GRENOBLE INP UGA Ensimag

https://ensimag.grenoble-inp.fr/

Programmation Orientée Objet : Rapport de projet

Robots pompiers

Programmation Orientée Objet
Année universitaire 2022 - 2023

Table des matières

Introduction

Notre projet porte sur le développement d'une application en Java permettant de simuler une équipe de robots pompiers opèrant de manière autonome en milieu naturel. Nous allons dans ce rapport détailler notre approche, nos choix de conception et les stratégies mise en place pour répondre aux attentes de l'application. Ainsi, nous vous présenterons les principaux résultats grâce aux différents tests élaborés.

1 Choix de conception

1.1 Package `Carte`

Notre `Carte` est représentée par un tableau en deux dimensions de `Case`s. Chaque `Case` est définie par sa ligne et sa colonne. De plus, on garantit l'unicité de chaque `Case` car la création de `Case`s s'effectue uniquement lors de la génération de la `Carte`, et aucune méthode ne permet ensuite de modifier des `Case`s ou d'en créer de nouvelles.
Une `Carte` contient des méthodes pour vérifier l'existence des voisins d'une `Case` et les parcourir. Les `Incendie`s sont dans ce package et sont définis par leur `Case` position et leur intensité. Les `Incendie`s ainsi que les `Carte`s ont un attribut `DonneesSimulation` pour que chacune des deux classes ait accès à l'autre.
Ainsi une `Carte` ne manipule pas les `Incendie`s et inversement, seule `DonneesSimulation` centralise la gestion de données, tandis qu'`Incendie` et `Carte` ne sont que des structures de données.

1.2 Package `Robot`

`Robot` est une classe abstraite définit par des `DonneesSimulation`, une postion et un type qui lui confère des caractéristiques propres : une vitesse, un volume d'eau, un temps d'intervention linéaire et un temps de remplissage. Ils ont aussi un éventuel `Incendie` à laquelle ils pourraient être affectées et une disponibilité. Les classes filles se composent des 4 types de `Robots` : `PATTES, CHENILLES, ROUES, DRONE`. Chaque méthode liée à une action de `Robot` se situe dans ce package et respecte au maximum le principe d'héritage.

1.3 Package `Events`

Chaque `Evenement` est défini par la `Simulation` auquel il appartient, le `Robot` qui va effectuer cette tâche ainsi que la date où cet `Evenement` va être exécuté.
La classe abstraite `Evenement` possède une méthode abstraite `execute()` propre à chaque classe d'`Evenement`. Il arrive que lors de l'exécution d'un `Evenement` un nouvel `Evenement` similaire soit ajouté dans le scénario pour répéter cette action en boucle à intervalle régulier.

1.4 Package `Simulation`

1.4.1 Classe `DonneesSimulation`

Une classe `DonneesSimulation` centralise toute les données de la `Simulation`.
Les `Incendie`s sont sous forme de Hashmap<`Case`,`Incendie`>, nous permettant alors de vérifier si un `Incendie` est sur une `Case` donnée et d'y avoir accès avec une complexité constante. De par l'unicité des `Case`s et l'utilisation de la Hashmap, on garantie un seul `Incendie` par `Case`.

Les `Robots` sont stockés dans une Queue. En effet, il était impossible de mettre nos `Robots` dans une Hashmap < `Case`,`Robot` > , car dans ce cas il ne pouvait y avoir qu'un seul `Robot` par `Case`.

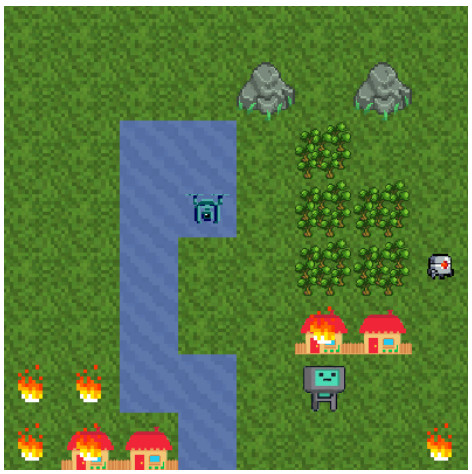
On implémente aussi une méthode renvoyant un itérateur sur les `Incendies`. Comme supprimer des éléments d'un itérateur pose problème, nous avons fait le choix de laisser un `Incendie` éteint dans la structure de données mais de complètement l'ignorer (on ne le dessine plus et on interagit plus avec lui). `DonneesSimulation` possède de plus un attribut `Carte`, ainsi qu'un attribut `fichierDonnees` pour relancer la `Simulation` avec le bouton Début (Restart).

