Rapport de TPL de Programmation Orientée Objet

Equipe 53

Sommaire :

Introduction :

Avant de commencer la lecture de notre code, nous vous invitons à vous concentrer sur la partie test. Par la suite, dans la deuxième partie nous détaillons un résumé de ce que fait l'ensemble de notre code. Bien sûr, en même temps nous détaillerons de manière efficace et concise les points importants de notre code que nous avons mis en place lors de la partie Analyse du sujet.

I. Tests:

Pour les tests, nous vous invitons à vous placer dans le dossier correspondant à notre sujet et de suivre les indications suivantes :

Pour compiler le test TestSimulation.java :

make Simulation

Pour lancer la simulation sur une carte:

make exeSimulation carteSujet
make exeSimulation desertOfDeath
make exeSimulation mushroomOfHell
make exeSimulation spiralOfMadness

Sur la fenêtre, selon la taille de la carte qui peut être plus ou moins grande, nous vous invitons à choisir les paramètres 100 et 100.

II. Données du problème, analyse et mise en place de la solution

Pour commencer, après avoir analysé la construction des cartes, nous avons décidé de suivre la structure suivante :

Tout d'abord nous avons représenté une carte par une matrice de taille (nbLine x nbCol) de cases de taille caseSize, chaque case est identifiée par ses coordonnées (line x col), associée à la nature du terrain correspondant (type énuméré LandNature) et à un boolean isOnFire qui renvoie true si la case est un incendie, on y associe alors un incendie qui contient donc cette case mais aussi un entier qui décrit le nombre de litres d'eau qui faut pour éteindre le feu.

Pour la mise en place des différents robots, nous avons créé une classe abstraite Robot qui rassemble les attributs communs à tous les robots, aussi bien que des fonctions communes:

Pour effectuer la partie visuelle à l'aide du package gui, nous avons utilisé le fonctionnement suivant :
→ pour les drones, on les représente sur la map, grâce à un ImageElement associé. Comme ça, lorsque le robot bouge, il suffit de translater cette image selon les directions que va prendre le robot : on ne crée donc pas une nouvel ImageElement à chaque déplacement de robot.
→ pour les cases, celles-ci ne bougent pas, on représente donc dès le début les cases en créant un ImageElement associé, puis on les affiche directement.
→ pour les Fire, on les affichent dès le début, lorsqu'une d'entre elles se fait éteindre par un robot, c'est-à-dire quand neededWater est nul, alors grâce à la classe CeaseFire, on enlève la case Fire du dictionnaire associé et de l'affichage.

Nous pouvons maintenant nous concentrer sur la partie Simulation, maintenant que nous pouvons représenter notre map et notre jeu de robot.

Tout d'abord, juste après la lecture du fichier grâce à LecteurDonnees (voir commandes dans le MakeFile), on crée notre fenêtre et tout ce qu'il va s'y passer grâce à GUISimulator mais surtout on récupère grâce à la lecture toutes les données principales du problèmes qui sont les suivantes : la carte (avec ces cases associées, etc..), un dictionnaire avec les cases Fire et une liste avec les robots que l'on possèdent (avec leurs positions, etc..).

On lance ensuite la simulation, la partie importante à bien comprendre est la suivante :
on crée une PriorityQueue d'événements, ce qui va permettre par la suite grâce à une relation d'ordre entre les événements sur leurs dates, où les événements se déroulent les uns après les autres les événements au bons moments, pour afficher tout cela, tout se passera dans la méthode next par la suite. Mais avant cela, concentrons-nous sur la partie principale de notre code.

On donne par la suite à celui qui orchestrera la résolution du problème : notre SillyChief les données du problème, la liste d'événement qu'il va devoir remplir, et une date où la simulation commence pour avoir un repère de temps.
Notre chef pompier crée un dictionnaire d'incendies à éteindre et boucle sur cette dernière jusqu'à ce qu'elle devienne vide, tout en lui affectant un robot grâce à la fonction sendRobot() qui renvoie la date où le robot finit sa tâche (à savoir éteindre le feu).
sendRobot() quant à elle, itère sur chaque robot jusqu'à trouver un robot disponible auquel cas elle l'envoie. Sinon, elle renvoie la valeur 0 qui fait office de failure. Maintenant, si le robot est disponible, on vérifie s'il existe un chemin possible vers l'incendie. Si tel est le cas, grâce au pathconstructor on invoque l'algorithme de Dijkstra pour trouver le plus court chemin vers cet incendie puis ajoute à la liste des événements tous les événements nécessaires afin d'atteindre la case voulue. Ensuite, on calcule le nombre total d'interventions que le robot doit effectuer selon son réservoir et sa capacité d'intervention en volume afin d'éteindre totalement le feu. Ce qui nous permet encore une fois de répéter l'intervention unitaire jusqu'à ce que le feu cesse. On ajoute finalement l'événement CeaseFire dont on a parlé un peu plus haut.
Directement après, on dessine sur notre fenêtre gui la map, les cases en feu et les robots (cet ordre est important pour éviter d'avoir une mauvaise superposition des images).
Juste après, on utilise notre SillyChief pour s'occuper de tous les événements qui vont permettre de manipuler les robots, et lors de l'envoi d'un robot, on rajoute dans notre PriorityQueue son événement correspondant à faire. Pour résumer les événements les plus importants sont les suivants : éteindre un feu, se déplacer à une case correspondante et remplir le réservoir.
Mais, le plus important ici est de comprendre le fonctionnement de comment le robot va se déplacer à une case correspondante. Pour ce faire, nous utilisons l'algorithme de Dijkstra, pour résumer, nous en déduisons une LinkedList de Direction que le robot va devoir suivre, ce qui permettra grâce à goToCase qui dépend de la case source et de la direction de déplacer notre robot. Ainsi l'événement associé va attraper cette LinkedList.

Maintenant que nous avons accès à tous les événements qui vont se dérouler pendant la simulation, qui vont permettre de résoudre le problème comme démontré juste précédemment. Nous allons configurer notre bouton next, ce qui va permettre de gérer l'affichage de la simulation. On crée un itérateur sur notre PriorityQueue de nos événements. On regarde par la suite avec une boucle while, tant que la date d'événements (qui sont triés grâce à la relation d'ordre) est plus petite que la date de simulation qui se trouve modifiée grâce à notre gui(cf l'affichage). A chaque fois que l'événement qui a la date la plus petite sera pioché avec peek dans la boucle while a sa date est plus petite que la dateSimulation on le poll (on l'extrait de la liste d'événement : il n'est plus dedans et on garde toujours la relation d'ordre) puis bien sûr, on "travaille" directement sur l'affichage par exemple avec des translations sur un événement associé à un robot, puis avant de se replacer dans la boucle while, on replace notre "curseur" sur l'événement suivant où l'on va regarder de nouveau les conditions de la boucle while.

Lorsque l'on a fait tous les next nécessaires, c'est-à-dire lorsque la liste d'événements est vide, la simulation est alors terminée!

Le problème qui était déjà résolu, a maintenant son affichage qui fonctionne étape par étape

III. Conclusion :

Nous avons donc répondu à toutes les spécificités demandées par le sujet. Les pompiers font bien leur travail, et tout le monde est sauvé car les feux ne sont plus d'actualité dans le monde féerique de notre affichage graphique.

Remerciements

On vous remercie pour le temps accordé à notre lecture de notre compte-rendu, et pour la lecture du code qui suit.