ISI

Programmation Orientée Objet

Rapport TPL Java

CAFFIN Clement | CATHELINEAU Benjamin | EBOUKY Brown

1 Utilisation du programme

Il faut simplement le lancer la classe Test avec comme seul argument le chemin absolu du fichier de carte. Une javadoc est disponible dans le dossier javadoc

2 Présentation du travail réalisé

Pour la résolution de ce problème, nous avons mis en place un ensemble de classes dont la structure est donnée à la figure 1 de l'annexe.

2.1 Robot

Un robot est une classe **abstraite** qui renferme toute la logique relative à un robot (ses propriétés et ses méthodes). Nous avons en effet comme propriétés :

- **sa position** : il s'agit de la position que le robot occupe dans la carte. Cette position est elle même déterminée par une classe **Case** (position instance de cette classe)
- **sa position initiale** : il s'agit de la position initiale du robot au debut de la simulation. Cette position est utilisée pour pouvoir retrouver où se trouvait le robot au tout debut et donc d'être capable d'effectuer **un restart**(se remettre au debut de simulation et tout recommencer).
- **sa vitesse** : il s'agit de la vitesse du robot
- **son reservoir et reservoir initial** : qui sont respectivement le reservoir actuel du robot et le reservoir initial qu'il avait au debut de la simulation (ceci est utilisé pour les mêmes objectifs que la propriété position initiale).
- **son occupation** : permet de définir l'occupation du robot ; c'est à dire si à un instant un robot est entrain de faire une autre tâche et ne peut pas être sollicité. Elle est définie par la classe **OccupationRobot**.

Pour ce qui est des méthodes des robots, nous avons :

- **boolean peutRemplir(Carte carte)** : qui permet de dire si un robot peut se remplir à la position à laquelle il se trouve.
- **abstract boolean dureeDeversementUnitaire()** : méthode abstraite qui donne le temps pour le robot de déverser un 1L de volume d'eau sur une incendie.
- **abstract boolean peutAtteindre(Case position)** : méthode qui permet de dire si un robot peut atteindre une case ou non.
- les méthodes **resetPosition(DonneesSimulation donnees)** et **void resetReservoir()** qui permettent de remettre aux conditions initiales la position du robot et son reservoir.
- **chercherEau(Carte carte)** permet de calculer le temps le chemin optimal pour un robot pour pouvoir aller se remplir.

Il y a également des constantes que tout les robots définissent mais qui ne sont pas les même pour chaque robot. Pour faire cela nous avons fait des méthodes abstraites également en nous basant sur ce post stackOverflow [1]

- **getVitesseMax()**
- **getReservoirMax()**
- **tempRemplissage()**
- **getVitesseDefault()**

1. `https://stackoverflow.com/questions/11896955/force-subclasses-to-include-constant-in-abstract-java-class` Consulté le 20/11/2020

Les différentes implémentations de cette classe sont les classes : **Roue (couleur noir) , Patte (couleur jaune), Chenille (couleur rose), Drone (couleur magenta)** qui redéfinissent les différentes méthodes abstraites suivant leurs spécifications.

2.2 NatureTerrain

L'énumération des natures de terrain

2.3 Direction

L'énumération des Direction (NORD,SUD,EST,OUEST)

2.4 Incendie

La classe **Incendie** permet de représenter un incendie. Ses différentes propriétés sont : **la position** de l'incendie, **l'intensité**, ainsi que **l'intensité initiale**.

2.5 Case

Cette classe permet de définir la position à laquelle peut se trouver un robot, ou un incendie. Elle est principalement caractérisée par les **ligne** et **colonne** où se trouvent la case ainsi que la **nature** du terrain de la case.

2.6 Carte

Cette classe définit globalement la carte avec : un **tableau de cases**, la **taille d'une case** et les nombres de **ligne et colonne** de la carte. Elle fournit entre autres au moyen de son interface la possibilité de pouvoir déterminer le voisin d'une case suivant une direction à partir de la méthode **getVoisin(src, direction)**, ainsi qu'une méthode pour dire si oui ou non un voisin existe dans une direction donnée à partir d'une position grâce à la méthode **voisinExiste(src, direction)**

2.7 DonneesSimulation

Cette classe représente l'ensemble des données qui sont utilisées par un simulateur pour réaliser un ensemble d'évènement : les différents robots (une liste) dont on dispose, les incendies(une liste) et la carte toute entière.

