2024 - 2025

CS534- Intelligence artificielle distribuée et systèmes multi-agents

Rapport de travail

Hands-on lab with MaDKit

Rédigés par :

Zié Ismaël KONE

Elsy Reine NGOULA KENFACK

Année Académique
2024-2025

4 Application exercise - Role-playing games

- L'agent **EmitterAgent** est un acteur clé dans la simulation dynamique. Il génère des messages de manière aléatoire, ce qui stimule les agents **CounterAgent** et les oblige à effectuer leur rôle de comptage. Sa terminaison automatisée assure un renouvellement naturel des agents dans le système.

Activation (activate) :

L'agent rejoint la société artificielle en créant ou rejoignant un groupe nommé "communication" et en prenant le rôle "RolePlay".

Il s'enregistre également sous le rôle "EmitterAgent" pour indiquer sa nature spécifique.

L'utilisation de pause (500) permet de donner un délai pour assurer la création du groupe avant de continuer.

Phase de vie (live) :

L'agent détermine de manière aléatoire le nombre de messages qu'il va envoyer (valeur comprise entre 0 et 9).

Il recherche un agent ayant le rôle "CounterAgent" dans le groupe "RolePlay". Cette recherche boucle jusqu'à ce qu'un agent cible soit trouvé.

Il envoie ensuite des messages de type IntegerMessage à cet agent cible, contenant la valeur entière 1. Chaque envoi est espacé d'un délai aléatoire (entre 100 ms et 1100 ms).

Une fois les messages envoyés, l'agent termine son exécution.

- L'agent **CounterAgent** joue un rôle critique dans l'équilibre de la simulation. Il agit comme une passerelle entre les émetteurs et le contrôleur en transmettant des informations agrégées tout en assurant la continuité du cycle par la création de nouveaux émetteurs. Sa conception garantit une dynamique fluide et maintient un nombre équilibré d'agents dans le système.

Activation (activate) :

L'agent rejoint une société artificielle en créant ou rejoignant un groupe nommé "communication" avec le rôle "RolePlay".

Il s'identifie spécifiquement en tant que "CounterAgent".

Un délai (pause (500)) permet d'assurer la synchronisation du système avant de commencer son exécution.

Phase de vie (live) :

Une limite de temps de transformation (transformationTimeLimit) est définie aléatoirement, comprise entre 5 et 15 secondes à partir du moment où l'agent démarre.

Tant que cette limite n'est pas atteinte (méthode shouldTransform), l'agent attend les messages entrants à l'aide de waitNextMessage.

Si un message reçu est une instance de IntegerMessage, son contenu est ajouté à un compteur interne (this.counter).

Une fois la limite atteinte :

L'agent envoie un message à un ControllerAgent contenant le total accumulé.

Il crée et lance un nouvel EmitterAgent pour assurer la continuité des émetteurs, tout en se terminant.

- L'agent **ControllerAgent** joue un rôle central en agissant comme un régulateur dans le système. Il s'assure que le nombre d'agents compteurs reste constant, tout en relayant l'information des anciens compteurs vers les nouveaux. Cela garantit une continuité opérationnelle et maintient une structure dynamique dans le cadre de la simulation. Ce comportement est essentiel pour éviter une perte d'agents compteurs ou une surcharge d'émetteurs, assurant ainsi la stabilité et l'équilibre du système.

Activation (activate) :

L'agent rejoint la société artificielle en créant ou en rejoignant un groupe nommé "communication", avec le rôle "RolePlay".

Il s'identifie spécifiquement en tant que "ControllerAgent".

Une pause (pause (500)) permet de synchroniser le système avant l'exécution active de l'agent.

Phase de vie (live) :

L'agent entre dans une boucle continue pour traiter les messages entrants.

Il utilise waitNextMessage pour recevoir des messages envoyés par les CounterAgent lorsqu'ils signalent leur transformation imminente.

