Projet de Programmation Orientée Objet

Groupe 40

Table des matières

Intro

La problématique imposée par ce TP était de concevoir en java une interface graphique montrant des robots en train d'évoluer pour éteindre des incendies de façon autonome. Nous avons commencé par mettre en place l'affichage et le déplacement des robots, avant de réfléchir à des stratégies leur permettant d'éteindre les incendies.

Note : Sur les très grandes cartes, le calcul des chemins est un peu lent et il vaut mieux attendre un peu avant de cliquer sur "début".

1 Les données du problème

Pour modéliser correctement le problème, nous avons décidé de créer plusieurs classes d'objets, qui reprennent et traitent les données du sujet : les cases, la carte qui les contient, les robots et les incendies sont ainsi définis dans des classes à part, qui sont toutes regroupées dans une classe `DonneesSimulation`, qui sera la classe qui contient toutes les données propres au problème. Plusieurs classes d'énumération, telles que `Direction`, `NatureTerrain` ou `TypeRobot` NatureTerrain ou TypeRobot sont alors créées pour faciliter l'interaction entre les différentes classes.

Nous avons choisi d'écrire une classe `Simulateur`, en utilisant `GUISimulateur`. Cette classe définit concrètement les méthodes de l'interface et est caractérisée

par des données de simulation, le pas de la simulation et une liste chaînée d'événements. Lors de l'initialisation de la simulation, on place les robots aux bonnes positions et on remplit leurs réservoirs. La carte (avec toutes ses cases), les robots et les incendies ont au préalable été crées dans `LecteurDonnees`, que nous avons modifié, à partir des données de `DonneesSimulation`.

Le simulateur contient également des méthodes permettant d'ajouter, d'exécuter et d'afficher des événements, et une fonction renvoyant un booléen qui nous permettra de déterminer si la simulation est terminée, et de fermer la fenêtre le cas échéant.

Pour que les données contenues dans les fichiers .map soient utilisables, nous avons modifié la classe `LecteurDonnees` pour qu'elle retourne un objet de type `DonneesSimulation` contenant les données du problème. Pour faire cela, toutes les données sont créées en fonction de leur classe au cours d'une unique lecture du fichier .map, et stockées.

Au départ, avant d'écrire les événements et de prendre en compte la notion de temps, l'affichage se fait dans le constructeur du simulateur, par un appel sa méthode `draw` qui dessine les cases, les incendies et les robots en faisant appel des fonctions de dessin qui sont des mthodes propres chaque classe (on cre en fait des objets de type `GraphicalElement` en important des images). Plus tard dans le projet, nous avons implémenté les fonctions `next` et `restart` qui comportent des appels à la fonction `draw` chaque fois que cest nécessaire.

2 Les simulations et les scénarios

La simulation repose entièrement sur une incrémentation du temps par la méthode `next`. Afin de bien mettre celle-ci en place, une file d'événements eéée dans la simulation, chacun disposant dune date d'exécution, à laquelle la fonction `executeEvenement` (appelée à chaque itération de `next`) appelle les méthodes dexécution des différents événements, permettant ainsi un mouvement dans le temps. Une méthode `finSimulation` s'assure quil y a encore des événements à exécuter et coupe la simulation lorsque celle-ci est terminée.

Il est également possible de faire revenir la simulation à t = 0s : pour cela, une fonction `restart` a été implémentée. Plusieurs choix ont été faits pour que cette fonction marche correctement : ainsi, le fichier .map appelé pour créer la première fois le fichier est stocké dans `DonneesSimulation`, et relu à l'aide de `LecteurDonnees` à chaque appel de `restart`. Cela nous assure que les données sont bien dans leur état initial. Afin d'éviter de refaire les calculs d'événements, les événements contenus dans la liste dans `Simulateur` ne sont ni supprimés ni recréés : ils sont créés une unique fois à la création de la simulation. Afin de ne pas les rendre caduques, nous avons donc choisi de ne pas changer les robots et incendies, mais de simplement réinitialiser leurs états (position, intensité etc).

Nous nous sommes rendu compte, notamment via les différentes cartes proposées que le pas du simulateur devait changer suivant la taille des cases de la carte. En effet, un pas trop petit peut rendre la simulation très lente sur une grande carte, et un pas trop grand peut donner limpression que les robots se téléportent sur une petite carte. Nous avons donc essayé de trouver une relation intéressante entre le pas et la taille des cases.

