Rapport Agent BDI

Développement d'un environnement multi-agent avec JADE

Auteurs :

Amir Benziane

Joubrane Belabbas Bengraa

Université Claude Bernard Lyon 1

UFR Informatique – UE : Intelligent Agents

Encadré par : Dr. Nadia KABACHI

Date de rendu : 3 février 2026

Année universitaire 2024–2025

Master Informatique – Spécialité Intelligence Artificielle

Table des matières

1 Rappel sur le modèle BDI

Le modèle **BDI (Beliefs–Desires–Intentions)** est une architecture classique utilisée pour modéliser des agents rationnels. Elle repose sur trois composantes principales :

- **Beliefs (Croyances)** : représentent la connaissance de l'agent sur le monde, souvent incomplète ou incertaine.
- **Desires (Désirs)** : représentent les états que l'agent souhaite atteindre.
- **Intentions** : représentent les désirs choisis sur lesquels l'agent s'engage et qu'il planifie d'exécuter.

Le schéma suivant illustre le fonctionnement général d'un agent BDI :

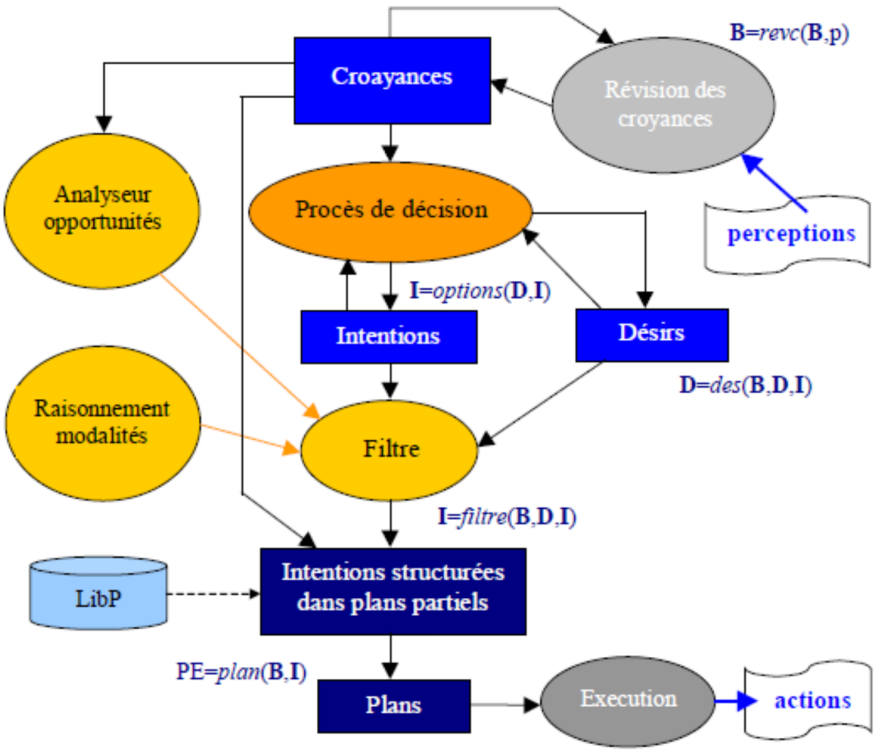

FIGURE 1 – Architecture du modèle BDI (Beliefs–Desires–Intentions).

Ce schéma montre comment les perceptions de l'environnement modifient les croyances de l'agent, comment les désirs sont générés, puis filtrés en intentions, avant d'être traduits en plans concrets et exécutés.

Le modèle BDI repose sur une représentation cognitive de l'agent : celui-ci raisonne en fonction de ce qu'il croit du monde (ses croyances), de ce qu'il souhaite atteindre (ses désirs), et de ce qu'il décide réellement de faire (ses intentions). Il définit un cycle de décision où les perceptions modifient les croyances, lesquelles influencent la génération des désirs, puis le choix des intentions et enfin l'exécution des actions.

2 Comment traduire un modèle BDI dans le code ?

Dans notre projet, cette architecture a été traduite concrètement dans les classes d'agent crée, ici nous allons citer l'exemple de `RobotAgent`, écrite en Java avec le framework JADE. Chaque composante du modèle théorique correspond à une partie bien identifiée du code :

2.1 Croyances (Beliefs)

Elles sont représentées par la structure `Map<String, Object> beliefs`, qui stocke toutes les informations internes de l'agent — sa position, son niveau d'énergie, la carte de l'environnement, la position du vaisseau ou encore les pierres détectées. Leur mise à jour s'effectue à travers la méthode `updateBeliefs()`, laquelle tient compte à la fois des perceptions (messages reçus d'autres agents) et des événements du monde (position actuelle, énergie, stock, etc.).

