Rapport ProjetRobotsPompiers

Equipe 31

November 2024

1 Introduction

Dans le cadre de ce projet, nous avons développé une simulation en Java pour modéliser et gérer les interventions d'une équipe de robots pompiers dans un environnement naturel, sous la supervision d'un chef pompier. L'objectif est de permettre à ces robots de se déplacer sur une carte, d'éteindre des incendies et de gérer leurs ressources de manière autonome, en s'appuyant sur une modélisation claire des entités (carte, incendies, robots), une interface graphique interactive, et un simulateur d'événements discrets.

2 Partie 1: Implémentation des classes de base et affichage des données

Nous avons développé les classes nécessaires à la modélisation des entités principales du problème, telles que la carte, les cases, les incendies et les robots. Ces classes ont été conçues en suivant les principes de modularité et d'encapsulation, avec l'utilisation de types énumérés pour représenter les éléments fixes comme les natures de terrain et les directions. L'intégration de l'interface graphique (GUISimulator) permet d'afficher visuellement les données initiales, en positionnant correctement les robots, les incendies et les différents types de terrains sur la carte.

La classe principale 'DonneesSimulation' centralise toutes les données, servant de point d'entrée pour manipuler les entités du système. Pour valider cette première étape, un programme de test (TestAffichage) a été mis en place pour charger un fichier de description conforme au format spécifié et afficher l'état initial de la simulation. Cette phase constitue les fondations nécessaires au développement des fonctionnalités de simulation et de gestion des événements dans les étapes suivantes.

3 Partie 2 : Simulation des scénarios

Dans cette étape, nous avons conçu et testé un simulateur d'événements discrets capable de gérer dynamiquement une liste ordonnée d'événements (comme

`Deplacement`, `Versement`, ou `Remplissage`). La classe `Simulateur` permet d'ajouter, d'exécuter et de supprimer les événements une fois qu'ils ont été réalisés, tout en offrant une fonctionnalité de redémarrage pour réinitialiser la simulation. Ces événements sont programmés en fonction de leur date, et leur exécution est synchronisée avec l'avancée du temps dans la simulation.

Pour visualiser les scénarios et interagir avec la simulation, la classe `SimulationVersion1` connecte le simulateur à une interface graphique interactive. Les boutons "Suivant" et "Début" permettent de naviguer dans la simulation et de la réinitialiser respectivement, tout en mettant à jour l'affichage en temps réel. Les tests, réalisés à l'aide de la classe `TestScenario`, ont confirmé la cohérence et la précision de la simulation, avec une gestion efficace des événements, une représentation fidèle des données, et une interception appropriée des erreurs. Cette phase établit les bases pour des extensions futures, comme l'intégration d'algorithmes de calcul de chemins optimaux ou de stratégies avancées pour coordonner les actions des robots.

4 Calcul des plus courts chemins

Dans cette partie du projet, un algorithme basé sur la classe `Dijkstra` a été implémenté pour permettre aux robots de calculer et suivre les plus courts chemins sur la carte, en prenant en compte les propriétés spécifiques des robots et les caractéristiques du terrain. Le graphe, représenté par la classe `Graphe`, est construit dynamiquement à partir d'une carte et d'un robot donné. Chaque case de la carte est modélisée comme un nœud, et chaque déplacement possible entre deux cases traversables est représenté par une arête via la classe `Arete`, dont le poids est calculé grâce à la méthode privée `calculerPoids`. Ce poids dépend de la vitesse moyenne du robot sur les cases adjacentes et de la taille des cases.

La méthode principale `calculerPlusCourtChemin` de la classe `Dijkstra` permet de déterminer le chemin optimal vers une destination donnée. Elle retourne un objet de type `ResultatChemin`, qui contient la liste ordonnée des cases constituant le chemin, le temps total nécessaire pour l'atteindre, et un détail des temps associés à chaque étape via la méthode `getNormalizedData` de la classe `Normalisation`.

Pour garantir la cohérence et faciliter les vérifications, des méthodes comme `printAllCases` et `printAllAretes` de la classe `Graphe` permettent d'afficher toutes les cases et les arêtes du graphe. Enfin, une normalisation des temps de déplacement a été ajoutée grâce à la classe `Normalisation`, transformant les données en une échelle standardisée pour une meilleure analyse et comparaison des performances.

5 Partie 4 : Simulation finale avec coordination des robots

La méthode `next()` dans la classe `Simulation` orchestre le cœur de la simulation en appelant le chef pompier pour affecter les robots disponibles aux incendies non traités. Lors de chaque exécution, le chef pompier détermine le robot le plus proche pour chaque incendie en calculant le temps nécessaire pour atteindre l'incendie à l'aide de l'algorithme de plus court chemin. Une fois l'affectation réalisée, il génère une séquence d'événements comprenant le déplacement du robot vers l'incendie, l'intervention pour verser de l'eau, et le retour pour remplir son réservoir si nécessaire. Ces événements sont ensuite ajoutés au simulateur, qui les exécute chronologiquement à chaque pas de simulation, mettant ainsi à jour l'état de la carte et des entités.

Le chef pompier joue un rôle central dans la coordination des actions des robots. Il utilise des méthodes spécifiques pour gérer des cas complexes, comme le remplissage du réservoir lorsque le robot manque d'eau ou doit intervenir plusieurs fois sur le même incendie en raison de sa grande intensité. Par exemple, la méthode `RobotRemplissage` permet de planifier des allers-retours entre l'incendie et une source d'eau jusqu'à ce que l'incendie soit complètement éteint. De manière similaire, `RobotChemin` organise les étapes pour atteindre un incendie en prenant en compte les vitesses et contraintes du robot. Ces mécanismes permettent d'optimiser les ressources tout en assurant une gestion cohérente et dynamique des événements de la simulation.

Aspects à améliorer

Les aspects à améliorer dans notre code incluent : le temps de remplissage des robots et de vidage des réservoirs, ainsi que la complexité globale de l'implémentation.

Aspects pris en compte

"Les aspects pris en compte dans notre code incluent : le temps de déplacement de case en case en fonction de la vitesse et de la nature du terrain, la gestion de l'état du robot (marqué comme occupé lorsqu'il éteint un incendie ou se remplit), et l'attribution d'un incendie à un robot libre à chaque appel de la méthode next()."