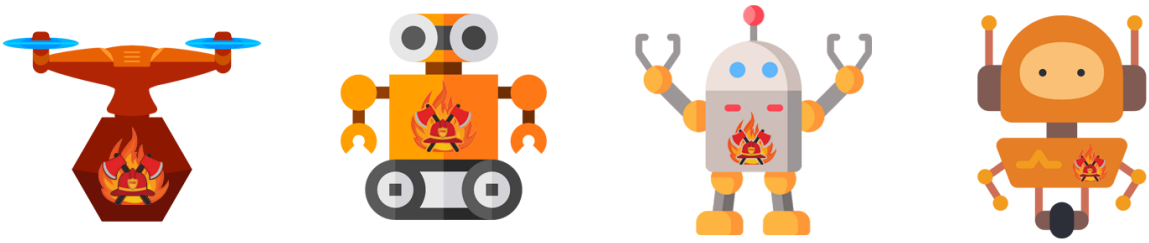

Les Robots Pompiers

Equipe 66

Choix de conception

Dessin de la carte

A chaque appel à next, nous avons décidé de redessiner entièrement la carte. Nous sommes bien au courant que l'on aurait pu utiliser la méthode translate pour effectuer le déplacement des robots sans avoir à tout redessiner, ce qui aurait été moins lourd. Mais cela ne nous aurait pas permis de conserver la barre sous les robots qui indique la quantité d'eau qu'il reste dans leur réservoir, car on ne peut pas modifier la taille d'un Rectangle sans le redessiner (pas de méthode set pour les attributs width et height), nous ne pouvons donc pas mettre à jour la quantité d'eau dans le réservoir des robots si on ne redessine pas tout.

Évènements

Une action (déplacement, remplissage, déversage) possède deux évènements associés :

- Un événement de début, qui va calculer la durée de l'action et ajouter l'événement de fin au simulateur en fonction de ça, mais cet événement n'exécute pas l'action
- Un événement de fin, qui va effectuer plus concrètement l'action (changer les coordonnées du robot, diminuer la quantité d'eau dans le réservoir ou l'augmenter)

Nous n'ajoutons jamais les événements de fin nous-même, seul les événement de début se charge de faire cela.

Le simulateur possède une date représentée par un entier (long), qui est incrémenté de 80 ou de 5 (en fonction de la carte pour que la simulation ne soit pas trop longue) à chaque appel à next.

La séquence d'événements du simulateur est représentée par une liste chaînée d'événements triée par date croissante, c'est pourquoi chaque événement possède un attribut suivant de type Evenement. (On aurait aussi pu utiliser une collection java)

Calcul du plus court chemin

Pour calculer le plus court chemin entre la position d'un robot et sa destination, on a utilisé l'algorithme A* (astar), implémenté dans la classe Gps, dans le package carte.

Cet algorithme a principalement été choisi pour sa simplicité de mise en place et ses performances. Il peut être vu comme une amélioration de l'algorithme de Dijkstra. Permettant de trouver le plus court chemin entre deux cases données, il est souvent utilisé dans le domaine du jeu vidéo.

Ici, nous l'avons choisi car il permet à un robot donné, d'atteindre le plus rapidement possible un incendie donné. Les positions d'arrivée et de départ sont donc connues. Dijkstra aurait permis potentiellement d'obtenir un chemin entre un robot et l'incendie le plus rapidement atteignable. On a préféré utiliser A* à Dijkstra, afin de séparer les tâches et de laisser à la partie stratégie le choix de l'incendie à atteindre.

Chef Pompier

Le chef pompier est implémenté dans le package strategie. Il peut utiliser deux stratégies :

- La stratégie naïve de la section 4.1)
- Une stratégie un peu plus avancée de la section 4.2)

Le chef pompier actualise sa stratégie et donne de nouveaux ordres à chaque appel à next. Il ordonne au robot disponible d'aller remplir son réservoir si son réservoir n'est pas plein et s' il n'est pas capable d'éteindre un des incendies toujours actifs. Sinon il lui ordonne d'aller vider son réservoir sur l'incendie.

Utilisation des classes

Nous vous invitons dans un premier temps à générer la javadoc avec eclipse, qui vous permettra de comprendre le fonctionnement de la plupart des classes. A présent, quelques explications sur le fonctionnement des classes un peu plus complexes à utiliser.

Classe Gps

Cette classe permet, pour un robot et une case donnés, de trouver l'itinéraire (comme le sous-entend le nom de la classe) le plus court pour que le robot arrive à destination. Pour cela, il suffit d'instancier un GPS pour le robot, sa position et la case destination. Puis, il faut exécuter la méthode trouverChemin sur l'objet instancié avec en paramètre le simulateur et les données de la simulation. Finalement, dans le cas où un itinéraire a été trouvé, pour que le robot se déplace suivant le chemin généré on utilise la méthode creationEvenement sur l'objet GPS créé au début.

Compilation & lancement des simulations

Pour compiler le code, lancez simplement la commande make (avec l'archive gui.jar dans le dossier bin). Pour effectuer les simulations, lancez : make exeSimulateur1 pour carteSujet.map, make exeSimulateur2 pour desertOfDeath-20x20.map, make exeSimulateur3 pour mushroomOfHell-20x20.map et make exeSimulateur4 pour spiralOfMadness-50x50.map.

Tests des stratégies

Nous avons chronométré les différentes stratégies sur les différentes cartes, voici les résultats de nos tests.

Stratégie naïve

Carte	Date de fin simulation
carteSujet.map	28560
desertOfDeath-20x20.map	12685
mushroomOfHell-20x20.map	12640
spiralOfMadness-50x50.map	33230

Stratégie évoluée

Carte	Date de fin simulation
carteSujet.map	28560
desertOfDeath-20x20.map	12660
mushroomOfHell-20x20.map	13640
spiralOfMadness-50x50.map	33230

Après des tests, on remarque que la stratégie évoluée n'est pas tant évoluée que ça au final.

Pistes d'améliorations

Meilleure stratégie

Au vu des résultats obtenus pour nos deux stratégies, on peut admettre qu'il serait pertinent d'imaginer une nouvelle stratégie pour notre chef pompier. On pourrait, par exemple, placer plusieurs robots sur un même feu lorsque ce dernier nécessite beaucoup de litres d'eau. Cependant, en ce qu'il concerne l'attribution optimale des incendies, cela va relever d'un problème plus complexe de recherche opérationnel à notre avis.

Trouver les cases d'eau les plus proches

On aurait pu, à la lecture du fichier, stocker toutes les cases d'eau dans une liste, comme on le fait pour les incendies, afin de les trouves plus rapidement en cas de besoin. Par la suite, on aurait calculé les différentes distances pour les éléments de la liste afin d'obtenir le point d'eau le plus près.