2.2 Désirs (Desires)

Les désirs sont définis dans la méthode `generateDesires()`, et expriment les objectifs possibles de l'agent selon son état actuel. Par exemple : *remplir le stock de pierres*, *retourner au vaisseau* ou *attendre un appel*. Chaque désir est associé à une priorité, permettant à l'agent de hiérarchiser ses motivations selon la situation.

2.3 Intentions (Intentions)

Les intentions sont sélectionnées par la méthode `filterIntentions()`, qui choisit les désirs les plus prioritaires et les transforme en intentions concrètes. Elles sont ensuite traduites en plans d'action séquentiels (par exemple : aller vers une pierre ou revenir au vaisseau).

2.4 Plans (Plans)

Les plans sont implémentés sous forme de comportements `(OneShotBehaviour)` tels que `QuitVaisseauPlan`, `CourtCheminPlan`, `Avancer1CasePlan` ou `RecupPierrePlan`. Ces plans traduisent les intentions en actions exécutables sur l'environnement : quitter le vaisseau, suivre un chemin, avancer d'une case ou ramasser une pierre. Leur exécution modifie à nouveau les croyances, bouclant ainsi le cycle BDI.

2.5 Cycle global du RobotAgent

L'ensemble du processus est orchestré par un cycle récurrent défini dans le `TickerBehaviour`, qui exécute successivement :

1. la mise à jour des croyances (`updateBeliefs()`),
2. la génération des désirs (`generateDesires()`),
3. le filtrage des intentions (`filterIntentions()`),
4. l'exécution des plans (`executePlans()`),
5. la vérification de la satisfaction des désirs (`checkSatisfaction()`).

Ce cycle correspond directement au flux représenté dans le schéma BDI, reliant la perception, la révision des croyances, la génération des désirs, le filtrage des intentions et l'exécution des actions. Ainsi, l'agent `RobotAgent` illustre une traduction concrète du modèle BDI : il raisonne en fonction de son environnement, ajuste ses décisions en continu et agit de manière autonome selon ses priorités et ses contraintes énergétiques.

3 Ressources partagées entre agents

Dans le système multi-agent développé, certaines informations sont communes à l'ensemble des agents et ne sont donc pas stockées dans leurs croyances individuelles. C'est notamment le cas des classes `Carte` et `CarteExploree`, qui constituent des ressources globales accessibles à tous.

3.1 La classe `Carte`

La classe `Carte` représente la structure fixe de l'environnement dans lequel évoluent les agents. Elle contient les informations sur la nature de chaque case (sol libre, obstacle, pierre, vaisseau, etc.) et permet aux agents de connaître l'état du terrain à tout moment. Grâce à cette carte, les agents peuvent :

— déterminer le type de case sur laquelle ils se trouvent ou qu'ils souhaitent atteindre ;
— calculer le plus court chemin entre deux positions (par exemple entre une pierre et le vaisseau) ;
— anticiper leurs déplacements en fonction des obstacles ou des zones déjà explorées.

La carte agit donc comme une représentation globale de l'environnement physique partagé par l'ensemble des agents.

3.2 La classe `CarteExploree`

L'objet `CarteExploree` est implémenté sous la forme d'un *singleton*. Cela signifie qu'il n'existe qu'une seule et unique instance de cette classe dans tout le système. Chaque agent y accède via la méthode `CarteExploree.getInstance()`, garantissant qu'ils partagent tous la même carte explorée en mémoire.

Cette conception permet :

— de centraliser les informations sur les zones déjà explorées ;
— d'éviter les duplications de données entre agents ;

— de favoriser la coopération et la coordination dans la prise de décision ;
— d'assurer la cohérence globale des connaissances sur l'environnement.

Ainsi, les classes `Carte` et `CarteExploree` forment les deux piliers des ressources partagées du système. La première décrit la configuration statique du monde, tandis que la seconde en constitue la **mémoire collective dynamique**, mise à jour en temps réel par les agents au fil de leurs explorations.

4 Nos agents

4.1 Le ResearcherAgent

Le **ResearcherAgent** (ou agent explorateur) a pour rôle de parcourir la carte afin de découvrir de nouvelles zones et de détecter les pierres à collecter. Il signale ensuite la position de ces pierres au `RobotAgent` pour qu'elles soient récupérées. Son comportement repose également sur le modèle BDI (*Beliefs – Desires – Intentions*), qui lui permet d'explorer de manière autonome et intelligente son environnement.

