ROBOT POMPIER
Projet de programmation orientée objet
par
Léonard Mommeja
Jérémie O'Brien
Raphäel Prêtre-Heckenroth

Résumé de l'avancée du projet :

N'importe quelle carte peut être lue, et tous les incendies seront alors éteints. Une stratégie avancée est implémentée avec un chef des pompiers qui envoie vers chaque incendie, le robot le plus proche. Grâce à l'algorithme de Dijkstra, celui prend le chemin le plus court pour aller à sa destination et intervient au volume nécessaire. Une fois vide, le robot va remplir son réservoir au point d'eau le plus proche. Toutes les cartes fournies marchent à condition d'accélérer la simulation graphique au plus rapide. Chaque action est à la même échelle de temps et remplir les réservoirs prend beaucoup de temps comparé au déplacement et à l'intervention.

Le projet se lance à l'endroit du Makefile. Il faut faire make puis lancer la carte voulue avec make exesujet, make exemushroom, make exedesert, make spiral.

La première partie du projet fut la création des classes des Robots et de l'interface graphique ainsi que de la carte. Pour un maximum de clarté, ces classes et les éventuelles classes qui en héritent ou les énumérations liées sont mises dans des packages différents.

Lecture et stockage des données

L'algorithme effectue un parsing de la carte source et stocke les valeurs lues dans les différentes classes. De plus, les valeurs par défaut (notamment le débit et le volume des robots lus) sont également stockées.

Interface graphique

L'interface graphique s'affiche en lisant la carte donnée en argument et en associant chaque objet de la carte à un symbole graphique et une couleur.

- Incendie : petit carré orange (nombre de litres nécessaires à l'extinction du feu écrit en bleu)
- Robot : petit carré rouge (nombre de litres dans le réservoir écrit en blanc)
- EAU : bleu
- HABITAT : beige
- FORET : vert
- ROCHE : gris
- TERRAIN_LIBRE : marron

Nota bene : lorsque deux robots sont sur la même case, ils sont confondus.

L'affichage graphique correspond simplement à une lecture des données de Carte. La difficulté de l'interface graphique réside plutôt dans l'implémentation des méthodes next() et restart().

next() : Si la simulation n'est pas terminée, on incrémente la date *d* du pas *p* choisi. On exécute ensuite tous les événements dont la date de fin ne dépasse pas la date *d+p*
restart() : On relit le fichier .map passé en argument de manière à avoir de nouveau une carte identique à la carte initiale. Cependant, il faut également faire en sorte que les robots de l'ancienne carte (avant le restart) ne soient plus actifs car ce sont eux qui sont pointés dans la liste des évènements à réaliser. Pour cela, on redirige les robots désignés dans les événements vers les robots créés lors de la relecture de la carte ensuite. Ensuite, on déplace simplement le pointeur de l'événement courant vers le premier événement de la suite d'événements pour reprendre au début de la simulation.

La Carte :

Pour ce qui est de la classe Carte, nous avons privilégié un constructeur nécessitant uniquement les dimensions de la carte afin de la créer. En effet, les informations telles que les incendies, le nombre d'incendie et les robots seront ajoutés ensuite vu que ces informations sont données à la fin du fichier .map. Les cases ont leur propre classe contenant leur position, la nature du terrain et de l'incendie. De plus, un HashMap<Direction,Case> permet d'enregistrer tous les voisins de la case à l'intérieur de la case. Ce choix a eu lieu très tôt et a permis de grandement faciliter l'écriture des méthodes de mouvement vu qu'il était très facile de détecter si le robot pouvait bouger ou pas. Plus tard, cela a aussi permis de faciliter le remplissage en eau du robot en cherchant une case contenant un voisin avec de l'eau.

Le Robot :

Dans un effort de factoriser le code, nous avons pris avantage au maximum du caractère abstrait de la classe Robot. Ainsi par exemple la methode move(Direction Dir) est définie dans la classe abstraite, en revanche, la méthode canMove va être abstract donc propre à chaque classe car elle va dépendre de la nature du terrain de la destination. En effet dès lors que les robots ont des caractéristiques différentes, il vaut alors mieux définir des abstract methods dans Robot puis les définir dans les classes héritées. Par ailleurs, à l'heure où la programmation agile est de mise, il vaut mieux avoir le code le plus modulable possible. Ici il est facile de rajouter de rajouter un nouveau type de robot ou une nouvelle action.

Pour gérer les événements, le Robot dispose aussi d'un champ dateWhenFree. C'est la date à partir de laquelle le robot est disponible. Ainsi un robot est libre si dateWhenFree < DateSimulation.

Le Simulateur :

Nous avons décidé de stocker les Evenements dans une ArrayList où nous accédons à chaque Evenement par l'indice rangEvent. Tout d'abord nous avions choisi une ArrayDeque pour ne pas avoir d'indice et une FIFO paraissait simple pour gérer les évenements. Cependant, nous nous sommes rendus compte que ce n'était pas adapté à la méthode restart car une fois le simulateur vidé de ses évenements (spécialement dans le cas sans chef des pompiers), il était impossible de les récupérer. Avec une ArrayList, il suffit de remettre à 0 le rangEvent et la date.

