Grenoble INP UGA
ENSIMAG
Modélisation mathématique, Image et Simulation

TP en temps libre de Programmation orientée objet

Simulation d'une équipe de robots pompiers

Réalisé par :
Er-rabhi Imane
Mohammed Reda Belfaida
Taha Aftiss

Année : 2024/2025

Table des matières

Chapitre 1

Première partie : les données du problème

Introduction

La première partie du projet se concentre sur la création et la mise en place des classes nécessaires pour simuler une équipe de robots pompiers. Ces classes comprennent la représentation de la carte, des cases, des incendies et des robots. Chaque entité est modélisée pour fournir les informations basiques afin de visualiser et gérer les éléments de simulation.

Classes Principales

Les classes clés dans cette partie sont Carte, Case, Incendie, Robot et LecteurDonnees. Ces classes sont conçues pour représenter les éléments statiques de la simulation.

1. **Classe `Carte`** : La classe `Carte` organise la structure de la carte en une matrice de cases. Elle permet de connaître les dimensions de la carte, la taille de chaque case, et d'accéder aux

cases selon leurs coordonnées. La méthode `getVoisin` sert à trouver la case voisine dans une direction donnée, tout en respectant les limites de la carte.

- **Attributs** : `tailleCases` (dimension de chaque case en mètres), `matrice` (matrice de cases).
- **Méthodes clés** :
  - `getCase(int i, int j)` : retourne la case aux coordonnées spécifiées.
  - `voisinExiste(Case src,Direction dir)` vérifie si un voisin existe dans une direction donnée.
  - `getVoisin(Case src, Direction dir)` : retourne le voisin d'une case donnée en fonction de la direction.

2. **Classe `Case`** : Cette classe représente une case individuelle de la carte, qui est définie par ses coordonnées et la nature de son terrain (eau, forêt, roche, habitat, etc.). Elle comprend des méthodes pour obtenir et définir les propriétés de la case.
   - **Attributs** : `ligne` et `colonne` (coordonnées de la case), `nature` (type de terrain).
   - **Méthodes clés** :
     - `getNature()` : retourne le type de terrain de la case.
     - `setNature(NatureTerrain nature)` : définit la nature de la case, utilisée pour initialiser les terrains lors de la lecture de la carte.
3. **Classe `Incendie`** : Cette classe modélise un incendie sur une case spécifique, avec une intensité qui correspond au volume d'eau nécessaire pour éteindre le feu.
   - **Attributs** : `position` (case où l'incendie est situé), `intensite` (volume d'eau requis pour l'éteindre).
   - **Méthodes clés** :

- `getIntensite()` : retourne l'intensité de l'incendie.
- `getPosition()` : donne la position de l'incendie sur la carte.

4. **Classe `Robot`** (classe abstraite) : Elle est définie comme une classe de haut niveau pour représenter différents types de robots. Chaque robot a des attributs communs, comme la quantité d'eau qu'il peut transporter et la position sur la carte. Les méthodes dans cette classe abstraite sont destinées à être spécialisées pour chaque type de robot.
   - **Attributs communs** : `position` (case sur laquelle le robot est situé), `eau` et `eauMax` (quantité d'eau actuelle et maximale), `vitesse` (vitesse de déplacement).
   - **Sous-classes** : `Drone`, `RobotRoues`, `RobotChenilles`, `RobotPattes` implémentent des caractéristiques spécifiques à chaque type de robot, comme les terrains accessibles et la vitesse.
5. **Classe `LecteurDonnees`** : Cette classe est responsable de lire les données d'un fichier texte et de les transformer en objets utilisables dans la simulation. Elle parse les informations de la carte, des cases, des incendies, et des robots pour les ajouter dans une instance de `DonneesSimulation`.
   - **Méthodes clés** :
     - `lireCarte()` : lit la structure de la carte depuis le fichier.
     - `lireCase(int lig, int col)` : initialise chaque case avec son type de terrain.
     - `lireIncendies()` et `lireRobots()` : lisent et ajoutent les incendies et les robots avec leurs caractéristiques.

Gestion des Données Initiales avec DonneesSimulation

La classe `DonneesSimulation` regroupe les informations sur la carte, les incendies et les robots pour faciliter la modularité et l'accès par d'autres parties de la simulation. Elle propose des méthodes pour ajouter ou retirer des incendies et des robots, ainsi qu'une méthode `afficher()` pour présenter le contenu chargé à des fins de vérification.

Chapitre 2

Simulation de scénarios

Introduction

La documentation suivante décrit le fonctionnement des scénarios dans une simulation de robots et de gestion d'événements. Chaque scénario repose sur une série d'événements représentés par des classes Java. Les classes permettent de modéliser les déplacements, les extinctions d'incendies et les remplissages des robots en interaction avec une carte.

L'objectif est de simuler et documenter le comportement de ces entités en respectant des contraintes spécifiques.

Classe `Evenement`

La classe `Evenement` est une classe abstraite représentant un événement générique. Elle sert de base pour tous les autres types d'événements.