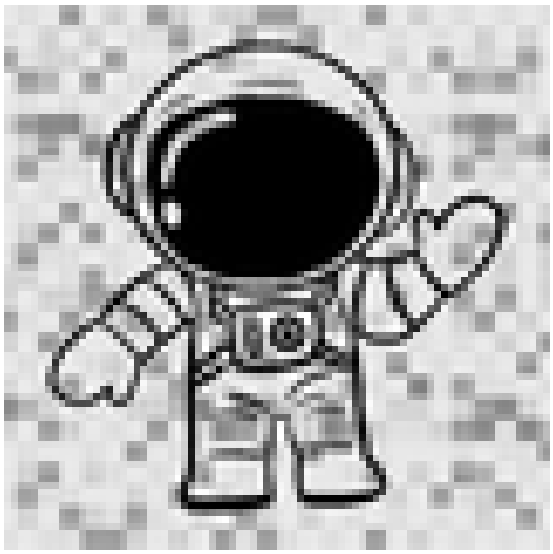

FIGURE 2 – Représentation du ResearcherAgent dans l'interface graphique.

Croyances (Beliefs)

Les croyances du `ResearcherAgent` représentent sa connaissance actuelle de la carte et de son état interne. Elles sont stockées dans une structure `Map<String, Object> beliefs` et mises à jour en continu au fil des explorations.

— **position** : position actuelle de l'agent explorateur.
— **positionVaisseau** : position fixe du vaisseau (base).
— **energie** : niveau d'énergie de l'agent (maximum : 1000).
— **pierresTrouvees** : liste des positions des pierres détectées.

— **cheminExploration / cheminVaisseau** : chemins calculés pour explorer ou revenir à la base.

L'agent maintient également une référence à la carte globale ainsi qu'à une instance partagée de `CarteExploree`, utilisée par l'ensemble des agents pour mémoriser les zones déjà explorées.

Révision des croyances

La méthode `updateBeliefs()` assure la mise à jour des informations de l'agent :
— recharge automatique de l'énergie lorsqu'il retourne au vaisseau ;
— réception de messages `ACL` confirmant des recharges effectuées ;
— actualisation continue de sa position dans la carte partagée.

Désirs (Desires)

Les désirs du `ResearcherAgent` sont générés dynamiquement dans la méthode `generateDesires()`. Ils expriment les objectifs possibles selon l'état actuel de l'agent (niveau d'énergie, zones explorées, etc.).

— **explorerCarte (priorité 80)** : explorer activement les zones inconnues de la carte.
— **retournerVaisseau (priorité 90–100)** : retourner au vaisseau lorsque la carte est entièrement explorée ou lorsque l'énergie devient faible.

Les désirs sont hiérarchisés selon leur priorité, afin que l'agent choisisse toujours la tâche la plus importante à accomplir.

Intentions et plans associés

Le désir prioritaire est transformé en intentions concrètes via la méthode `filterIntentions()`. Ces intentions définissent la séquence d'actions à réaliser pour atteindre le but choisi. Elles sont ensuite exécutées à travers des `OneShotBehaviour` spécifiques.

1. Désir : explorer la carte
— **Intentions générées** :
  — `choisirZoneInexploree` – choisir une zone frontalière à explorer.
  — `avancer1case` – se déplacer vers la zone sélectionnée.
  — `scannerPierre` – détecter et signaler les pierres présentes sur la case actuelle.
— **Plans associés** : `ChoisirZonePlan`, `Avancer1CasePlan`, `ScannerPierrePlan`.

2. Désir : retourner au vaisseau
— **Intentions générées** :
  — `atteindre(vaisseau)` – calculer le chemin vers le vaisseau.
  — `avancer1case` – avancer d'une case sur le chemin.

— `scannerPierre` – continuer à signaler les pierres rencontrées sur le trajet.

— **Plans associés** : `CourtCheminVaisseauPlan`, `Avancer1CasePlan`, `ScannerPierrePlan`.

Plans

Chaque plan est implémenté sous forme d'un comportement (`OneShotBehaviour`) exécuté par l'agent :

— `ChoisirZonePlan` : sélectionne une zone inexplorée adjacente à une zone connue et calcule le chemin pour s'y rendre.

— `CourtCheminVaisseauPlan` : calcule un chemin vers la base pour permettre le retour et la recharge.

— `Avancer1CasePlan` : déplace l'agent d'une case sur le chemin prévu et réduit son énergie.

— `ScannerPierrePlan` : détecte les pierres présentes sur la case actuelle et envoie un message au `RobotAgent` pour les signaler.

