Rapport TPL POO

Equipe n°20

Partie 1

Les classes implémentées sont celles décrites dans le sujet:

Classes:

Carte, Case, Incendie, DonneesSimulation, Robot, Drone, RobotAChenilles, RobotARoues, RobotAPattes, Simulateur

Enumeration:

Direction, NatureTerrain

Partie 2

Classes implémentées:

Evenement, EvenementRobot, DeplacerRobot, RemplirEau, IntervenirRobot

Organisation des événements:

Le simulateur stocke les événements dans une priorityqueue(minHeap) en comparant leur date d'exécution grâce à l'implémentation de l'interface comparable. Une priorityqueue a été utilisée car on cherche à exécuter les événements qui ont une date inférieure à la date courante d'exécution en commençant par l'événement le plus faible. En prenant ainsi le premier événement avec la priorityqueue, la complexité est en O(1).
Une simulation se termine lorsqu'il n'y a plus d'évènements dans la queue.

Exécution des événements:

La fonction next sera appelée par l'utilisateur, quand la simulation n'a pas encore terminé la fonction incrementeDate est appelée qui incrémente la date du simulateur et appelle la fonction runEvents() qui exécute tous les événements de la priorityqueue qui ont une date plus faible que la date de la simulation.

Événement robot:

Les événements exécutés par les robots héritent de la classe abstraite EvenementRobot qui hérite elle-même de la classe Evenement.

On a ajouté à la classe robot des attributs relatifs à l'événement en cours d'exécution par le robot:
-tempsEvenement: Temps total de l'événement en exécution par le robot.
-tempsEvenementCase: Temps passé par le robot dans la case d'exécution.
-evenementExecution: Correspond à l'évènement qui est en cours d'exécution.

La fonction addTimeAndCallNewEvent est appelée quand le robot exécute un événement et n'a pas encore terminé l'exécution. Elle incrémente le temps d'exécution de l'événement et le réinsère dans la Priorityqueue.

Remplissage du robot:

On ne remplit un robot que si sa position par rapport au point d'eau satisfait les conditions de remplissage. Son réservoir est rempli de façon immédiate est totale à la fin de la durée de remplissage propre à son type.

Superposition des événements:

On a fait le choix de ne pas pouvoir interrompre la réalisation d'un événement en cours.
Si un événement veut réaliser une action sur un robot occupé alors l'appel à IncrementAllOthers décalera dans le temps les événements relatifs à ce robot (mise à part celui en exécution) de la durée de l'événement en cours.

Partie 3:

Classes implémentées:
Sommet, Graphe, Chemin, Dijkstra
Interfaces:
PlusCourtChemin

Interface PlusCourtChemin:

Pour calculer les chemins empruntés par les robots nous avons implémenté une interface PlusCourtChemin. Les algorithmes qui l'implémentent doivent pouvoir retourner un chemin à partir d'une case de départ et une case d'arrivée. Chaque robot créé se voit attribué à l'initialisation un algorithme de type PlusCourtChemin. On respecte ainsi le patron de conception stratégie: on peut dynamiquement changer les algorithmes de PlusCourtChemin des robots. Dans notre simulation, tous les robots utilisent l'algorithme de Dijkstra.

Dijkstra:

Notre implémentation de Dijkstra s'appuie sur un graphe pondéré des temps de parcours. Afin de faciliter son utilisation et ne se préoccuper de rien, l'algorithme est instancié avec un graphe et un robot. Seul le type du robot a de l'importance car il définit la pondération du graphe. L'algorithme gère aussi la mémorisation des exploration précédentes des graphes, là encore dans un but de praticité d'utilisation.

Fonctionnement:
L'algorithme fait une deep copie du graphe initial et en parcourt les sommets en fixant pour chaque sommet la distance minimale à la case d'arrivée et le voisin suivant par lequel passer. On obtient ainsi un graphe "exploré" à partir duquel on construit le chemin à partir de la case de départ. On explore en premier les sommets les plus proches. Pour ce faire, on parcourt le set de sommets à explorer pour prendre celui de distance minimale. On peut donc penser qu'un treeset aurait été plus avantageux que notre hashset mais nous aurions dû retirer et réinsérer les sommets à chaque mise à jour de leur distance ce qui au final aurait été plus coûteux.

Mémorisation:
On stocke les graphes "explorés" dans un attribut hashmap de la classe pour ne pas refaire 2 fois les mêmes calculs. Les graphes y sont identifiés par leur case d'arrivée. Avant de calculer un nouveau chemin, on regarde donc si la HashMap contient un graphe dont la case d'arrivée correspond à la

case d'arrivée ou de départ recherchée. Si c'est le cas alors on peut directement reconstruire le chemin (et l'inverser si c'est la case de départ recherché qui correspond à la case d'arrivée du graphe déjà calculé).

