Programmation Orientée Objet : Simulation de robots pompiers

Équipe 12

`!!! Remarque importante: Apres l'execution des fichiers veuillez attendre que les calculs soient finis avant de lancer lecture, cela s'affichera dans le terminal. Si vous appuyez sur lecture avant que les calculs ne soit terminé la simulation va se planter. !!!`

1 Choix de conception

1.1 Structure des sources

Pour notre projet, nous avons fait le choix de rassembler en différents packages nos sources pour les organiser :

- Le package `Classes` comporte les classes des objets que l'on va manipuler pendant la simulation (`Carte`, `Case`, `Direction`...) ainsi que `DonneesSimulation` qui represente les donnees qu'on a besoin pour simuler.
- Le package `Robot`,comporte notre classe abstraite de `Robot` ainsi que toutes les classes de robots qui en héritent.
- Le package `Evenements` comporte la classe abstraite `Evenement` ainsi que tous les événements qui en héritent.
- Le package `Tests` comporte tous les tests créés pour tester les différentes parties du sujet.
- Le package `ChefPompier` comporte la classe `StrategieElemenntaire` ainsi que `ChefPompier` qui implemente la strategie evoluee. en place les deux stratégies proposées par l'énoncé.
- Le package `Lecteur` gère les donnees en entree du probleme et les stocke grace aux deux classes : `LecteurDonnees` et `StockageDonnees`.
- Le package `cheminn` comporte la classe `CheminMIn` qui calcule le plus court chemin grace a l'algorithme de djikstra et la classe `DeplacerRobotMin` qui traduit ce plus court chemin en des evenements afin que le robot peut se delacer.
- le package `dessin` comporte les classes qui permettent de dessiner les donnes du probleme.
- Le package `Simulation` comporte la classe `Simulateur` qui implemente l'interface `Simulable` et qui permettent de simuler le probleme. Ainsi que la classe `LancerSimu` qui lance la simulation
- Le package `Intervention` decrit comment les robots interviennent lorsqu'il y a un incendie grace aux deux classes : `EteindreIncendie` et `ResultatIntervention`.
- Enfin, il y a le package `Helper` qui comporte la classe `Helper` et qui permet de calculer la duree des evenements ainsi que de choisir le robot avec la date minimale pour qu'il intervienne.

1.2 Simulation

- La classe `Simulateur`, située dans le package `simulation`, implémente l'interface `Simulable` et constitue le cœur du système de simulation. Elle est responsable de la gestion et de l'exécution des événements au cours du temps dans la simulation. Voici ses principales caractéristiques :
  - **Méthodes principales** :
    - `next()` : Exécute les événements planifiés pour la date actuelle et incrémente la date de la simulation. Les événements exécutés sont retirés de la liste, et l'interface graphique est mise à jour pour refléter l'état actuel des robots et incendies.
    - `restart()` : Redémarre la simulation en rechargeant les données initiales et relance la configuration initiale grâce à la méthode `LancerSimu.lancerSimu()`.
    - `simulationTerminee()` : Vérifie si la simulation est terminée en s'assurant que tous les événements ont été exécutés.

  Cette classe assure une orchestration fluide de la simulation, en intégrant la gestion du temps, l'affichage graphique et la réactivité aux stratégies définies. Elle offre une interface claire pour avancer dans la simulation (`next`) ou la redémarrer (`restart`), tout en fournissant des messages détaillés sur le déroulement des événements.
- La classe `LancerSimu` est responsable de l'initialisation et du lancement de la simulation à partir d'un fichier de données. Elle joue un rôle clé dans la configuration de l'environnement de simulation, en instanciant les objets nécessaires et en configurant l'interface graphique. Voici un résumé de ses fonctionnalités :

- **Méthode** `lancerSimu` : Cette méthode statique est utilisée pour démarrer la simulation. Elle prend en entrée le nom du fichier de données `fichier`, une instance de `GUISimulator` pour l'affichage graphique, et un booléen `StratEvoluee` pour choisir entre une stratégie élémentaire ou évoluée. La méthode crée une carte initiale, charge les données de simulation à partir du fichier, et initialise la simulation avec les robots affectés selon la stratégie choisie.
- **Création des objets essentiels** :
  - Un objet `Carte` est créé avec une matrice vide pour représenter la carte de simulation.
  - Les données de simulation sont stockées dans un objet `DonneesSimulation` qui est alimenté par la méthode `StockageDonnees.stock()` pour lire le fichier de données.
  - L'interface graphique `GUISimulator` est initialisée avec une couleur de fond noir par défaut si elle n'a pas été fournie.
- **Stratégie du chef pompier** : Selon la valeur du paramètre `StratEvoluee`, la méthode crée un chef pompier avec une stratégie élémentaire (`ChefPompierElementaire`) ou évoluée (`ChefPompierEvolue`). Le chef pompier affecte les robots en fonction de la stratégie choisie et un scénario est généré.
- **Gestion des erreurs** : Si un problème survient lors du chargement du fichier de données, une exception `FileNotFoundException` est levée si le fichier est introuvable, ou une exception `DataFormatException` est levée si les données sont mal formatées. Dans ce cas, un message d'erreur est affiché à l'utilisateur.

