** dans des zones sans issue, en permettant à l'agent explorateur d'ignorer des nœuds inaccessibles de manière fiable.

4. Conclusion

Synthèse générale Ce projet nous a permis de concevoir un système multi-agent coopératif dans un environnement partiellement observable, dynamique et contraint. À travers l'exploration, la communication distribuée et la collecte de ressources, nous avons pu expérimenter différentes stratégies à la fois robustes et adaptatives. Les agents sont capables d'explorer efficacement la carte, de partager leurs connaissances de manière incrémentale, et de se coordonner pour éviter les blocages structurels.

Regard critique Plusieurs limitations subsistent dans notre implémentation actuelle.
Tout d'abord, la **collecte collective**, où plusieurs agents unissent leurs compétences pour ouvrir un coffre trop complexe pour un seul, n'a pas pu être mise en œuvre faute de temps. Une stratégie envisagée consistait à ce que, une fois les trésors individuels récoltés, les agents se coordonnent pour traiter les coffres restants un à un, en mutualisant leurs compétences.

De plus, nous n'avons pas exploité l'odeur du Golem dans notre projet. Cette fonctionnalité aurait pu permettre une détection indirecte de sa position, ce qui serait utile pour anticiper ses déplacements ou retrouver des trésors déplacés.

Sur le plan du code, nous avons fait le choix de séparer les classes et comportements de chaque type d'agent (Tanker, Explore, Collect) pour des raisons de clarté et de lisibilité. Cependant, cette décision a conduit à une certaine redondance de code, notamment entre les agents `Explore` et `Collect` qui partagent plusieurs comportements similaires (exploration, observation, partage de données, etc.). Une factorisation de ces comportements communs serait une amélioration structurelle pertinente.

Pistes d'amélioration Pour améliorer notre système, plusieurs extensions sont envisageables :

— Implémenter la coordination pour la collecte collective ;
— Utiliser l'odeur du Golem comme capteur indirect pour la planification adaptative ;
— Réduire la duplication de code en mutualisant les comportements communs dans une classe abstraite ou via l'héritage ;

Ce projet a été l'occasion d'explorer concrètement les fondements des systèmes multi-agents et de faire émerger des comportements collectifs cohérents à partir de règles simples et locales.