Partage des PlusCourtChemin entre les robots:
Pour éviter de faire plusieurs fois les mêmes calculs, nous instancions pour chaque type de robot de la simulation un algorithme de PlusCourtChemin que nous stockons dans une hashMap de la simulation. On attribue ensuite à chaque robot d'un même type le même objet PlusCourtChemin stocké dans la hashMap. Ainsi, un robot peut "réutiliser" les graphes déjà calculés par d'autres robots lorsqu'il fait appel à son algorithme de plus court chemin.

Avantages et inconvénients:
Avec cette méthode on ne fait jamais 2 fois les mêmes calculs pour l'exploration d'un graphe. La contrepartie est l'espace de stockage car on stocke plusieurs graphes pour chaque type de robots.

Ce choix nous a paru pertinent avant tout car les incendies et les points d'eau sont fixes et mais aussi car ils sont peu nombreux. En effet, si il y avait beaucoup d'eau il n'y aurait pas d'incendie et s'il y avait beaucoup d'incendies alors d'autres stratégies seraient plus intéressantes (comme chercher l'incendie le plus proche tant que le robot a de l'eau).

Nous avons pensé à implémenter une seconde stratégie (un algorithme glouton) qui consiste à explorer le graphe en étudiant à chaque fois le sommet le plus rapidement accessible parmi les voisins des sommets déjà explorés. Nous pourrions arrêter l'exploration dès que le sommet est trouvé ou bien dès qu'un point d'eau est trouvé si c'est ce que l'on recherche. L'exploration serait ainsi plus rapide et le chemin serait une bonne approximation du plus court chemin. Grâce au patron de conception stratégie, il serait rapide d'implémenter un tel algorithme.

Une possibilité pour améliorer l'utilisation de la mémoire serait de vider les hashmap après chaque élaboration de stratégie. En effet, c'est à ce moment là que les calculs sont très redondant et une fois que l'incendie a été éteint alors il est peu probable qu'un robot ait à retourner sur les lieux).

Problèmes rencontrés:

Le plus gros problème que j'ai rencontré dans la partie 3 est le suivant : Je souhaitais qu'un robot ait des instances et méthodes de classe mais il est impossible en java de déclarer dans une classe abstraite des attributs ou méthodes static que devront implanter les classes filles.

J'ai donc donné comme attribut une instance de robot à certains objets (comme Dijkstra ou graphe) pour qu'ils puissent appeler les méthodes et attributs du robot que j'ai forcé à définir dans les sous classe. Sans cette impossibilité, j'aurais simplement donné comme attribut la Class de robot car les ce qui nous intéresse ce sont les méthodes relatives au type du robot mais pas à un robot en particulier.

Partie 4:

Classes implémentées:
ChefPompier, LancerStrategie, ChefPompierElementaire, ChefPompierElementaire2, ChefPompierAdvanced,

ChefPompier:

ChefPompier est une classe abstraite qui est héritée par tous les chefs pompiers. Cette classe contient une méthode abstraite executeStrategie(), fonction principale d'un chef pompier. Cette méthode donne des ordres aux robots suivant leur état (occupé ou non).

On garde en mémoire une liste des cases d'eau et une liste des cases voisines à des cases d'eau. Pour connaître l'état des robots, on utilise aussi une map des robots avec le temps restant pour exécuter les évènements qu'ils sont en train d'exécuter.

LancerStrategie:

La classe LancerStrategie est un événement dont son constructeur prend comme paramètre un chefpompier qui correspond à la stratégie qu'on va exécuter. Cet événement sera appelé à chaque date d'exécution quand il existe encore des feux.

ChefPompierElementaire 1 & 2:

La classe ChefPompierElementaire commence par donner une capacité infinie aux réservoirs de tous les robots. La stratégie consiste simplement à itérer parmi les incendies pour envoyer à chacun d'entre eux le robot le plus proche.
ChefPompierElementaire2 implémente la stratégie inverse: elle envoie chaque robot à l'incendie le plus proche.

ChefPompierAdvanced:

ChefPompierAdvanced envoie elle aussi le robot le plus proche à chaque incendie (comme ChefPompier) mais ne considère pas la capacité des réservoirs infinie.
Dès qu'un robot à son réservoir vide, il cherche la case la plus proche dans laquelle il peut se remplir.

Test réalisées:

3 stratégies ont été réalisées avec un test pour chaque map. Les stratégies fonctionnent comme attendu. Il est possible de compléter le projet en implémentant de nouvelles stratégies ou algorithme de plus court chemin.

Exécution du code:

Mise en place:
Décompresser l'archive dans un dossier puis lancer make all depuis ce dossier.
Exécution:
Pour lancer une simulation, utilisez la commande "make <nom_test>". Les noms des tests peuvent être trouvés dans le MakeFile. Ils sont de la forme:
"exeTestStrategie<Elementaire1 ou Elementaire2 ou Advanced>Map<1 ou 2 ou 3 ou 4>"
Par exemple, pour tester la stratégie 1 sur la carte 1, il suffit de faire après le "make all": "make exeTestStrategieElementaire1Map1" .
Pour générer la documentation, il suffit de lancer la commande qui est dans le fichier ReadMe.