Communication inter-agents

Lorsqu'une pierre est détectée, l'agent envoie un message :

— type : `ACLMessage.INFORM`

— destinataire : `Robot1` (le `RobotAgent`)

— contenu : `"Pierre détectée à x=.. y=.."`

Il marque ensuite la position détectée comme explorée dans la carte partagée pour éviter toute redondance.

En cas de panne énergétique, l'agent envoie un message `SOS` au `RescueAgent` afin d'obtenir une assistance :

— type : `ACLMessage.REQUEST`

— destinataire : `Robot2` (le `RescueAgent`)

— contenu : `"SOS besoin d'énergie à x=.. y=.."`

Satisfaction et adaptation

La méthode `checkSatisfaction()` vérifie la complétion des désirs :

— le désir `retournerVaisseau` est satisfait lorsque l'agent atteint la position du vaisseau ;

— le désir `explorerCarte` reste actif tant qu'il existe des zones inexplorées.

Synthèse

Le `ResearcherAgent` est un agent proactif et autonome, capable d'explorer la carte, de signaler les ressources et de coopérer avec les autres agents. Grâce au modèle BDI, il alterne entre exploration, communication et retour à la base selon l'état de son environnement. Il joue un rôle essentiel dans la coordination du système : il guide le `RobotAgent` vers les

ressources tout en s'appuyant sur le `RescueAgent` pour assurer sa survie en cas de panne énergétique.

4.2 Le RobotAgent

Le **RobotAgent** est l'agent principal du système. Il est responsable de la collecte des pierres sur la carte, de leur dépôt au vaisseau, et de la gestion de son énergie. Il agit de manière autonome selon le modèle BDI implémenté dans son code Java.

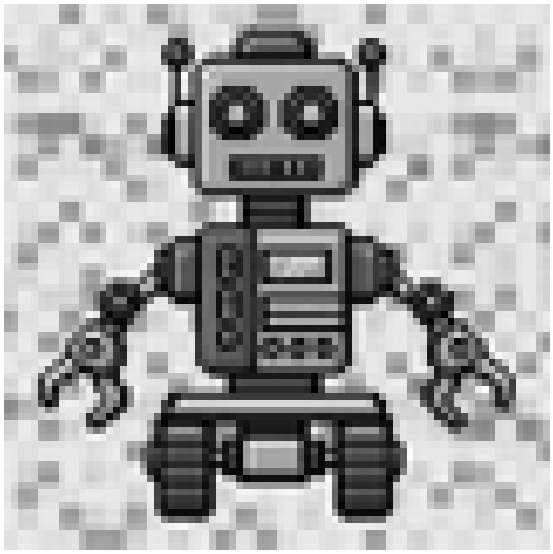

FIGURE 3 – Représentation du RobotAgent dans l'interface graphique.

Croyances (Beliefs)

Les croyances constituent la base de connaissance du robot. Elles regroupent toutes les informations sur son environnement et son état interne. Dans le code, elles sont stockées dans une structure `Map<String, Object> beliefs`.

- **position** : position actuelle du robot sur la carte.
- **positionVaisseau** : position fixe du vaisseau.
- **energie** : niveau d'énergie restant (valeur maximale : 300).
- **pierre** : nombre de pierres actuellement transportées.
- **pierresEnAttente** : file des pierres détectées mais pas encore ramassées.
- **ciblePierre** : position de la pierre actuellement visée.
- **cheminPierre** / **cheminVaisseau** : chemins calculés vers la pierre ou vers le vaisseau.
- **retourAuto** / **dontcheckreturn** : indicateurs de décision liés au comportement de retour.

Désirs (Desires)

Les désirs du robot sont générés dynamiquement dans la méthode `generateDesires()`. Ils représentent les buts possibles en fonction de la situation actuelle (niveau d'énergie, stock, etc.). Chaque désir est associé à une priorité (valeur entière), permettant à l'agent de choisir celui qui est le plus urgent.

- **remplirStock (priorité 70)** : collecter des pierres tant que le stock n'est pas plein.
- **retournerVaisseau (priorité 100)** : retourner au vaisseau pour recharger ou déposer les pierres.
- **attendreAppel (priorité 10)** : attendre la détection d'une nouvelle pierre si aucune cible n'est disponible.

Intentions et plans associés

Chaque désir sélectionné est traduit en intentions concrètes dans la méthode `filterIntentions()`. Les intentions sont ensuite exécutées à travers des plans (`OneShotBehaviour`) décrits ci-dessous.

1. Désir : remplir le stock

- **Intentions générées** :
  - `quitVaisseau` – quitter la base.
  - `atteindre(pierre)` – calculer le chemin vers la pierre cible.
  - `avancer1case` – avancer d'une case selon le chemin.
  - `recupPierre` – ramasser la pierre à la position actuelle.
- **Plans associés** : `QuitVaisseauPlan`, `CourtCheminPlan`, `Avancer1CasePlan`, `RecupPierrePlan`.

