Simulation d’une équipe de robots pompiers

Robin Lamy - Théo Bekhedda - Arnaud Degouy

Contexte

L’objectif de ce projet est de simuler l'intervention de robots pompiers sur des incendies pour une carte donnée. Nous avons donc implémenté en Java un simulateur qui prend en entrée une certaine configuration, c'est-à-dire une carte composée de cases de différents types, des robots pompiers, ainsi que des incendies déclarés à différents endroits de la carte. Nous avons dû simuler le comportement des robots pompiers à l’égard des incendies, leurs mission étant d’éteindre ces derniers en allant puiser de l’eau.

Choix de conception

Classes

(Voir annexes 1).

Simulateur

Nous avons choisi d’implémenter une classe simulateur pour gérer les évènements discrets. Le simulateur implémente l’interface Simulable et possède un GuiSimulator afin d’afficher sa simulation dans une fenêtre. Il possède également une instance de DonneesSimulation afin de conserver les données relatives à la simulation comme la carte ou la position des robots par exemple. Simulateur possède également une liste d'événements qu’il traite dans l’ordre chronologique.

Lorsque l’on appelle la méthode next(), la simulation va exécuter la prochaine itération de la simulation en demandant à un chef pompier de prendre une décision en fonction de l’état courant, puis en traitant les événements. Chaque appel à la fonction next() incrémente la date courante avec un pas de 1000ms. L’affichage est par la suite mis à jour.

La méthode restart() remet à zéro la simulation, aussi bien au niveau du modèle qu’au niveau de l’affichage.

Robot

Nous avons implémenté une classe abstraite Robot qui regroupent les attributs et les méthodes qui sont communs aux quatres types de robots. Puis quatres sous classes héritées (une par type de robot) viennent préciser leurs attributs et comportements différents vis-à-vis de leur environnement et de leurs interactions avec celui-ci.

Calcul du plus court chemin

Pour se déplacer, le robot doit pouvoir calculer le plus court chemin d'un point à un autre en prenant compte de la géographie de la carte. Pour ceci il nous a fallu considérer une case comme le nœud d'un graphe. Selon les contraintes et la récurrence de calcul du plus court chemin, nous avons décidé d'implémenter l'algorithme de Dijkstra au projet.

L'algorithme de Dijkstra présente la particularité d'obtenir tous les plus courts chemins pour une position de départ, et ce, quel que soit le point d'arrivée. L'intérêt est dû au fait que les calculs de plus court chemin se font de manière successive sur le même point de départ pour trouver le point le plus proche, parmi un ensemble de nœuds, avec le temps de parcours minimal et le chemin pour y accéder. De plus, l'algorithme de Dijkstra est plutôt intéressant niveau temps de calcul que la plupart des algorithmes.

Ainsi, nous avons décomposé l'Algorithme de Dijkstra en 2 parties : le calcul des temps minimaux pour aller sur chaque sommet et le nœud qui doit précéder un nœud pour avoir un chemin de plus court temps de parcours, ainsi que la remonté de ces noeud pour trouver le chemin en question. Les résultats de la partie 1 de l'algorithme sont stockés dans la classe de chaque robot pour pouvoir être réutilisés. Lors d'une demande temps minimum la partie 1 suffit et si un appel pour ce robot avec la même position initiale, le temps de calcul se fait en temps constant. Nous avons choisi d'implémenter les résultats de la partie 2 dans une Map pour stocker toutes les informations à l'aide d'une clé (Un nœud du graphe). Pour une demande de plus court chemin, on peut réutiliser le calcul de la partie 1 si il existe déjà. Ensuite, on reconstitue le plus court chemin, ce qui constitue la partie 2 de l'algorithme.

Malgré le fait que Dijkstra soit compétitif, A* est en général bien plus performant. Cependant, l'algorithme "classique" d'A* ne contient le résultat que pour un seul point d'arrivée alors que nous avons besoin d'avoir les distances aux différents feux et points d'eau pour réaliser nos stratégies. Plus tard nous avons découvert qu'il était possible d'appliquer A* à un ensemble de points (le point choisi sera le plus

proche) avec une heuristique particulière. Il nous aurait fallu cependant plus de temps pour l'implémenter et le comparer à notre algorithme. Le résultat aurait sûrement été en faveur du A* à l'heuristique particulière.

Choix des Collections

La liste des évènements

La liste des événements est stockée dans une PriorityQueue, ainsi, la liste est triée à l'aide de la fonction de comparaison par ordre chronologique des événements. Chaque événement exécuté est alors supprimé de la PriorityQueue.

Dijkstra

Pour stocker, les résultats du Dijkstra, nous utilisons une HashMap pour stocker pour chaque nœud accessible (la clé), le temps pour y aller ainsi que le nœud précédent pour y accéder à un temps minimal.