2.8 Evenement

La classe **Evenement** est une classe abstraite qui concerne la logique relative aux différents évènements qui peuvent se faire c'est à dire : **déplacement, d'extinction de feu, de remplissage**. Les différents propriétés de cette classe sont :

— **date** : qui donne la date à laquelle doit se réaliser un évènement
— **Robot** : qui donne le robot sur lequel s'applique l'évènement

On distingue ainsi les classes concrètes qui héritent de cette classe : **Deplacement** qui permet de déplacer un robot vers une destination ; **ExtinctionFeu** qui permet à un robot d'éteindre un feu à la position à laquelle il se trouve ; **RemplissageReservoir** qui permet à un robot de se remplir ; tout ceci prend en compte les différentes spécifications relatives à chacun de ces évènements.

2.9 Simulateur

La classe **Simulateur** permet d'effectuer la simulation des différentes évènements proprement dit. Elle a comme propriétés entre autres : **l'interface graphique** associé à la simulation, **les données de simulations**, **la taille des cases** la **date de simulation courante** ,le **chef pompier** lorsqu'il y'en a un qui gère les différents évènements pour éteindre les incendies dans la carte ainsi qu'une **table de hashage d'évènements** qui permet d'avoir une liste d'évènements pour chaque robot des données de simulation. Les différentes méthodes de cette classe sont entre autres :

— **ajouteEvenement(...)** qui permet d'ajouter un évènement dans la bonne liste d'évènements dans la table de hashage avec comme clé le robot considéré. Ceci est effectué pour permettre du parallélisme dans l'exécution des évènements de différents robots. En ajoutant un évènement on lui associe directement le "bon" temps auquel il est censé se terminer en prenant en considération le temps du dernier évènement qui vient avant lui dans la liste des évènements du robot considéré.
— **draw(...)** qui permet au simulateur de dessiner les différents graphiques des données simulation (les cases, les robots, les incendies avec leurs spécificités).
— **incrementeDate()** : c'est cette méthode qui permet de déclencher l'exécution des evènements de la liste des évènements qui ont leur date égale à la date de simulation actuelle.

2.10 PlusCourtChemin

Cette classe implémente l'algorithme de Dijkstra. Nous avons choisis Dijkstra parce que c'est une solution fiable et reconnue pour la recherche de plus court chemin. Contrairement a A* les solutions sont optimales.

2.10.1 Sommet

Une classe qui stocke les sommets utilisé par plus court chemin..

2.11 ChefPompier

La classe ChefPompier répond à la question 4 du sujet, l'affectation des tâches des robots. la stratégie implémentée est la numéro 2 "stratégie un peu plus évoluée".

2.12 Interface : Drawable

Permet de spécifier qu'une classe peut avoir une position sur une case (les robots, cases sont Drawable)

3 Expérimentation et remarques

La méthode **chercherEau()** de Robot a une très grande complexité algorithmique, mais cela semble inévitables pour avoir une solutions optimales (c'est à dire trouver l'eau la plus proche possible).

Nous avons également fait le scénarios de test 0 et 1 du sujet, ils fonctionnent.

Évidemment nous avons également testé toutes les cartes données par le sujet et elles fonctionnent également, ce que vous pouvez vérifier par vous même. En figure 2 de l'annexe vous pouvez voir notre logiciel en action avec la carte desertOfDeath-20x20.map.

4 Conclusion

Nous avons réussi à faire fonctionner toutes les cartes ce projet est donc une réussite. Des extensions possibles sont d'implémenter d'autre algorithmes de plus court chemin, ou encore de nouvelles stratégies pour le ChefPompier (celle qui est implémentée actuellement n'est clairement pas optimale).