2. Désir : retourner au vaisseau

- **Intentions générées** :
  - `atteindre(vaisseau)` – calculer le chemin vers le vaisseau.
  - `avancer1case` – suivre le chemin jusqu'au vaisseau.
- **Plans associés** : `CourtCheminPlan`, `Avancer1CasePlan`.

3. Désir : attendre un appel

- **Intentions générées** : aucune (état passif).
- **Comportement** : l'agent reste en veille jusqu'à la réception d'un message contenant la position d'une nouvelle pierre détectée.

Communication inter-agents

Le `RobotAgent` communique principalement avec deux autres agents du système :

- Le `ResearcherAgent`, qui lui envoie la position des pierres détectées sur la carte :

- **type** : `ACLMessage.INFORM`
- **contenu** : `"Pierre détectée à x=.. y=.."`

— Le `RescueAgent`, qu'il peut contacter en cas de panne énergétique :

- **type** : `ACLMessage.REQUEST`
- **contenu** : `"SOS besoin d'énergie à x=.. y=.."`

Le robot met à jour ses croyances dès réception de ces messages. Lorsqu'il reçoit la position d'une nouvelle pierre, celle-ci est ajoutée à sa file de `pierresEnAttente`. En cas de message de recharge, il met son énergie à jour et relance le cycle de collecte.

Satisfaction et adaptation

La méthode `checkSatisfaction()` vérifie la complétion des désirs :

— le désir `remplirStock` est satisfait lorsque le stock atteint la capacité maximale ;

— le désir `retournerVaisseau` est satisfait lorsque l'agent se trouve à la position du vaisseau ;

— le désir `attendreAppel` reste actif tant qu'aucune pierre nouvelle n'a été signalée.

Une fois un désir satisfait, le `RobotAgent` réévalue ses croyances, génère de nouveaux désirs, et poursuit son cycle de décision. Cette adaptation continue lui permet de s'ajuster en temps réel à la dynamique de l'environnement.

Synthèse

Le `RobotAgent` illustre ainsi un agent autonome capable de raisonner sur son environnement : il met à jour ses croyances à chaque cycle, génère des désirs cohérents, choisit l'intention la plus pertinente et exécute le plan correspondant. Ce comportement lui permet de s'adapter dynamiquement à l'évolution du monde et d'agir de manière rationnelle.

4.3 Le RescueAgent

Le **RescueAgent** est l'agent de secours du système multi-agent. Sa mission consiste à assister les robots en détresse lorsque leur niveau d'énergie atteint zéro. Lorsqu'un robot envoie un signal de détresse (`SOS`), le `RescueAgent` quitte le vaisseau, se rend à la position du robot, le recharge, puis retourne automatiquement à la base. Comme les autres agents, il adopte le modèle BDI (*Beliefs – Desires – Intentions*) pour raisonner et agir de manière autonome.

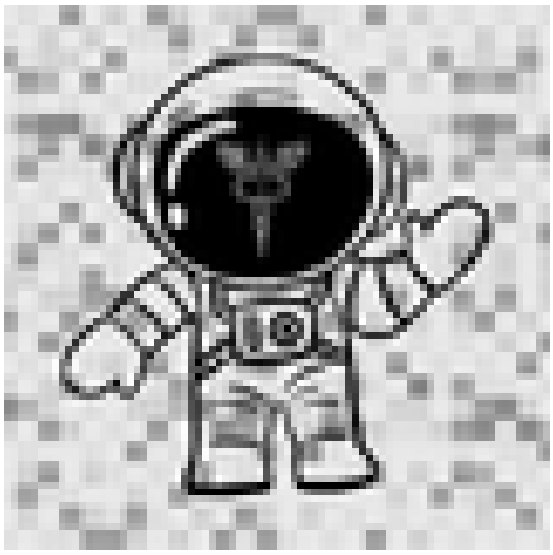

FIGURE 4 – Représentation du RescueAgent dans l'interface graphique.

Croyances (Beliefs)

Les croyances représentent les connaissances internes de l'agent sur son environnement et son état. Elles sont stockées dans une structure `Map<String, Object> beliefs` et mises à jour à chaque cycle de décision.

- **position** : position actuelle de l'agent de secours.
- **positionVaisseau** : position fixe du vaisseau.
- **energie** : niveau d'énergie actuel du `RescueAgent` (valeur maximale : 3000).
- **robotPosition** : position du robot en détresse à secourir.
- **appelSOS** : indicateur de réception d'un signal de détresse.
- **missionTerminee** : indique si la mission de secours est terminée.
- **cheminRobot / cheminVaisseau** : chemins calculés pour rejoindre le robot ou retourner à la base.