1.4.2 Classe `Simulateur`

C'est dans la classe `Simulateur` que les `Evenements` sont ajoutés dans une PriorityQueue représentant le scénario. Le `Simulateur` va exécuter tous les `Evenements` qui ont leur attribut date égal à la date courante, puis dessiner la `Carte`, les `Robots` ainsi que les `Incendies`. Une fois fini, il incrémente la date courante.

1.5 Aspect Graphique

Il est possible d'afficher la `Carte` pour des petits tests basiques dans un terminal de commande grâce à la méthode `toString()` de `Carte` (cf 3.1). Nous avons d'abord implémenté une version 2D de la `Simulation`, puis une version en 3D isométrique. Les `Robots` et `Incendies` sont animés et la taille des `Incendies` s'adapte à leur intensité. Chaque `Case NatureTerrain` a différents assets possibles qui sont choisis aléatoirement, ainsi pour chaque `Simulation` la carte est unique.



(a) Version 1

(b) Version 2 Isométrique

FIGURE 1 – Aspect graphique de la `Simulation`

2 Stratégies mises en oeuvre

2.1 Calcul du plus court chemin

Les opérations et méthodes relatives au calcul du plus court chemin sont situées dans le package `Stratégie` et non pas avec les `Robots`. Ce principe de séparation que nous avons appliqué tout au long du projet facilite la maintenance du code, comme changer de stratégie rapidement.
Nous avons choisi d'implémenter l'algorithme de Dijkstra bien que l'algorithme A* aurait pu être envisagé. Une fonction Dijkstra prend en argument une `Case` de départ, de destination et un booléen chercheEau.

Il y a 3 cas possibles :

- 1^{er} **cas** : connaissant la `Case` de destination, chercheEau est définit à `false` et on renvoie le plus court chemin.
- 2^{e} **cas** : chercheEau est égal à `true`, on renvoie le plus court chemin vers la case EAU la plus proche.
- 3^{e} **cas** : chercheEau est à false et pas de destination n'est précisée (null) : on calcul alors tous les plus courts chemins vers toutes les `Case`s accessibles.

Ces 3 cas définissent les 3 fonctions `calculeChemin`, `cheminDestination`, et `cheminRemplir`.

2.2 Stratégie

2.2.1 Stratégie basique

On itère sur les `Incendies` :

1. Si il y a un `Robot` disponible, on lui affecte l'`Incendie` et on calcule le plus court chemin.
2. Le `Robot` sera disponible une fois l'`Incendie` éteint.
3. Si il n'y a plus de feu à affecter, les `Robots` vont rester immobiles.

2.2.2 Stratégie finale

On itère sur chaque `Robot` :

1. On lui demande s'il est disponible, si non alors on ne fait rien.
2. S'il est disponible, on calcul ses plus courts chemins aux `Incendies` et on lui affecte le plus proche.
3. Le `Robot` va alors se diriger vers le feu et sera indisponible jusqu'à ce que ce feu soit éteint. Le `Robot` rempli automatiquement son réservoir d'eau.

Optimisations de la stratégie

(a) Si tous les `Incendies` sont affectés, on envoie un `Robot` disponible sur un `Incendie` pourtant déjà pris en charge par un autre `Robot`.

(b) Si tous les `Incendies` restants sont inatteignables pour un `Robot`, alors ce `Robot` se dirige vers un `Incendie` atteignable déjà affecté.

(c) On a mit en place la propagation des `Incendies`. Chaque nouvel Incendie fils a une intensité égale à la moitié de son père. Quand un Incendie a une intensité inférieure à une valeur limite, il ne peut plus se propager. Il ne peut pas y avoir de propagation d'Incendie sur de l'eau.