Nota bene : La classe s'appelle SimulateurBis dans notre code, dû au changement de stratégie.

Les Evenements:

Nous avons défini trois événements différents à partir d'une classe abstraite Evenement : Deplacement, Intervention, Remplissage. Chacun nécessite le Robot et Deplacement nécessite aussi une direction. Selon les besoins, chaque Evenement est ajouté dans l'ordre à l'ArrayList du Simulateur. Chaque Evenement a par ailleurs une date et une durée pour pouvoir mettre à jour la dateWhenFree de Robot.

Nota bene : Lors de l'écriture du simulateur, on ajoute les événements sans nécessairement les exécuter instantanément. Notamment, si on effectue un mouvement de A vers B, puis un calcul de plus court chemin vers C, alors l'algorithme de base effectuerait le calcul de plus court chemin de A vers C. Pour pallier ce problème, nous avons donc créé un champ *PositionSimulee* qui est mise à jour à chaque ajout d'événement de type Deplacement et qui correspond donc à la position future du robot, à l'instant où le déplacement sera terminé. On peut alors effectuer le calcul de plus court chemin sur la *PositionSimulee* pour obtenir un résultat correct.

Le CheminOptimal:

Ainsi, l'objet CheminOptimal est défini pour **un robot donné**, et est valable à une position donnée. Nous avons accès à la carte et au simulateur à travers le robot.

Nous avons 2 méthodes publiques : **getShortestTime** et **travelTo**. getShortestTime renvoie le temps nécessaire pour atteindre une case cible, et travelTo envoie les instructions au simulateur nécessaires pour déplacer le robot, le plus rapidement possible, à une case cible. Ces deux méthodes prennent en argument **une case cible donnée**.

Nous avons fait le choix d'utiliser l**'algorithme de Dijkstra**, vu en cours de Recherche Opérationnelle, car la situation peut être modélisée par un graphe pondéré positivement, et on recherche un plus court chemin entre deux sommets.

Pour ce faire, nous avons créé 2 tableaux : un tableau de long pour répertorier le temps minimal pour atteindre chaque case, et un tableau de directions pour répertorier quelle direction a servi pour atteindre chaque case, dans le cas d'un plus court chemin.

La première fois qu'une méthode est utilisée, on calcule les valeurs adéquates pour les 2 tableaux en explorant toutes les cases avec l'algorithme de Dijkstra. Or, pour un même robot dans une même position, il n'est pas nécessaire de recalculer ces tableaux. Donc pour tout appel de méthode par la suite, **on utilise les tableaux déjà remplis**. On sait qu'il ne faut pas refaire le calcul, car la pile des cases à explorer par l'algorithme de Dijkstra est vide.

getShortestTime est un simple accès au tableau des temps minimum.

travelTo fait appel à une pile pour mémoriser les directions prises pour atteindre la case cible. Comme on récupère ces directions en retraçant le chemin inverse à prendre, la structure de pile est parfaitement adaptée. En effet, cette structure nous permet de seulement avoir accès au dernier élément ajouté, or c'est le seul élément dont on a besoin pour ensuite envoyer une-à-une les directions au simulateur.

Plus généralement, on préfère utiliser des piles que des listes car c'est une structure plus efficace si on a seulement besoin d'accéder au dernier élément. Mais l'utilisation de

tableaux est adaptée si on connaît ses dimensions à l'avance et qu'on a besoin d'accéder à plusieurs de ses éléments. Nous avons donc tiré parti de ces deux structures de données pour CheminOptimal.

Les Tests:

Pour les tests, nous avons appliqué le paradigme du Test Driven Development (TDD) c'est à dire que nous rédigeons nos tests avant de rédiger le code. Ainsi, pour tester le simulateur, nous avions juste écrit le squelette des classes de Simulateur et Evenement afin de pouvoir rédiger les tests sans erreur puis nous avons ajouté quelques événements au simulateur et notre but était de réussir ce test. La plupart des tests sont obsolètes et ne sont pas joints au Makefile mais sont dans l'archive.

Dans la version finale du robot pompier, toutes les cartes fournies par le projet sont résolues avec tous les incendies éteints en un temps court à condition de régler l'interface graphique pour avancer rapidement (un pas toutes les ms). Chaque action est à la même échelle de temps et remplir les réservoirs prend beaucoup de temps comparé au déplacement et à l'intervention.

La classe main est src/test/TestChefPompier.java

Optimisations supplémentaires possibles :

Il y a encore beaucoup d'optimisations susceptibles de réduire le temps d'extinction d'un feu. Par exemple :

- un robot qui a presque vidé son réservoir va aller le vider sur un incendie même si celui-ci est éloigné et qu'il ferait mieux de remplir son réservoir avant.
- l'ordre dans lequel les incendies sont considérés est encore arbitraire.

Le choix de l'incendie à éteindre en premier peut lui-même faire l'objet d'une optimisation poussée.