`Remarque` : On a fait le choix de ne pas afficher la taille reelle des cases dans les cartes car ils etaient tres grandes pour l'ecran de l'ordinateur. La taille qu'on affiche est : 1000/(nbre de cases + 2) que vous trouverez implémentée dans la méthode `transform` de la classe `Helper`.

1.3 Evenements

Le package `Evenements` contient les classes responsables de la gestion des événements dans la simulation. Chaque événement représente une action spécifique réalisée par un robot dans le cadre de l'intervention sur les incendies. Voici une description des principales classes :

- `Evenement` : Classe abstraite qui représente un événement générique dans la simulation. Toutes les autres classes d'événements, comme `Deplacer` et `DeverserEau`, en héritent. Elle contient les attributs et méthodes communs nécessaires à la gestion des événements.
- `DiminuerIntensite` : Cette classe change l'intensité d'une incendie et met a jour l'affichage de son intensité à la fin du versement de l'eau .
- `Deplacer` : Cette classe modélise le déplacement d'un robot d'une case à une case voisine. Elle prend en compte la taille réelle des cases et met à jour la position du robot en fonction des données de la carte.
- `DeverserEau` : Cette classe gère l'action de déverser de l'eau sur une case en feu. Elle est utilisée dans les stratégies élémentaire et évoluée pour éteindre les incendies. Bien que la stratégie élémentaire soit entièrement opérationnelle, des améliorations sont encore nécessaires pour la stratégie évoluée.
- `RemplirEau` : Cette classe modélise le remplissage du réservoir d'eau d'un robot, une action essentielle pour qu'il puisse poursuivre ses interventions.
- `Scenario` : Cette classe orchestre la séquence des événements dans la simulation. Elle contient l'ensemble des événements planifiés, leur ordre d'exécution, et permet de gérer la synchronisation entre les actions des différents robots.

Remarque : *Puisque notre approche se base sur la détermination des événements en amont du début de la simulation, nous avons conçu une classe* ***Scenario*** *qui contient une liste ordonnée d'événements. Plutôt que de transmettre les événements directement au simulateur, nous lui passons un* ***Scenario****, ce qui sépare la planification des événements de leur exécution.*

1.4 Algorithme de plus court chemin : Dijkstra

Pour permettre aux robots de trouver le chemin le plus rapide jusqu'à leur destination, nous avons conçu une fonction de recherche de chemin optimal basée sur une version adaptée de l'algorithme de Dijkstra. Chaque case de la carte est considérée comme un sommet dans un graphe pondéré, où le poids de chaque arête représente le temps nécessaire au robot pour se déplacer d'une case à une autre. Ce temps est calculé en fonction du type de robot et du type de terrain.

Voici le fonctionnement de l'algorithme :

- `Initialisation` : Une map 'temps' stocke le temps minimal nécessaire pour atteindre chaque case, initialisée à l'infini pour les cases accessibles (sauf la case de départ, qui est à 0). Les cases inaccessibles pour le robot sont marquées comme 'null'. Une map 'predecesseurs' enregistre le prédécesseur de chaque case pour reconstituer le chemin final.
- `Exploration` : On utilise une file de priorité 'frontiere' pour explorer les cases, en commençant par celle avec le temps minimal actuel. Pour chaque case, on examine ses voisins accessibles.

— `Mise à jour des temps` : Pour chaque voisin accessible, on calcule le temps de déplacement en fonction des vitesses du robot sur le terrain actuel et le terrain voisin. Si le temps calculé est inférieur à celui déjà enregistré, on met à jour 'temps' et on enregistre la case actuelle comme prédécesseur du voisin.
— `Résultat` : Lorsque la case d'arrivée est atteinte, l'algorithme retourne le chemin optimal en suivant les prédécesseurs depuis la case d'arrivée jusqu'à la case de départ.

La fonction a été codée dans 'CheminMinimum' et est appelée chaque fois que le robot doit se déplacer pour atteindre une case cible.

1.5 Intervention

Lors de la simulation, les événements sont créés avant d'être exécutés, ce qui implique que les données ne sont pas mises à jour en temps réel au moment de leur création. Pour remédier à cela, nous avons conçu la classe `ResultatIntervention` , qui fournit à la fois la liste des événements générés et les positions finales des robots après chaque intervention.