Attributs

- `date` : *long* – La date de l'événement.

Méthodes

- `getDate()` : Renvoie la date de l'événement.
- `setDate(long date)` : Modifie la date de l'événement.
- `execute()` : Méthode abstraite pour exécuter l'événement.
- `clone()` : Clone l'objet.

`EvenementMessage`

Description

Cette sous-classe représente un événement qui affiche un message spécifique à une date donnée.

Constructeur

```
public EvenementMessage(int date,
String message)
```

- `date` : La date de l'événement. - `message` : Le message à afficher.

Méthodes

- `public void execute()` : Affiche la date et le message dans la console.

Deplacement

Description

Cette sous-classe représente un déplacement d'un robot soit vers une direction spécifique, soit vers une case donnée.

Constructeurs

```
public Deplacement(long date, Carte carte,
ArrayList<Robot> robots, int indice,
TypesEnums.Direction direction)
```

- Déplace le robot dans une direction donnée.

```
public Deplacement(long date, Carte carte,
ArrayList<Robot> robots, int indice,
Case destination)
```

- Déplace le robot vers une case spécifique.

Méthodes

- `public void execute()` : Exécute le déplacement basé sur la direction ou la case de destination.
- `public void executeWith( TypesEnums.Direction direction)` : Gère les contraintes et effectue le déplacement en direction.
- `public void executeWith(Case destination)` Déplace directement le robot vers la destination.

— `public String toString()` : Retourne une description du déplacement.

`Extinction`

Description

Cette sous-classe représente l'extinction d'un incendie par un robot.

Constructeur

```
public Extinction(long date,
ArrayList<Incendie>
incendies, ArrayList<Robot> robots,
int indice)
```

- `date` : La date de l'événement. - `incendies` : Liste des incendies. - `robots` : Liste des robots. - `indice` : Indice du robot impliqué.

Méthodes

— `public void execute()` : Identifie l'incendie à éteindre et effectue l'opération d'extinction.
— `public String toString()` : Retourne une description de l'opération d'extinction.

`Remplissage`

Description

Cette sous-classe représente un événement où un robot remplit son réservoir d'eau.

Constructeur

`public Remplissage(long date, Carte carte, Robot robot)`

- `date` : La date de l'événement. - `carte` : La carte du système.
- `robot` : Le robot impliqué.

Méthodes

- `public void execute()` : Vérifie les conditions et remplit le réservoir d'eau du robot.
- `private boolean hasAdjacentWater(Case position)` : Vérifie si une case adjacente contient de l'eau.

Chapitre 3

Calculs de plus courts chemins

Introduction

La troisième partie du projet porte sur l'implémentation d'algorithmes pour calculer les plus courts chemins pour les robots dans la simulation. Cette étape est cruciale pour que les robots puissent se déplacer efficacement vers leurs destinations, que ce soit pour éteindre un incendie ou recharger leur réservoire.

Structure et Fonctionnalités

Pour cette partie, plusieurs classes et méthodes ont été développées ou modifiées pour implémenter ces fonctionnalités :

Algorithme de Calcul du Plus Court Chemin

L'algorithme utilisé est une variante de Dijkstra, adaptée pour travailler avec les contraintes spécifiques des robots et de la carte. Cet algorithme trouve le chemin le plus rapide en tenant compte :

— des terrains accessibles pour chaque type de robot ;

— des vitesses spécifiques des robots sur différents terrains ;
— des obstacles présents sur la carte.

Classe `Dijkstra`

Cette classe implémente l'algorithme de Dijkstra et inclut les méthodes suivantes :

1. `recherchePlusCourtChemin(Robot robot, Case depart, Case destination)` : Calcule et retourne la liste des cases constituant le chemin le plus court entre deux positions.
2. `initGraph()` : Initialise le graphe en prenant en compte les voisins de chaque case et les vitesses associées.
3. `calculChemin()` : Implémente la logique de l'algorithme en assignant des coûts à chaque case et en construisant le chemin optimal.

Classe `Simulateur`

Le calcul du chemin le plus court est intégré dans la classe `Simulateur`, qui utilise l'algorithme pour planifier les déplacements des robots :

1. `planifierDeplacement(ArrayList<Robot> robots, int index, Case destination)` : Cette méthode utilise l'algorithme pour générer une liste de déplacements pour un robot donné, ajoutés à la liste d'événements du simulateur.
2. Les déplacements sont représentés par des instances de la

classe `Deplacement`, qui sont ajoutées à la liste des événements de simulation.

Événements de Déplacement

La classe abstraite `Evenement` est spécialisée pour inclure une classe `Deplacement`, utilisée pour modéliser chaque étape du chemin parcouru par un robot. Chaque déplacement :

- met à jour la position du robot sur la carte ;
- est exécuté au bon moment, en fonction de la date actuelle de la simulation.

Optimisation

Pour éviter des calculs répétés, l'algorithme utilise un cache pour stocker les résultats des chemins déjà calculés. Cela améliore la performance, surtout dans les simulations complexes avec plusieurs robots et incendies.