Nous avons dû implémenter une fonction de hachage pour obtenir les résultats plus facilement en définissant la fonction *hashCode()* pour les Cases de la carte. Nous avons choisi d'utiliser comme fonction :

$$h(c,\ l) = l + 10000c$$

avec c l'indice de la colonne et l l'indice de la ligne de la case
Cette fonction garantit une absence de collisions des valeurs de hachage pour des maps de largeur inférieure à 10000.

Stratégies des robots

Glouton
Le pompier dit glouton suit des règles de décision très simples, chaque robot, si il est disponible et que son réservoir n'est pas vide, est envoyé sur l'incendie accessible et non éteint le plus proche de lui. Si son réservoir est vide, alors il va le remplir au point d'eau le plus proche.

Intelligent
Le pompier dit intelligent dirige les robots ainsi : chaque robot, si il est disponible et que son réservoir n'est pas vide, est envoyé sur l'incendie accessible dont aucun robot ne s'occupe déjà le plus proche. Si tous les incendies sont déjà pris en charge

par un ou plusieurs robots, alors il va à l'incendie accessible le plus proche de lui. Lorsque son réservoir est vide, il va le remplir au point d'eau le plus proche.

Tests

Nous avons essayé de concevoir une simulation la plus robuste possible, ainsi la simulation gère les déplacements qui tentent de sortir de la carte. De plus, les robots savent gérer une case qu'ils ne peuvent pas atteindre et n'essaieront pas de s'y rendre, ils tenteront un autre incendie. Dans le cas où aucun incendie n'est accessible pour eux, alors ils ne font plus rien de la simulation. Pour ce faire, nous avons utilisé le mécanisme des exceptions.

Résultats

Carte\Algo	Pompier Glouton	Pompier Intelligent
carteSujet.map	6h 16min 58s	5h 54min 0s
desertOfDeath	3h 13min 05s	3h 12min 04s
mushroomOfHell	3h 16min 47s	3h 32min 20s
spiralOfMadness	8h 54min 03s	8h 52min 40s

Nous pouvons voir que le Pompier Intelligent est légèrement plus efficace que le Pompier Glouton, mis à part sur la carte mushroomOfHell. Une stratégie qui semble plus simple n'est donc pas nécessairement moins efficace qu'une stratégie "plus avancée". Ainsi, certaines configurations de cartes pourraient mettre en défaut le Pompier Intelligent et rendre plus efficace le Pompier Glouton. Pour déterminer ces configurations, il serait nécessaire de regarder de plus près les comportements de groupe des robots et non pas seulement les comportements individuels.

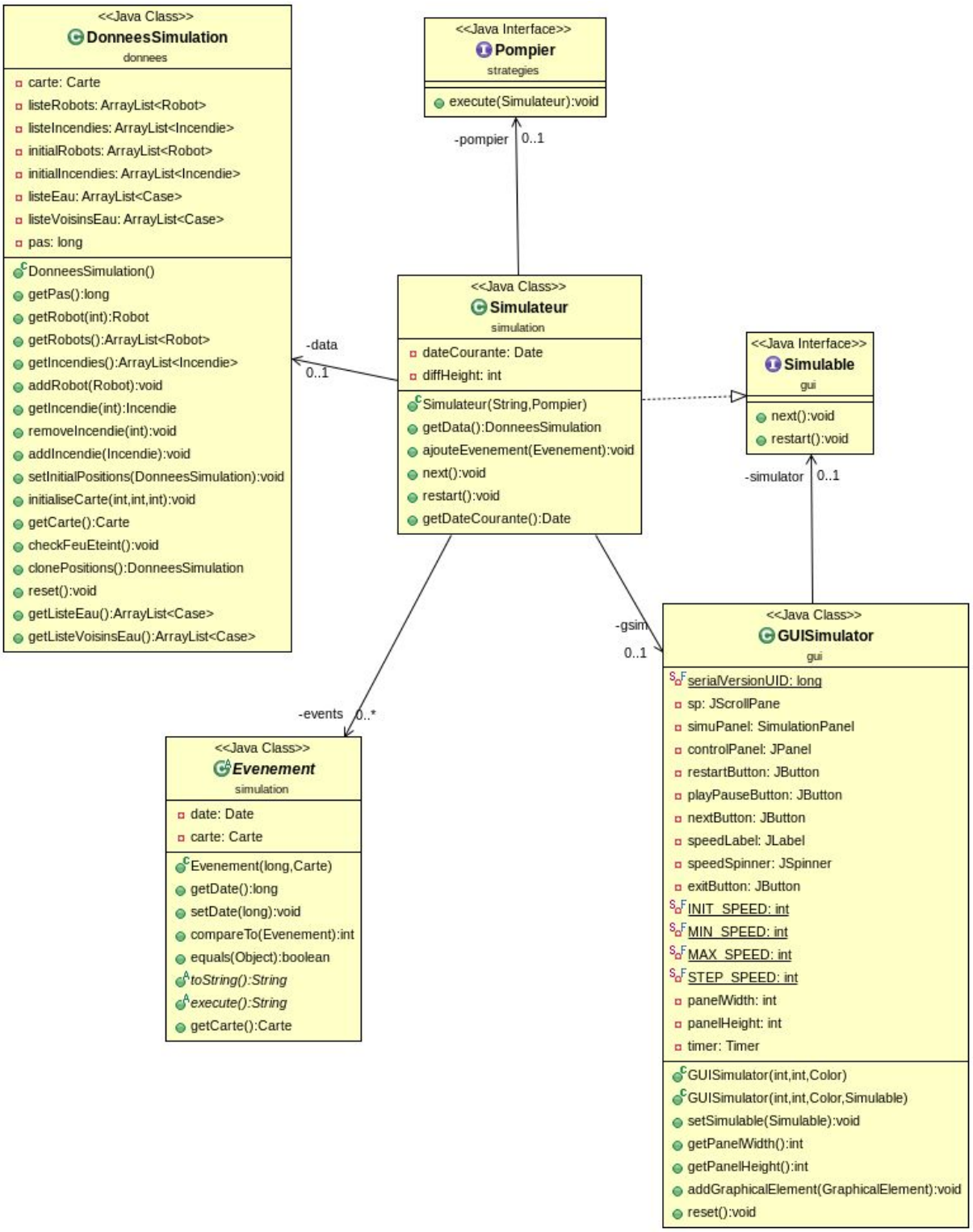
Pour conclure, un pompier encore plus avancé prenant en compte :