3 Tests et principaux résultats

Afin de rendre notre code plus robuste et plus clair, nous avons créé de nouvelles exceptions : `TerrainIncorrectException`, `VitesseIncorecteException` et `VolumeEauIncorrectException`.

3.1 TestCarte

TestCarte est le premier test élaboré au sein de notre projet. Il permet d'afficher une `Carte` et de vérifier l'implémentation de `LecteurDonnees`.

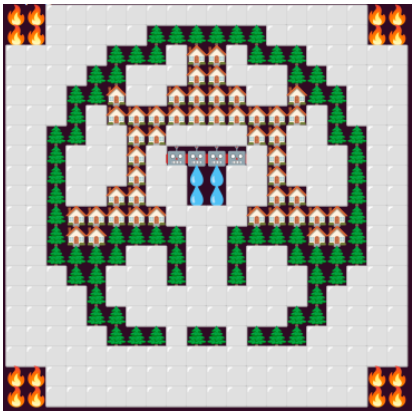


FIGURE 2 – Affichage dans shell

3.2 TestRobot

TestRobot nous permet de nous assurer la stabilité de nos différentes entités implémentées, avant d'aborder la notion de `Simulation` et d'`Evenements`. Le test nous génère une `Carte` simpliste diverses interactions sont possibles avec l'utilisateur afin de tester les différentes méthodes. Un seul `Robot PATTES` est présent sur la `Carte` et peut évoluer selon les entrées spécifiées.

3.3 TestDijkstra

TestDijkstra nous permet de tester notre implémentation du plus court chemin d'un `Robot` jusqu'à un `Incendie`. Le test nous affiche la carte Mushrooms of Hell (en version simpliste). On spécifie la `Case` où l'on souhaite se déplacer et on peut suivre notre `Robot` étape par étape.

3.4 TestSimulateurOK

TestSimulateur est le premier test implémentant la notion d'`Evenement`. Le test crée un `Simulateur` où sont ajoutés manuellement les évènements. Le test affiche donc l'enchaînement des `Evenement`s selon le scénario prédéfini, qui déplace le `Robot`, remplit son réservoir et déverse de l'eau sur un incendie (dont la position était préalablement connue).

3.5 TestChemin

TestChemin reprend le principe de TestDijkstra, en ajoutant la gestion d'ajouts successifs d'évenements de manière automatique à travers création d'un chemin.

3.6 TestSimulation

TestSimulation montre le test final de notre projet. Il expose les différentes parties du projet sur notre interface graphique finale. Ainsi, on peut observer une `Carte` avec des `Incendie`s d'intensité variables et différents `Robot`s. Une fois la `Simulation` lancée, on peut voir les `Robot`s évoluer au sein de la `Carte`, mettant en place la stratégie implémentée pour éteindre chaque `Incendie` le plus rapidement possible. La `Simulation` prend fin lors de l'extinction de tous les `Incendie`s.

3.7 TestStrategie

TestStrategie est initialement un test de débuggage interactif mais il nous a semblé pertinent de le laisser disponible. À travers ce test, nous pouvons générer une `Carte` avec un nombre d'`Incendie` spécifié, chacun à un emplacement choisi et à une intensité fixée, un nombre de `Robot` spécifié, chacun à un emplacement et selon le type spécifié. Une fois la `Carte` générée, on peut observer les `Robot`s mettant en place la stratégie calculée par notre `Simulation`, jusqu'à l'extinction complète des feux.

3.8 Test des optimisations

- Test de l'optimisation (a) (plusieurs `Robot`s sur un même `Incendie`) : visible sur le testSimulation.
- `exeOptimisationB` : Test de l'optimisation (b) (on ignore les `Incendie`s inatteignables.
- `exeSimulateurPropagation` : Test de l'optimisation (c) (test de la propagation des `Incendie`s)
- `./testCarte.sh <n> <PATH_TO_CARTE>` : script shell pour executer la simulation avec ou sans propagation pour n'importe quelle carte. Mettre n à 1 pour une simulation avec propagation, 0 sinon.
  Testez avec la map `desertOfDeath-20x20.map` pour voir que la propagation d' Incendies bat les `Robot`s.