GRENOBLE-INP ENSIMAG

Programmation Orientée Objet

TP en temps Libre

Equipe 36

20-11-2020

Introduction:

Dans le cadre de notre deuxième année du cycle ingénieurs à **Grenoble-INP ENSIMAG**, il nous est proposé un projet de programmation orientée objet qui consiste à implementer une simulation d'une équipe de robots pompiers, ces derniers doivent fonctionner de manière autonome dans un environnement naturel.
Le sujet est constitué de 4 parties principales, avec une difficulté croissante, dans les parties 1 et 2, on implémente les classes de base de notre TP (les robots, la carte, les cases...) et on commence à tester les scénarios de mouvement et fonctionnement des robots dans les cartes
Ensuite , dans la partie 3 on essaie de rendre la simulation plus optimale en attribuant aux robots la possibilité de faire des calculs de plus court chemin afin de rendre leurs déplacements plus rapides et efficaces. Dans la dernière partie, on implémente une stratégie afin de rendre le travail des robots autonome ayant pour objectif éteindre les incendies sur la carte
Dans ce qui suit, on vous proposera notre algorithme pour réaliser la tâche et l'étude de ses classes principales.

1 Partie I

1.1 Classes de base

Dans La première partie, on a commencé par une implémentation simple des classes citées dans le sujet, à savoir la classe **carte**, la classe **Case** et la classe **robot**. Cette dernière est du type abstrait, donc on l'a implémenté avec plus de détails dans ses sous-classes tout en respectant les relations indiquées par le diagramme de classes des données du problème. Ainsi, on met des attributs dans chaque classe qui la relient à l'autre classe de relation (par exemple : la classe **Carte** contient un attribut sous type de liste qui présente les **cases** de cette **carte**) .

Ensuite, on a implémenté les deux classes **CreationDonnees** et **DonnesSimulation**, la première sert à lire un fichier (map) et au lieu d'afficher ses données, on l'utilise pour remplir un objet de la deuxième classe qui sert à regrouper la totalité des données de problème (les **robots** et la **carte**) .

1.2 Interface graphique et simulation

À l'aide de l'interface graphique fournie **Gui jar**, on a implémenté l'interface **Simulable** dans la classe **Simulateur**, en utilisant les méthodes fournies et en reliant les deux classes **Simulateur** et **GUISimulator** .

2 Partie II

Dans cette partie, on est censé mettre les robots en mouvement dans les cartes à l'aide des scénarios définis à l'avance. Pour faire ceci, on a implémenté une classe **Evenement** et on a ajouté des méthodes à la classe déjà définie **Simulateur** afin de traiter les objets de classe **Evenement**.

2.1 La classe Evenement

Cette classe du type abstrait nous permet d'implementer les différentes actions et mouvements dans la carte (déplacement d'un robot, verser d'eau ...). En effet , chaque action est présentée par un objet de classe **Evenement**, et puisque chaque action dépend de l'objet qui l'exécute, on implémente des sous-classes de la classe **Evenement** (**EvenementRobot** et **EvenementIncendie**), chacune correspond à des événements réalisés par un objet des deux classes de base (les robots et les incendies)

2.2 La classe Simulateur

Afin de traiter les objets de classe **Evenement** (les actions à exécuter), on a ajouté quelques méthodes à la classe **Simulateur** définie auparavant dans la partie I. Les evenements à réaliser sont stocker dans un dictionnaire, les clés étant les dates de réalisations, et les valeurs étant les listes des évenements à réaliser. En plus, elle contine un nouveau constructeur qui a parmi ces paramètres cette liste des événements. Parmi les méthodes ajoutées on cite :
executionPas: Cette méthode prend comme paramètre une date t qu'on convertit à un entier avec la fonction **Math.floor** (cette conversion est nécessaire pour discrétiser les événements). Ensuite, elle cherche cette date parmi les clés de dictionnaire **evenementsdates** qui est sous forme (date, événement). Enfin elle exécute tous les événements qui sont programmés à cette date t.

3 Partie III

Dans cette partie, mettre les robots en mouvement n'est plus suffisant, on cherche à le faire d'une manière optimale en calculant les (meilleurs) itinéraires pour qu'un robot se déplace dans la carte, autrement dit, en calculant le plus court chemin permettant à un robot de se rendre à une case X .

Pour faire ceci, on a présenté notre carte par un graphe ayant comme sommets les cases et on a implémenté les deux classes :

3.1 La classe Sommet

Cette classe nous aidera à implementer la classe suivante **Graphe** et a comme attributs : un objet "maCase" de la classe **Case** qui nous indique la case présentée par le sommet, une liste "pccs" du type **Sommet** qui contient tous les sommets appartenant au plus court chemin depuis la source de graphe vers la case présentée par le sommet, "distance" qui nous indique la longueur de ce chemin et enfin un dictionnaire "sommetsAdjacents" qui contient les sommets adjacents et les distance pour se rendre à ces sommets. Les méthodes implémentées dans cette classe sont détaillées dans la documentation du projet.

3.2 La classe Graphe

Cette classe nous permet de modéliser la carte sous forme d'un graphe. Après avoir modélisé la carte, présenté les cases par des sommets et les déplacements possibles entre cases adjacentes par des arcs, on utilise la vitesse d'un robot pour calculer le coût de chaque déplacement depuis une case source vers la case destination, après on applique l'algorithme de Djikstra qui nous retourne le chemin liant les deux cases et ayant le coût minimal. La description de chaque méthode utilisée pour modéliser la carte et calculer le plus court chemin est dans la documentation du projet .

4 Partie IV

Dans cette partie, on est censé rendre l'exécution des tâches de la simulation totalement autonome et plus efficace, ainsi les robots doivent fonctionner à l'aide des décisions prises par "le chef pompier", ce dernier est un robot qui possède une vue intégrale de la carte . Pour faire ceci, on implémente la classe **Strategie** du type abstrait, ensuite on commence par une implementation de la sous-classe **StrategieElementaire** qui décrit une simple stratégie qui organise la distribution des tâches entre les robots, enfin on essaie d'optimiser cette stratégie en implémentant une nouvelle sous-classe **StrategieEvoluee** .

4.1 La classe StrategieElementaire

Cette classe représente un type rudimentaire du "chef pompier" qui envoie les robots vers les incendies et organise le déroulement de la simulation en suivant les instructions citées dans le sujet . Parmi ses méthodes, on cite : **executionstrategie**, cette méthode est une implémentation simple des 4 étapes citées dans le sujet .

4.2 La classe StrategieEvoluee

Cette classe représente un "Chef pompier" intelligent qui donne l'ordre à chaque robot non occupé d'intervenir dans l'incendie le plus proche qui n'a toujours pas été affecté. Les robots deviennent autonomes après, cherchant chacun à se remplir dans la case la plus proche qui le permette et répétant le cycle après.

5 Conclusion

Pour résumer, Notre simulation était simple et non autonome dans les deux premières parties. Ainsi on définissait des scénarios pour déplacer les robots et éteindre les incendies .Ensuite dans la partie III on calcule le plus court chemin entre deux cases afin d'optimiser les déplacements des robots sur la carte. Enfin dans la partie IV la simulation devient autonome et les robots fonctionnent sans aucune intervention à l'aide des décisions prises par le "chef pompier".