Rapport Projet : Simulation d'une équipe de robots pompiers

Equipe n° 16

Introduction :

Le projet sur lequel nous avons travaillé consiste à développer en Java une application permettant de simuler une équipe de robots pompiers évoluant de manière autonome dans un environnement naturel. Pour faciliter l'organisation de notre code et en augmenter sa visibilité, nous avons choisi de répartir nos classes dans des packages dont chacun a une spécificité différente. Dans ce qui ce suit, nous présenterons la description des packages et nous expliquerons nos choix de conception.

Présentation des packages

Package affichage :

Il contient la classe $Affichage$ englobant les méthodes de dessin permettant d'afficher les éléments de la simulation dans la fenêtre graphique.

Package donneesSimulation :

Il contient la classe *DonneesSimulation* qui permet de regrouper les données de la simulation (la carte, la liste des robots et la liste des incendies).

Package io :

Il contient la classe *LecteurDonnees* qui fournit une méthode permettant de construire une instance *DonneesSimulation* à partir d'un fichier texte contenant les éléments de la simulation.

Package geographie :

Il regroupe les classes des élements statiques de la simulation: *Carte* - *Case* - *Incendie*. Elles définissent le domaine sur lequel les robots se trouveront.

Package itineraires :

Il rassemble les classes indispensables pour pouvoir calculer le meilleur itinéraire d'un robot à une case donnée en utilisant l'algorithme de Dijkstra tout en tenant compte des propriétés de chaque type de robot et des caractéristiques du terrain.

Package robot :

Il contient la classe abstraite *Robot* qui regroupe les différentes méthodes d'accès aux propriétés utiles d'un robot. Il s'agit de la classe mère des autres classes du package. Chacune de ces dernières implémente les méthodes de la classe *Robot* d'une façon qui lui est appropriée.

Package scenario :

Il regroupe les classes des événements susceptiples d'avoir lieu lors de l'exécution de la simulation et la classe *Simulateur* qui les traite de manière adéquate en fonction des données et de l'état de la simulation.

Package strategie :

Ce package présente principalement deux stratégies d'affectation de tâches d'un chef pompier: une élémentaire et une plus évoluée.

Justification de choix :

- Pour définir la méthode $restart()$ de l'interface Simulable, nous avions utilisé des méthodes de copie pour pouvoir retrouver les données de la simulation dans leur état initial, ce qui était très couteux en mémoire. Pour remédier à ce problème, nous avons opté pour des méthodes $reset()$ qui permettent de réinitialiser les attributs des instances avec les valeurs initiales sans devoir les stocker une nouvelle fois dans la mémoire.
- Quant au stockage du gestionnaire des événements, nous avons recouru en premier lieu à un $TreeMap$ qui, à une date d'exécution associe une liste d'événements. Or nous nous sommes aperçus que nous n'avions pas besoin de la notion d'ordre qui représente l'intérêt majeur de l'utilisation d'un $TreeMap$. Le $HashMap$ est donc mieux adapté à ce cas. De plus, les opérations dont nous avions besoin étaient Get et $ContainsKey$. Leur coût est en $O(1)$ pour un HashMap et en $O(log\ n)$ pour un TreeMap. Par conséquent, nous avons choisi d'utiliser le HashMap à la place du TreeMap. Dans ce map, à chaque date correspondait une $LinkedList$ d'événéments, ce qui n'était pas optimale car l'opération effectuée sur la liste était simplement Add dont le coût est en $O(1)$ pour une $ArrayList$. Pour cela, nous l'avons changé par une $ArrayList$.
- Pour trouver le plus court chemin à une case donné, nous avons opté pour l'algorithme de recherche en largeur de $Dijkstra$ parce qu'il est performent et plus adapté à ce type de graphe. Sa complexité temporelle est en $O(4(n^2 - 4n + 4) + n^2 log\ n^2) = O(n^2 log\ n)$.
- Pour le stockage du chemin, nous avons eu recours à une $LinkedList$ réalisant l'interface $Deque$ pour pouvoir ajouter des sommets lors de l'exécution de l'algorithme de $Dijkstra$ en début de la liste grâce à l'opération $addFisrt$ avec un coût en $O(1)$.

Commentaire sur la stratégie d'affectation de tâches :

Pour cette partie, nous avons opté pour la solution de calculer toute la résolution et de la stocker sous forme d'évènements puis lancer l'animation en une fois, plutôt que faire une résolution au fur et à mesure que le déplacement des robots est affiché sur l'interface. Ainsi, les calculs ne sont pas réeffectués à chaque restart mais il faut un temps de calculs avant de pouvoir lancer l'animation.

Stratégie élémentaire :

Cette stratégie consiste à affecter chaque incendie à un unique robot libre jusqu'à ce qu'il soit éteint ou que le robot n'ait plus d'eau. L'occupation des robots et l'état d'affectation des incendies sont alors stockés sous forme d'un paramètre booléen présent dans chacune des deux classes respectives. Les évènements étant associés à des dates, on crée un Long date qui sert de repère temporel et qu'on incrémente au fur et à mesure de la résolution. On crée aussi des $TreeMap$ qui, à une date donnée, associe les robots ou les incendies qui ne sont plus occupés ou affectés. Cela permet de repérer si à une date t, tel robot ou tel incendie a fini sa tâche ou n'est plus affecté à un robot pour ensuite l'associer à des évènements.

Stratégie avancée :

La méthode de résolution reste la même mais un incendie peut être affecté à plusieurs robots en même temps et il faut essayer d'occuper tous les robots à chaque instant. Pour ce faire, nous avons remplacé les TreeMap de libération et désaffection par une PriorityQueue permettant de trier la date de libération des robots de leurs précédentes actions plus facilement que sur un TreeMap car on ne traite que le premier élément à chaque fois. Nous avons ajouté également un HashMap pour les affectations Robot-Incendie.

Cette fois-ci, les évènements d'intervention sont beaucoup plus mesurés. En effet, le robot ne déverse plus tout son réservoir mais l'équivalent de 30 secondes de son débit d'eau. Cela permet d'avoir des dates de libération plus proches et l'affectation de plusieurs robots sur un même incendie de façon à minimiser le gâchis d'eau. A la fin de chaque itération, on essaie d'affecter chaque robot encore libre à l'incendie le plus proche et on passe à la date suivante de libération d'un robot.

Description des tests effectués :

Afin de tester notre code, nous avons effectué des tests unitaires pour vérifier le fonctionnement de chaque package. Notre implémentation ayant beaucoup évoluée depuis le début du projet, nous avons regroupé ces différents tests dans un test global. Ce dernier prend un fichier *.map* en paramètre contenant les données sur la carte, les robots et les incendies. On effectue ensuite une affectation des tacˆhes nécessaires pour éteindre l'incendie par un chef pompier.

Les cartes utilisées dans ces tests sont celles fournies avec le sujet du projet, qui possèdent des caractéristiques et dimensions différentes permettant ainsi de tester de manière exhaustive notre code.

Les différentes étapes de compilation et d'éxecution sont précisées dans le $Makefile$.

Conclusion :

Nous avons apprécié le travail d'équipe sur ce sujet. En effet, il nous a permis de revoir et d'approfondir plusieurs notions de la POO traitées dans le cours ainsi que d'utiliser des méthodes plus optimisées.