- la possibilité de recharger son eau sur le chemin vers un incendie
- le niveau d'eau restant des robots
- la quantité d'eau nécessaire pour éteindre les incendies

pourrait être implémenté comme suggéré dans le sujet partie 4.
Un tel pompier pourrait être mis en place en utilisant un algorithme de Dijkstra ou A* prenant en compte non pas seulement les déplacements mais aussi les actions des robots.

Annexes

Annexe 1 : Diagramme de classes UML incomplet montrant les relations à partir de la classe Simulateur


<<Java Class>>
DonneesSimulation
donnees
carte: Carte
listeRobots: ArrayList<Robot>
listeIncendies: ArrayList<Incendie>
initialRobots: ArrayList<Robot>
initialIncendies: ArrayList<Incendie>
listeEau: ArrayList<Case>
listeVoisinsEau: ArrayList<Case>
pas: long
DonneesSimulation()
getPas():long
getRobot(int):Robot
getRobots():ArrayList<Robot>
getIncendies():ArrayList<Incendie>
addRobot(Robot):void
getIncendie(int):Incendie
removeIncendie(int):void
addIncendie(Incendie):void
setInitialPositions(DonneesSimulation):void
initialiseCarte(int,int,int):void
getCarte():Carte
checkFeuEteint():void
clonePositions():DonneesSimulation
reset():void
getListeEau():ArrayList<Case>
getListeVoisinsEau():ArrayList<Case>
<<Java Interface>>
Pompier
strategies
execute(Simulateur):void
-pompier
0..1
<<Java Class>>
Simulateur
simulation
dateCourante: Date
diffHeight: int
Simulateur(String,Pompier)
getData():DonneesSimulation
ajouteEvenement(Evenement):void
next():void
restart():void
getDateCourante():Date
-data
0..1
<<Java Interface>>
Simulable
gui
next():void
restart():void
-simulator
0..1
-events
0..*
-gsim
0..1
<<Java Class>>
GUISimulator
gui
serialVersionUID: long
sp: JScrollPane
simuPanel: SimulationPanel
controlPanel: JPanel
restartButton: JButton
playPauseButton: JButton
nextButton: JButton
speedLabel: JLabel
speedSpinner: JSpinner
exitButton: JButton
INIT_SPEED: int
MIN_SPEED: int
MAX_SPEED: int
STEP_SPEED: int
panelWidth: int
panelHeight: int
timer: Timer
GUISimulator(int,int,Color)
GUISimulator(int,int,Color,Simulable)
setSimulable(Simulable):void
getPanelWidth():int
getPanelHeight():int
addGraphicalElement(GraphicalElement):void
reset():void
<<Java Class>>
Evenement
simulation
date: Date
carte: Carte
Evenement(long,Carte)
getDate():long
setDate(long):void
compareTo(Evenement):int
equals(Object):boolean
toString():String
execute():String
getCarte():Carte