Révision des croyances

La méthode `updateBeliefs()` met à jour les croyances à chaque cycle :

- recharge de l'énergie lorsque l'agent revient au vaisseau ;
- réception et interprétation des messages `SOS` envoyés par le `RobotAgent` en panne ;
- mise à jour de la position du robot en détresse et initialisation de la mission de secours.

Désirs (Desires)

Les désirs du `RescueAgent` sont générés dynamiquement dans la méthode `generateDesires()`. Ils traduisent les buts possibles de l'agent selon la situation courante (présence d'un appel, état de la mission, position actuelle).

- **aiderRobot (priorité 100)** : se rendre auprès du robot en détresse et le recharger.

— **retournerVaisseau (priorité 50)** : retourner à la base après une mission terminée.
— **attendreAppel (priorité 10)** : rester en veille au vaisseau tant qu'aucun signal `SOS` n'est reçu.

Les désirs sont triés par priorité, afin que l'agent choisisse toujours le plus urgent à poursuivre.

Intentions et plans associés

Le désir sélectionné est transformé en intentions concrètes dans la méthode `filterIntentions()`. Ces intentions sont ensuite traduites en plans, implémentés sous forme de `OneShotBehaviour` exécutés séquentiellement.

1. Désir : aider un robot en détresse
— **Intentions générées** :
  — `quitVaisseau` – quitter la base.
  — `atteindre(robot)` – calculer le chemin jusqu'au robot.
  — `avancer1case` – suivre le chemin calculé.
  — `rechargerRobot` – recharger le robot une fois à proximité.
— **Plans associés** : `QuitVaisseauPlan`, `CourtCheminPlan`, `Avancer1CasePlan`, `RechargerRobotPl`

2. Désir : retourner au vaisseau
— **Intentions générées** :
  — `atteindre(vaisseau)` – calculer un chemin vers la base.
  — `avancer1case` – avancer sur le chemin de retour.
— **Plans associés** : `CourtCheminPlan`, `Avancer1CasePlan`.

3. Désir : attendre un appel
— **Intentions générées** : aucune (état passif).
— **Comportement** : l'agent reste en veille au vaisseau et surveille les messages entrants jusqu'à la réception d'un `SOS`.

Plans

Les plans sont implémentés sous forme de classes internes, chacune correspondant à une action spécifique :
— `QuitVaisseauPlan` : quitte le vaisseau et avance d'une case sur la carte.
— `CourtCheminPlan` : calcule le chemin vers le robot ou vers la base.
— `Avancer1CasePlan` : avance d'une case selon le chemin calculé.
— `RechargerRobotPlan` : recharge le robot en détresse lorsque le `RescueAgent` est adjacent à lui, puis signale la fin de la mission.

Communication inter-agents

Le `RescueAgent` communique principalement avec le `RobotAgent`. Deux types d'échanges sont possibles :

- **Réception d'un message SOS** :
  - **type** : `ACLMessage.REQUEST`
  - **expéditeur** : `Robot1` (le `RobotAgent`)
  - **contenu** : `"SOS besoin d'énergie à x=.. y=.."`
- **Envoi d'un message de confirmation de recharge** :
  - **type** : `ACLMessage.INFORM`
  - **destinataire** : `Robot1`
  - **contenu** : `"Recharge effectuée !"`

Grâce à cette communication, le `RescueAgent` assure une coordination fluide avec les autres agents : il reçoit des appels à l'aide, planifie le déplacement jusqu'au robot concerné, puis confirme la fin de la mission.

Satisfaction et adaptation

La méthode `checkSatisfaction()` vérifie si le désir actif est accompli :

- le désir `aiderRobot` est satisfait lorsque la mission de secours est marquée comme terminée ;
- le `RescueAgent` retourne alors automatiquement au vaisseau pour se remettre en veille.

Synthèse

Le `RescueAgent` illustre un agent BDI coopératif et réactif. Il perçoit les appels à l'aide, raisonne sur la meilleure action à entreprendre, planifie son déplacement, puis agit pour recharger le robot concerné. Une fois la mission accomplie, il met à jour ses croyances et retourne à la base. Ce comportement renforce la robustesse et la coordination du système multi-agent : les robots peuvent compter sur une assistance automatique et efficace en cas de panne d'énergie.

5 Illustrations de l'environnement multi-agent

Nous présentons ci-dessous trois captures d'écran issues de notre environnement JADE, illustrant les principales étapes du fonctionnement du système multi-agent.