1.6 Le chef pompier

La classe `ChefPompier` est responsable de la gestion des robots dans une simulation afin de les affecter aux incendies et d'optimiser les interventions en fonction des conditions de la simulation. La méthode principale de cette classe, `AffecterRobots`, prend en compte deux stratégies d'affectation des robots pour éteindre les incendies, en utilisant des heuristiques basiques ou avancées.

1.6.1 Attributs principaux

Les attributs principaux de `ChefPompier` incluent :
— `donnees` : un objet `DonneesSimulation` qui contient les informations de la carte, des incendies et des robots disponibles ;
— `simulateur` : un objet `Simulateur` pour gérer l'ajout des événements créés par `ChefPompier` ;
— `robotsEnMissions` : une liste de booléens qui suit l'état d'occupation de chaque robot (libre ou en mission).

1.6.2 Méthode `AffecterRobots`

La méthode `AffecterRobots` génère un scénario en créant des événements permettant aux robots d'intervenir sur les incendies. Cette méthode intègre deux stratégies :
— **Stratégie basique** : Le premier robot disponible capable d'atteindre l'incendie est sélectionné. Si l'intervention est possible, le robot choisi est affecté à l'incendie. La position et la date de disponibilité du robot sont alors mises à jour, permettant de définir l'ordre des interventions. Durant l'intervention, le robot effectue tous les allers-retours nécessaires pour remplir son réservoir si celui-ci se vide. Enfin, si le réservoir du robot est vide à la fin de l'intervention, il se déplace automatiquement pour le remplir.
— **Stratégie évoluée** : Avant chaque affectation, les robots sont triés par leur date de disponibilité. Pour chaque incendie, la méthode sélectionne le robot dont le temps d'intervention (temps de déplacement plus temps d'extinction) est le plus court, permettant de réduire la durée totale des interventions.

1.6.3 Fonctionnement détaillé

Pour chaque incendie, `AffecterRobots` :

1. Parcourt la liste des robots pour trouver un robot capable d'intervenir, en respectant la stratégie choisie.
2. Calcule le temps nécessaire pour chaque robot disponible à l'aide de l'objet `EteindreIncendie`, qui utilise les positions des robots et des incendies pour estimer le temps total de l'intervention.
3. Ajoute les événements de déplacement et d'intervention dans une liste `events`, en mettant à jour la position finale du robot et sa date de disponibilité.

1.6.4 Mise à jour et création du scénario

Après traitement de tous les incendies, `AffecterRobots` :
— Crée un `Scenario` avec tous les événements générés et l'ajoute au simulateur pour orchestrer l'exécution de ces événements ;
— Réinitialise la position et le volume d'eau de chaque robot pour permettre la répétition du scénario avec des conditions de départ inchangées.

Cette organisation permet une gestion efficace et flexible des interventions des robots pour éteindre les incendies dans la simulation.

2 Tests effectués et résultats

2.1 Le fonctionnement des tests

2.1.1 Lancer les tests

Des fichiers de test sont conçus pour pouvoir effectuer des tests, notamment :
— `TestEtape2` : Simule les scénarios 0 (KO) et 1 (OK).
— `TestEtape3` : Teste l'algorithme du plus court chemin.
— `TestChefElementaire` : Teste l'algorithme d'intervention basique.
— `TestChefEvolue` : Teste l'algorithme d'intervention évolué.
Pour plus d'informations sur la compilation, veuillez vous référer au manuel d'utilisation.

2.2 Résultats obtenus

Comme le démontrent nos différents fichiers de tests, l'algorithme élémentaire fonctionne correctement sur toutes les cartes fournies, conformément aux exigences de l'énoncé. De plus, la stratégie évoluée a également été implémentée et s'avère pleinement fonctionnelle.

2.3 Limitations

Avant l'exécution, pour chaque paire (incendie, robot), l'algorithme détermine la case de remplissage d'eau accessible au robot et la plus proche de l'incendie. Ce processus oblige à évaluer toutes les combinaisons (robot, incendie), bien que ce ne soit pas nécessaire, car certains robots ne peuvent pas intervenir sur certains incendies, ce qui augmente significativement le temps de calcul.

2.4 Pistes d'amélioration

— Une premiere optimisation serait de chercher un algorithme plus performant pour trouver la case de remplissage d'eau – accessible au robot concerné – la plus proche pour chaque incendie.
— Une autre optimisation serait de faire intervenir plusieurs robots pour l'extinction d'un meme incendie afin de diminuer le temps d'intervention.