Si le message reçu est une instance de IntegerMessage, il extrait la valeur (le compteur) et utilise launchAgent pour créer un nouvel agent CounterAgent avec cette valeur initiale.

Ce mécanisme garantit que le remplacement des compteurs est immédiat et fluide.

.

Cette partie a démontré l'efficacité d'un système multi-agents avec des rôles dynamiques. Les interactions entre EmitterAgent, CounterAgent, et ControllerAgent assurent un équilibre autonome : les émetteurs envoient des messages, les compteurs les collectent et se transforment en émetteurs après notification au contrôleur, qui remplace les compteurs disparus.

Le système fonctionne de manière fluide et adaptable grâce à :

Une communication inter-agents basée sur des messages. Une gestion dynamique des transformations et remplacements via le contrôleur.

Ce modèle offre une base robuste pour des applications distribuées nécessitant équilibre et flexibilité.

5 Adaptation of Bees

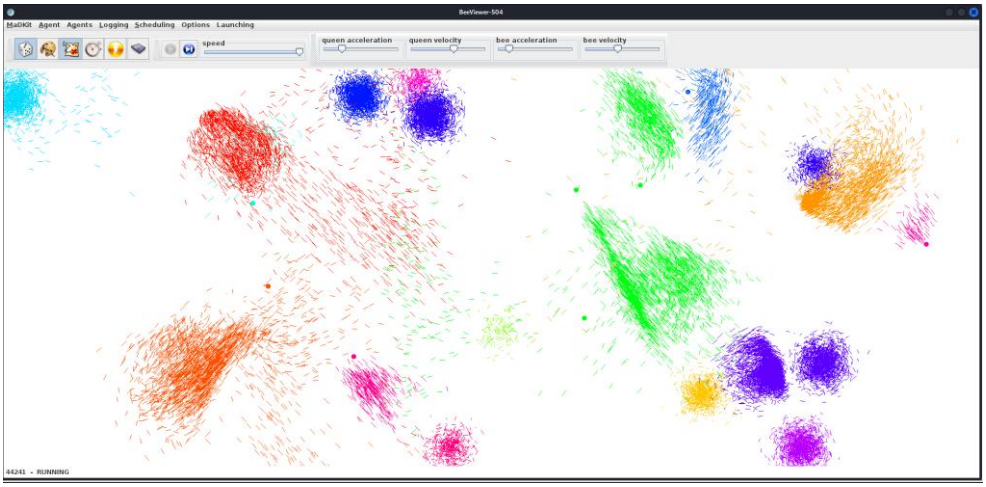
1. Pour adapter le code afin de représenter les reines par des cercles, nous allons ajouter une condition pour vérifier si l'abeille est une reine et utiliser fillOval pour dessiner un cercle à la place de la ligne. Voici comment adapter la méthode computeFromInfoProbe qui se trouve dans la classe BeeViewer qui est chargée de l'affichage dans la simulation :
   - Utilisation de instanceof : On vérifie si arg0 (l'abeille actuelle) est une instance de QueenBee avec if (arg0 instanceof QueenBee).
   - Dessin d'un cercle pour les reines : Si arg0 est une instance de QueenBee, on dessine un cercle centré sur la position de la reine avec fillOval.
   - Dessin des autres abeilles : Si l'abeille n'est pas une reine, elle est dessinée comme une ligne, comme dans le code original.

```
private void computeFromInfoProbe(Graphics g) {
    g.drawString("You are watching " + beeProbe.size() + " MaDKit agents", 10, 10);
    Color lastColor = null;
    final boolean trailMode = (Boolean) trailModeAction.getValue(Action.SELECTED_KEY);

    for (final AbstractBee arg0 : beeProbe.getCurrentAgentsList()) {
        final BeeInformation b = beeProbe.getPropertyValue(arg0);
        final Color c = b.getBeeColor();

        // Change de couleur si nécessaire
        if (c != lastColor) {
            lastColor = c;
            g.setColor(lastColor);
        }

        final Point p = b.getCurrentPosition();

        // Utiliser instanceof pour vérifier si l'abeille est une reine
        if (arg0 instanceof QueenBee) {
            int diameter = 10; // Taille du cercle pour la reine
            g.fillOval(p.x - diameter / 2, p.y - diameter / 2, diameter, diameter);
        } else {
            if (trailMode) {
                final Point p1 = b.getPreviousPosition();
                g.drawLine(p1.x, p1.y, p.x, p.y);
            } else {
                g.drawLine(p.x, p.y, p.x, p.y);
            }
        }
    }
}
```