Pour tester, nous avons créé des tests qui reprennent les scénarios de l'énoncé. Nous avions déjà prévu des levées d'exceptions pour les déplacements, le remplissage et le vidage du réservoir. Le test nous a confirmé leur bon fonctionnement. Une fois les scénarios validés, nous avons commencé la 3ème partie.

3 Calcul des plus courts chemins

Nous avons créé une classe `Chemin`, dont les attributs sont notamment un point de départ, un point d'arrivée, un robot et une liste de cases constituant le chemin. Le constructeur appelle une fonction `calculer`, qui applique l'algorithme de Dijkstra pour trouver le chemin le plus court. Cette fonction crée un dictionnaire qui donne la distance nécessaire pour arriver à chaque case depuis la case de départ. On fait une premire itération depuis la case de départ, puis on itère en recherchant des chemins plus courts (en temps) passant par les cases voisines, jusqu'à ce que le dictionnaire ne soit plus modifié. Il suffit alors de récupérer le chemin correspondant, qui a été stocké dans un autre dictionnaire. Une variable "possible" est alors créée pour vérifier rapidement qu'il existe bien un chemin allant à la case voulue.

Afin de ne pas avoir de chemin faisant passer les robots sur des cases impossibles, et pour diminuer le temps de calcul, seules les cases accessibles sont effectivement parcourues - grâce une fonction `cheminPossible` qui renvoie false si le robot ne peut pas aller sur la case. De plus, nous n'ajoutons pas les cases si le chemin pour y aller est plus long que le chemin trouvé précédemment, ce qui nous évite de parcourir toute la carte et réduit donc le nombre d'itérations.

4 Résolution du problème

4.1 Algorithme naif

Nous avons implémenté l'algorithme 4.1 du sujet : on parcourt les incendies non affectés et on prend le premier robot non occupé. Pour cela, nous avons ajouté un attribut à la classe `Robot` qui est `Etat`, pour connaître l'état des robots. Cette variable est mise à `Libre` à la création d'un événement. Lors de l'exécution, on remet l'état à `Libre` seulement si le robot n'a plus d'autre action à effectuer. La fonction `proposer_incendie_naif` appelle une fonction

`traitement_incendie` qui crée les événements de manière à éteindre l'incendie (il peut ne pas y avoir assez d'eau dans le réservoir donc il faudra faire des aller-retours).

4.2 Algorithme évolué

Pour cet algorithme, seul le choix des robots change. On trouve simplement le robot le plus proche de l'incendie. Nous avons décidé de garder le chemin candidat de manière à ne pas le recalculer.

4.3 Changements engendrés

Nous nous sommes rendu compte du problème de temporalité que le chef pompier créait. En effet, nous avions décidé de calculer le déplacement à l'exécution de l'événement. Nous avions contourné le problème pour le calcul de déplacement avec une variable correspondant à la case précédente. Ici, il fallait qu'on sache où serait le robot à la fin du dernier événement. Il y avait exactement le même problème avec les réservoirs. Nous avons donc dû ajouter des attributs au simulateur pour savoir la position et les capacités des réservoirs des robots à la fin du dernier événement les concernant.

Conclusion

En conclusion, nous arrivons à une résolution fonctionnelle du problème, dans un temps raisonnable et en respectant toutes les contraintes données. Toutefois, plusieurs modifications à notre projet restent possibles pour l'améliorer : l'algorithme de traitement des incendies pourrait être plus efficace en affectant non pas à chaque incendie le robot qui y arrive le plus vite, mais en assignant chaque robot l'incendie auquel il arriverait le plus vite, ce qui éviterait beaucoup de temps de déplacement. Pour aller encore plus loin, il serait possible de ne pas regarder le temps de déplacement, mais le temps total jusqu'à lextinction de lincendie, ce qui accélrerait encore un peu la simulation. Concernant la gestion des événements, il est également possible de ne pas les traiter comme chacun des événements concernant toute la simulation, mais qu'ils soient propres chaque robot, et donc traités dans la classe `Robot` plutôt que dans la classe `Simulation`. Cela aurait pour effet de réduire le temps de parcours de liste qu'on effectue dans `next`.

Ce projet, que nous avons trouvé très ludique, nous a permis de comprendre les bases de la programmation orientée objet, et ses forces, notamment lors de la répartition du travail. De nombreux points évoqués lors du cours, comme l'hérédité ou les classes abstraites, ont pris tout leur sens dans ce projet, où nous avons pu constater toute leur utilité.