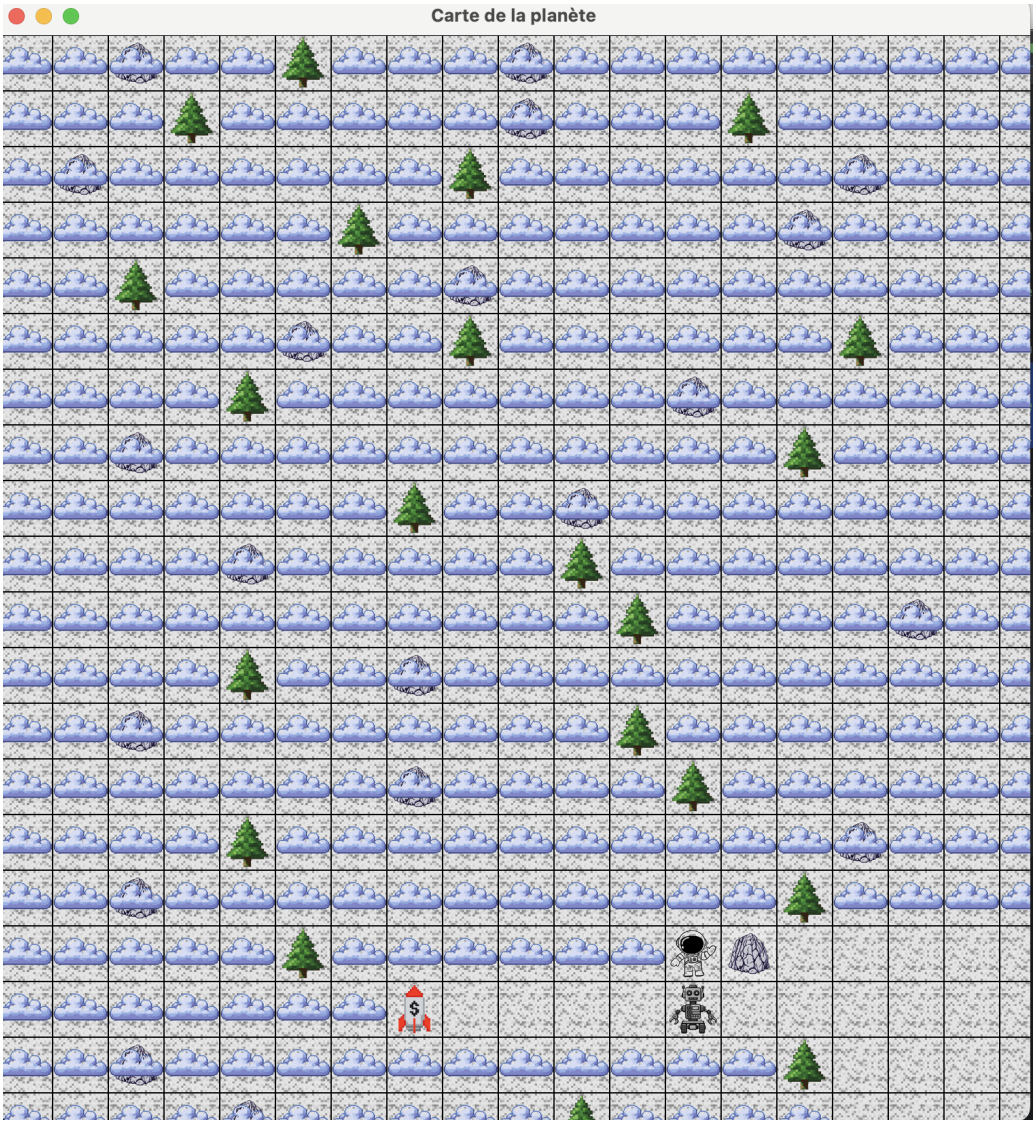

FIGURE 5 – Étape 1 — Début de l'exploration : le `ResearcherAgent` parcourt la carte, détecte une pierre, et envoie un message au `RobotAgent` pour qu'il intervienne.

