Rapport de Simulation d'une équipe de robots pompiers

Groupe X

18 novembre 2020

1 Données du problème

1.1 Carte

La classe définissant une carte est disponible dans le fichier *Carte* du package *game*.

Une carte est représentée par une HashMap non-modifiable dont les clés, représentées par des entiers et correspondant aux cases, sont associées à la nature du terrain correspondant (type énuméré). Plus précisèment, une case (i, j) est identifié par l'entier $i * largeur + j$ où $largeur$ représente la largeur de la carte.
Au vu des opérations effectuées sur la carte, l'implémentation avec une table de hachage ou un array 2D n'impacte pas les performances de notre application. Par ailleurs, précisons qu'avec une HashMap on s'affranchit de tous espaces mémoires inutiles dans le cas où dans un futur proche, nous souhaiterions gérer le cas de cartes non-rectangulaires : le problème serait plus simple à gérer, à raison de modifier le système de coordonnées.

Les relations entre positions utilisent l'énumération *Direction* qui nous permet d'obtenir à la fois la position voisine suivant une direction donnée mais aussi la direction à partir de deux positions voisines.

1.2 Incendies

Les incendies sont stockées en attribut de la classe *DonneesSimulation* du package *game*.

Les incendies sont représentés par une table de hachage associant à une position entière l'intensité de l'incendie présent sur cette position. On peut ainsi itérer simplement sur tous les incendies, sans avoir besoin de parcourir toute la carte.

1.3 Robots

Toutes les classes définies pour les robots se situent dans le package *game.robots*.

Nous proposons une représentation des entités Robots inspirée du *design pattern type-object* favorisant la composition plutôt que l'héritage. Le type d'un robot peut en effet être décrit par ses attributs, ce qui nous permet d'instancier un singleton *RobotTypes* et d'obtenir de cette instance un robot ayant ce type (voir *TestRobots*). Cette conception ne permet pas l'ajout de méthodes propres à un type de robot. Tous les robots possèdent les mêmes méthodes qui donnent des résultats différents en fonction des attributs définis dans le type du robot. Les types de robots donnés par le sujet sont instanciés dans *MyRobotTypes*. Il est très facile d'ajouter un type de robot supplémentaire défini selon vos envies.

Un robot est caractérisé par son type invariant, son volume et sa vitesse modifiables. Un robot est une entité identifiable grâce à un identifiant unique, daté afin de pouvoir ordonner une suite d'évènements s'appliquant au robot et possédant un état, libre ou occupé selon qu'il y ait ou non des évènements s'appliquant au robot. La méthode *init* permet de réinitialiser les attributs du robot plutôt que d'en créer un nouveau à l'appel d'un restart.

1.4 Interface Graphique

Toutes les classes définies pour l'interface graphique se situent dans le package *game.graphics*.

Nous avons choisi d'utiliser des images pour représenter les robots et les types de case. La classe *gui.ImageElement* qui nous était fournie ne nous satisfaisait pas. Il semble en effet que l'image soit rechargée depuis le fichier à

chaque appel de la méthode *paint* et que l'implémentation ne permette pas de facilement superposer des images. Nous avons donc développé notre propre classe *TileImg* implémentant *GraphicalElement* pour représenter les images. Une image est caractérisée par un fond, sur lequel on superpose plusieurs motifs qui peuvent être les images des robots et/ou les dessins des incendies.

L'interface graphique est implémentée dans *GraphicsComponent*. Nous utilisons un buffer d'images pour ne pas avoir besoin de les recharger depuis leurs fichiers. La classe *NormUtil* est utilisée pour normaliser l'intensité des incendies. Les images de type *TileImg* sont stockées dans une ArrayList. Les attributs des images concernées par le changement d'intensité d'un incendie ou le mouvement d'un robot sont mis à jour à chaque appel de la méthode *draw*.

2 Simulation de scénarios

2.1 Données des simulations

Les données de la simulation se trouvent dans le package *game*.

Les données d'une simulation sont la carte, les incendies et les robots représentés par une HashMap associant à une position l'ArrayList des robots présents sur cette position. La classe *DonneesSimulation* comprend aussi les méthodes mettant en relations les différents objets de notre simulation afin de garantir un faible couplage entre nos classes. Nul besoin d'ajouter à notre classe *Robot* les méthodes permettant d'obtenir le temps nécessaire pour se déplacer d'une case à une autre et le temps nécessaire pour un robot de déverser l'eau sur l'incendie. Elles sont implémentées dans *DonneesSimulation*. Pour finir nous avons implémenté un constructeur qui copie l'instance de *DonneesSimulation* passée en paramètre.

2.2 Simulateur

Le simulateur se trouve dans le package *game*.