2. Pour augmenter la durée de vie des reines, il faut modifier les conditions de suppression des reines dans la méthode live de la classe BeeLauncher chargée du lancement ou de l'arrêt d'une abeille dans la simulation. Dans ce code, la durée de vie des reines est influencée par les appels à killBees(true, ...), où true indique que ce sont les reines qui sont concernées.
   Pour prolonger la durée de vie des reines, on va :
   - Diminuer la probabilité de suppression des reines.
   - Réduire le nombre de reines supprimées à chaque appel.
   - Augmenter la probabilité de création de nouvelles reines dans launchQueens

```
@Override
protected void live() {
while (isAlive()) {
    Message m = waitNextMessage(500 + (int) (Math.random() * 2000));
    if (m != null) {
    proceedEnumMessage((EnumMessage<?>) m);
    }
    if (randomMode) {
    killBees(false, 150);
    if (Math.random() < .8) {
        if (Math.random() < .3) { // Réduction de la probabilité de suppression des reines
        if (queensList.size() > 1)
            if (queensList.size() > 10)// Augmentation du seuil avant suppression
            killBees(true, (int) (Math.random() * 4) + 1); // Réduction du nombre de reines supprimées
            else
            killBees(true, 1); // Suppression d'une seule reine si la liste est petite
        }
        else if (queensList.size() < 10)
        launchQueens((int) (Math.random() * 2) + 1);
    }
    else if (Math.random() < .3) {
        if (beesList.size() < 200000 && Runtime.getRuntime().freeMemory() > 100000) {
        launchBees((int) (Math.random() * 15000) + 5000);
        }
    }
    else {
        killBees(false, (int) (Math.random() * 500) + 1);
    }
    }
}
}
```

Et voici la simulation avec un fond blanc (modification à faire dans BeeViewer) :



Toutefois, nous préférons maintenir le fond noir que nous apprécions bien.

6 Prey and Predators

Pour implémenter les comportements des frelons, nous allons ajouter les classes et méthodes nécessaires pour simuler les interactions entre les abeilles et les frelons. Voici comment procéder étape par étape :

Définir une classe pour les frelons

Ajoutez une nouvelle classe appelée Hornet qui hérite de AbstractBee. Les frelons auront leurs propres comportements (comme chasser les abeilles et réagir à leur proximité).

```
public class Hornet extends AbstractBee {
```

Dans la classe BeeViewer, ajouter l'affichage du frelon comme un gros cercle, trois fois plus gros qu'une reine.

```
if (arg0 instanceof Hornet) {
    int diameter = 30; // Taille du cercle pour le frelon
    g.fillOval(p.x - diameter / 2, p.y - diameter / 2, diameter, diameter);
}

// Utiliser instanceof pour vérifier si l'abeille est une reine
if (arg0 instanceof QueenBee) {
    int diameter = 10; // Taille du cercle pour la reine
    g.fillOval(p.x - diameter / 2, p.y - diameter / 2, diameter, diameter);
} else {
    if (trailMode) {
        final Point p1 = b.getPreviousPosition();
        g.drawLine(p1.x, p1.y, p.x, p.y);
    } else {
        g.drawLine(p.x, p.y, p.x, p.y);
    }
}
```

Modifier les différentes classes en y ajoutant certains attributs ou méthodes afin de prendre compte les frelons. (Nous calquons à partir de ce qui existe déjà pour les abeilles simples et les reines.