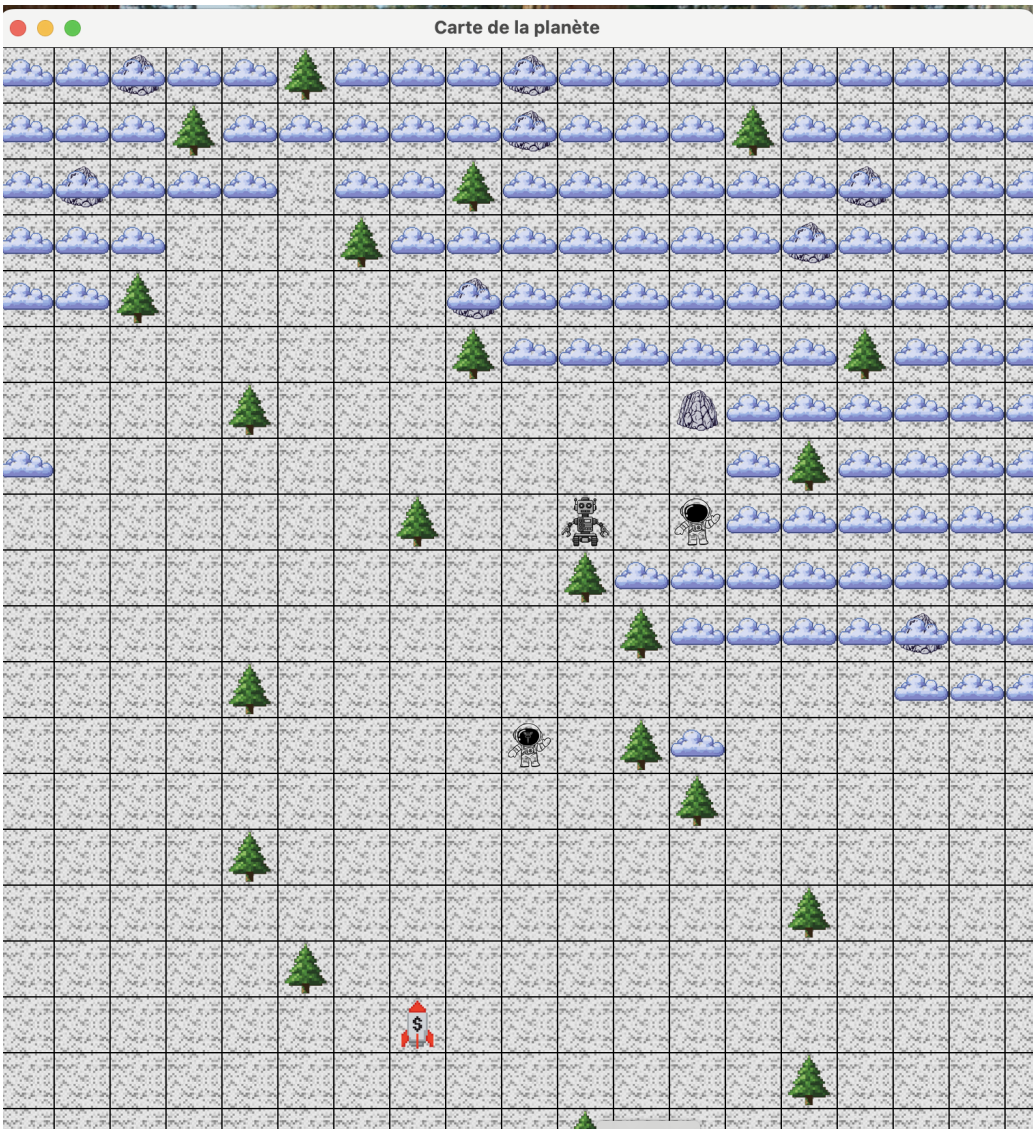

FIGURE 6 – Étape 2 — En pleine exploration : le `RobotAgent` tombe en panne d'énergie pendant la collecte, et fait appel au `RescueAgent` pour obtenir une recharge.

FIGURE 7 – Étape 3 — Fin de mission : toutes les zones ont été explorées, les pierres collectées, et les agents regagnent le vaisseau.

Ces images illustrent la coopération dynamique entre les agents du système : le `ResearcherAgent` explore et signale les ressources, le `RobotAgent` collecte et transporte les pierres, tandis que le `RescueAgent` garantit la continuité énergétique du groupe.

Conclusion

À travers ce projet, nous avons conçu et expérimenté une véritable architecture BDI au sein d'un environnement multi-agent. Chaque agent agit de manière autonome tout en collaborant avec les autres, illustrant la puissance du raisonnement basé sur les croyances, désirs et intentions.

Le `ResearcherAgent`, le `RobotAgent` et le `RescueAgent` forment un écosystème cohérent où la communication et la coordination permettent d'accomplir des tâches complexes sans supervision centrale.

Notre travail démontre que la plateforme **JADE** constitue une solution robuste, flexible et efficace pour le développement d'agents intelligents. Elle offre une structure claire pour la modélisation des comportements, la gestion des communications et la simulation d'environnements coopératifs à grande échelle.