Test et Validation

Pour tester cette partie, plusieurs scénarios prédéfinis ont été implémentés :

- Déplacement d'un robot vers une case voisine pour vérifier la validité du calcul des voisins.
- Calcul de chemin à travers plusieurs types de terrains pour tester l'intégration des vitesses spécifiques des robots.
- Simulation de l'évitement des obstacles, comme des terrains inaccessibles.

Ces tests ont confirmé que l'algorithme gère correctement les cas valides et rejette les chemins impossibles.

Chapitre 4

Stratégie & Mise en Application

Introduction

La méthode `strategieElementaire` est utilisée pour gérer les incendies dans une simulation. Elle attribue des robots aux incendies de manière élémentaire en suivant une logique basée sur la disponibilité et les positions des robots et des incendies. La stratégie met également à jour l'état des robots en fonction de leur progression dans la tâche.

Étapes Principales

Voici une explication étape par étape de la méthode :

1. Parcourir la liste des incendies

- La méthode parcourt tous les incendies présents dans la simulation. - Si l'intensité d'un incendie est de 0, il est ignoré car il est considéré comme éteint.

2. Vérifier si un incendie est déjà attribué à un robot

- Si un incendie n'est pas encore attribué (indiqué par la valeur `-1` dans la liste `IncendieAssigned`), la méthode tente de trouver un robot disponible pour l'attribuer.

3. Affecter un robot à un incendie

- Les robots sont parcourus pour trouver un robot avec l'état FREE. - Une fois trouvé, le robot est mis en état MOVING et un déplacement vers la position de l'incendie est planifié à l'aide de la méthode `planifierDeplacement` de la classe `Simulateur`. - Si le déplacement est planifié avec succès, l'incendie est assigné à ce robot.

4. Gérer les robots déjà assignés

- Si un robot est déjà assigné à un incendie :

— Si le robot atteint la position de l'incendie et est en état MOVING, il passe à l'état `FIREFIGHTING`. La méthode `ajouteExtinction` est appelée pour débuter l'extinction de l'incendie.

— Si le robot est en état RECHARGING, il est dirigé vers un point d'eau (le plus proche pour les robots non drones). Une fois rechargé, il retourne à l'incendie pour continuer l'extinction.

Diagramme des États du Robot

- **FREE** : Le robot est disponible pour être assigné.
- **MOVING** : Le robot se déplace vers un incendie.
- **FIREFIGHTING** : Le robot est en train d'éteindre un incendie.
- **RECHARGING** : Le robot se dirige vers un point d'eau pour remplir son réservoir.

Points Importants

- La méthode assure une répartition élémentaire des robots, sans optimisation complexe.
- Le système vérifie que les robots se déplacent correctement vers les incendies et effectuent leur tâche en fonction de leur état.
- Les points d'eau sont calculés lors de l'initialisation et sont utilisés pour recharger les robots si nécessaire.

Chapitre 5

Annexe

Compilation des scripts

Pour compiler et exécuter les scripts Java associés à ce projet, il est recommandé d'utiliser **IntelliJ IDEA**, qui gère automatiquement la compilation des fichiers source. Voici les étapes à suivre :

1. **Ouvrir le projet dans IntelliJ IDEA** : Importez le projet dans l'IDE en utilisant l'option d'ouverture de projet.
2. **Compilation automatique** : IntelliJ IDEA compile automatiquement les fichiers Java dès qu'ils sont enregistrés. Vous n'avez pas besoin de spécifier des commandes manuelles de compilation, car l'IDE gère cela en arrière-plan.
3. **Exécution** : Pour exécuter un script, sélectionnez la classe principale (celle contenant la méthode `main`) et cliquez sur le bouton "Run" dans la barre d'outils ou utilisez le raccourci `Shift+F10`.
4. **Répertoire de compilation** : Les fichiers compilés seront générés dans le répertoire `out` du projet, sous le sous-dossier `production`.

En cas de modifications importantes dans la configuration du projet, vous pouvez forcer la recompilation en allant dans **Build > Rebuild Project**.

Pour faciliter la compilation de votre projet en ligne de commande, un **Makefile** est à votre disposition. Celui-ci vous permet de compiler l'ensemble des fichiers sources en exécutant une simple commande.

Compilation par ligne de commandes

1. Assurez-vous que l'utilitaire `make` est installé sur votre système. 2. Ouvrez un terminal dans le répertoire racine de votre projet. 3. Utilisez la commande suivante pour compiler :

```
make all
```

Cela compilera tous les fichiers nécessaires et générera les fichiers binaires correspondants. 4. Pour exécuter un programme spécifique, vous pouvez utiliser les commandes définies dans le `Makefile`, par exemple :

```
make exeScenario1
```

Nettoyage des fichiers compilés

Pour supprimer les fichiers compilés et revenir à un état propre, utilisez la commande :

```
make clean
```

Note : Assurez-vous que le fichier `Makefile` est correctement configuré et correspond à la structure du projet.