5 Annexes

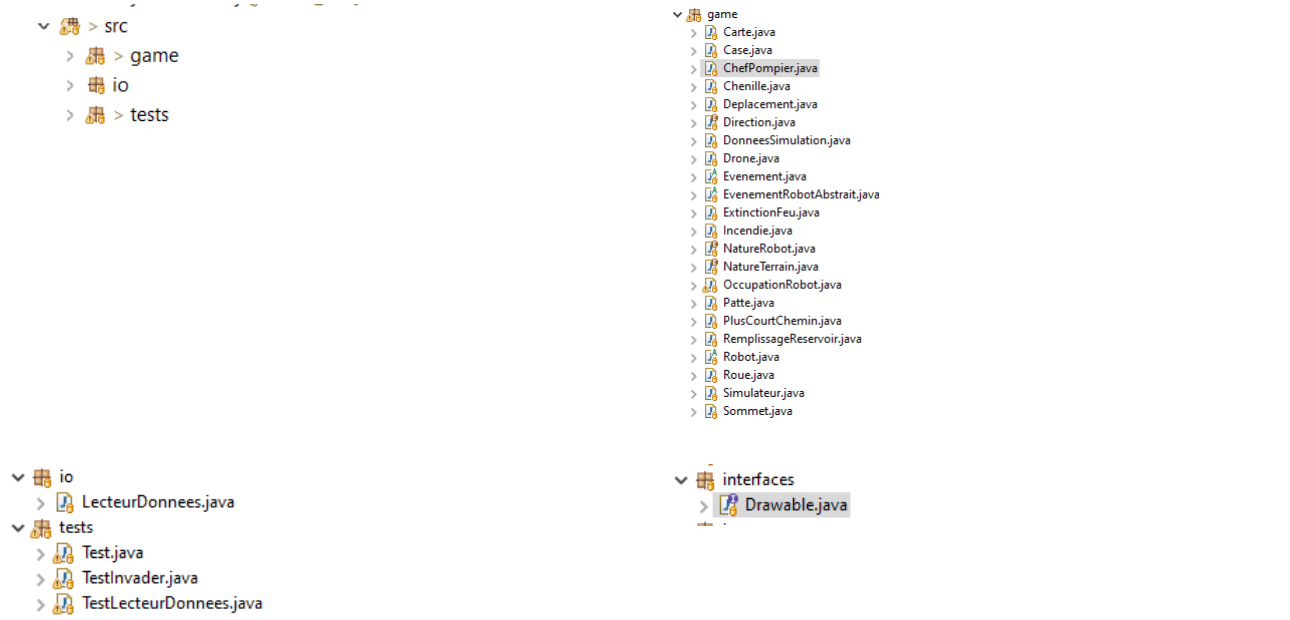

FIGURE 1 – Hiérarchie des classes du projet

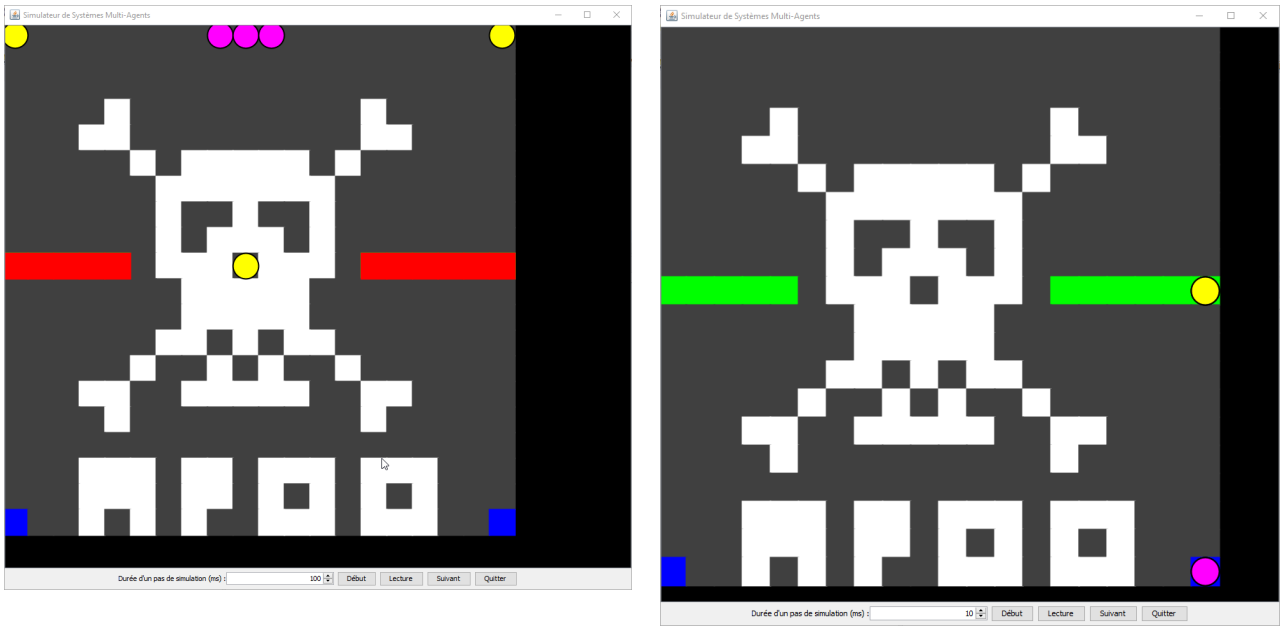

FIGURE 2 – Capture d'écran de l'interface graphique, avant et après l'extinction des feux.