Un simulateur permet l'ajoût d'évènements à la simulation et implémente les méthodes *next* et *restart*. Il a donc besoin de plusieurs composants : les données de la simulation, l'interface graphique, un event manager s'occupant de l'ordonnancement des évènements et d'une stratégie optionnelle. La variable *INCREMENT* est utilisée pour ordonnancer une suite d'évènements.

Une action est une entité agissant sur les données d'une simulation. Représentée par une classe abstraite, une action doit nécessairement, pouvoir être exécutée et avoir une durée. Un robot peut effectuer plusieurs actions, chacune représentées par des classes filles :

1. ActionEmpty : déverser de l'eau sur un incendie.
2. ActionFill : se remplir si le robot est sur de l'eau ou à côté.
3. ActionMove : se déplacer vers une case voisine si c'est possible.

Un événement est décrit par une date et une action. Tous les événements sont gérés par une entité *EventManager* les ordonnançant selon leur date respective. Plus précisèment, nous disposons d'une file de prioriété basée sur un comparateur des dates des événements.

3 Calculs de plus courts chemins

Nous n'inventons rien, nous utilisons l'algorithme A^* [1] pour déterminer le plus court chemin entre une source et une destination.

3.1 Conception et Implémentation

L'algorithme A^* se présente comme suit : on cherche à déterminer le plus court chemin (c'est-à-dire ayant la distance parcourue/temps de parcours le plus faible) d'une source vers une destination. Pour se faire, on stocke un arbre de chemins partant de la source et on ajoute les nœuds aux chemins jusqu'à ce que le critère d'arrêt soit respecté, c'est-à-dire quuand on a atteint la destination.

1. https ://en.wikipedia.org/wiki/A*_search_algorithm

Plus en détails, à chaque itération, on choisit le chemin qui doit être prolongé en se basant sur le coût du chemin et une estimation du coût nécessaire pour atteindre la destination. Plus précisément, on sélectionne le chemin qui minimise $f(n) = g(n) + h(n)$ (*fscore*) où n est le nœud suivant sur le chemin, $g(n)$ (*gscore*) est le coût du chemin de la source à n et $h(n)$ est une fonction heuristique (spécifique au problème) qui estime le coût du chemin le plus faible de n à la destination. À chaque étape de l'algorithme, le nœud x ayant la valeur $f(x)$ la plus faible est retiré de la file d'attente, les valeurs f et g de ses voisins sont mises à jour et ces voisins sont ajoutés à la file d'attente. L'algorithme continue jusqu'à ce qu'un nœud supprimé x (nœud avec la valeur f la plus faible parmi tous les nœuds voisins) soit le nœud de destination. La valeur $f(x)$ de ce nœud est alors le coût du chemin le plus court, puisque $h(x) = 0$.

Concernant l'implémentation, nous disposons d'une file de priorité des nœuds, où un nœud est une sous-classe *Node* implémentant l'interface *Comparable<Node>* comparant chaque nœud par rapport à son *fscore*. Pour représenter les *gscore*, nous disposons d'une HashMap *gScore* dont les clés sont des entiers (position x) associées à des entiers ($g(x)$). Enfin pour stocker le chemin, nous disposons d'une HashMap *cameFrom* dont les clés sont des entiers (positions) associées à d'autres entiers (position voisine dans un chemin).

Enfin, comme expliqué précédemment, dès qu'on atteint la destination, on appelle la méthode *reconstructPath* pour reconstruire le chemin à partir de *cameFrom*. L'heuristique choisie pour calculer la distance entre un nœud et une destination est la distance de Manhattan (rien de bien compliqué c'est la distance associée à la norme 1 très utilisée dans les systèmes de quadrillage).

3.2 Avantages et Inconvénients

Comparé à l'algorithme de Dijkstra, nous savons où se trouve notre destination : nous n'avons pas besoin de parcourir en largeur toute la carte, nous avons une direction. Néanmoins, cet algorithme ne prétend pas être plus efficace que d'autres algorithmes de recherche du plus court chemin, notamment les nœud les plus proches de la destination pourraient ne pas être sur le chemin menant vers cette destination (car on ne peut pas y accéder), ce qui peut coûter beaucoup de temps de calcul.

4 Résolution du problème

Comme dit précédemment, notre simulateur permet de spécifier une stratégie. Nous avons réalisé les deux stratégies proposés par le sujet, définies chacune dans des classes filles héritées d'une classe mère abstraite *Strategie* contenant les définitions des méthodes essentielles à toutes stratégies, notamment *execute()* pour éxecuter la stratégie choisie. Aussi, on dispose des méthodes pour récupérer et associer un plus court chemin actuel à notre stratégie et d'un compteur interne permettant d'ordonner une suite d'événements (utilisé pour effectuer les actions des robots en série).

4.1 Stratégie élémentaire

4.2 Stratégie avancée

5 Expérimentations

5.1 Tests

5.2 Résultats

6 Conclusion