Pour les actions du frelon durant sa vie nous avons ajouté les actions qu'il réalise à chaque « buzz ».

Nous avons la liste de toutes les abeilles qui est dans la classe BeeLauncher et qui est maintenu tout au long de la simulation (beesList).

```
private ArrayList<AbstractAgent> queensList = new ArrayList<>();
private ArrayList<AbstractAgent> beesList = new ArrayList<>(INITIAL_BEES_NB * 2);
```

Cette liste peut être pour nous une espèce d'annuaire dans lequel nous pouvons trouver toutes les abeilles à l'instant t et calculer la distance entre frelon et abeille (computeDistance). Même si le nombre d'abeilles est élevé cela est quand même aisé et rapide grâce aux streams en

Java. Toutefois la liste beesList contient des objets AbstractAgent à partir desquelles, on ne peut pas obtenir les informations sur l'abeille (la postion). Ainsi, à chaque mise à jour de cette liste, nous allons ajouter les références de cette liste de AbstractAgent en liste de AbstractBee dans une liste « bees » que nous allons utiliser pour calculer les distances.

```
public static ArrayList<AbstractAgent> beesList = new ArrayList<>(INITIAL_BEES_NB * 2);
public static CopyOnWriteArrayList<AbstractBee> bees = new CopyOnWriteArrayList<>();
```

Nous utilisons CopyOnWriteArrayList car BeeLauncher.bees est fréquemment modifiée par plusieurs threads, donc il nous fallait utiliser une collection thread-safe.

```
private void launchBees(int numberOfBees) {
getLogger().info(() -> "Launching " + numberOfBees + " bees");
//greatly optimizes the launching time
final List<AbstractAgent> beesBucket = launchAgentBucket(
    Bee.class.getName(),
    numberOfBees,
    COMMUNITY + "," + SIMU_GROUP + "," + BEE_ROLE,
    COMMUNITY + "," + SIMU_GROUP + "," + FOLLOWER_ROLE);
beesList.addAll(beesBucket);
beesBucket.stream()
.filter(Bee.class::isInstance) // Vérifie que l'agent est une instance de Bee
.map(Bee.class::cast) // Cast l'agent en Bee
.forEach(bees::add);
}
```

Les objets dans beesBucket ne sont pas copiés ou recréés. Ils restent les mêmes références, simplement castées dans le type Bee au lieu de AbstractAgent.

Après tout ça, nous pouvons enfin implémenter le comportement de nos frelons chasseurs d'abeilles :

```
@Override
   public void buzz() {
    final List<AbstractBee> listNearBees;

    synchronized (BeeLauncher.bees) {
           listNearBees = BeeLauncher.bees.stream()
               .filter(e -> computeDistance(e.myInformation.getCurrentPosition(), this.myInformation.getCurrentPosition()) < 10)
               .collect(Collectors.toList());
       }
    int beeCount=listNearBees.size();
    if (beeCount > 7) {
           // Frelon entouré par trop d'abeilles, il meurt
           getLogger().info(()->"Le frelon est entouré par plus de 7 abeilles ! Mort dans 5 secondes...");
           killAgent(this,5);
       } else if (beeCount > 4) {
           // Frelon entouré par 5 à 7 abeilles, il ne peut plus attaquer
           getLogger().info(()->"Merde, il y en a trop ! !");
       } else if (beeCount > 0) {
           getLogger().info(()->"Le frelon va tuer !"+listNearBees.get(0));
           // Frelon attaque une abeille proche
           killAgent(listNearBees.get(0),3);
       }

   super.buzz();
   }
```

Et notre simulation fonctionne bien avec le comportement souhaité des frelons selon les cas de figure. Le temps nous est un peu compté sinon, nous avons pleins d'idées d'améliorations

dans les schémas d’attaques, d’interactions et de communications pour cette version de la simulation que nous pouvons ajouter et que nous vous présenterons lors de notre présentation.