Simulation d'une équipe de robots pompiers

Rapport TPL de POO

Équipe 27

Durant ce TPL, nous avons été chargés de programmer une application de robots pompiers évoluant dans un environnement donné.

1 Organisation du code

Nos classes sont organisées dans plusieurs packages tel que suit :

`data` : Contenant la classe `DonneesSimulation`.

`event` : Contenant les classes relatives aux évènements : `AttributionTache, EteindreFeu, Evenement, PauseChefPompier, ReservoirRempli, RobotSeDeplace, SeRemplir`.

`fire` : Contenant la classe `Incendie`.

`io` : Contenant les classes `LecteurDonnees, Simulateur`.

`map` : Contenant les classes relatives aux cartes et aux positionnements `Carte, Case, Direction, Itineraires, NatureTerrain, Sommet`.

`robot` : Définit les différents robots et contient `Robot, RobotReservoir, RobotSansReservoir, Drone, RobotChenille, RobotPattes, RobotReservoir, RobotRoue, RobotSansReservoir, ZoneRemplissage`.

`strategie` : Contenant les classes `ChefPompier, StrategieNaive, StrategiePCC` relatives à la définition des différentes stratégies.

2 Choix de conception : Données du problème

Cette section présente notre choix de conception des différentes données du problème.

2.1 Les Robots

Les robots illustrent particulièrement l'importance de l'utilisation d'un langage orienté objet pour la modélisation du problème, en effet, leur conception repose essentiellement sur la propriété de l'héritage qui nous permet d'avoir plusieurs types de robots de caractéristiques différentes mais qui possèdent un "essence" commun.

Nous mettons donc en place une classe mère abstraite `Robot` définissant les attributs et les méthodes communs à tous les types de robots. Par exemple la possession d'une carte, d'une position, d'une vitesse, la capacité de garder en mémoire un trajet ou son l'incendie dont il est en charge ainsi que la capacité de se déplacer, de verser un agent incendiaire, de calculer son itineraire ...

De cette classe hérite deux classes abstraites, RobotSansResevoir correspondant aux robots ayant une capacité de réservoir infinie, et RobotReservoir correspondant aux autres robots. Cette distinction est importante puisqu'elle permet de réserver les opérations liées à la gestions du reservoir d'eau et tous les attributs s'y rapportant (capacité du réservoir,...) aux robots ayant des réservoirs, et permet ainsi d'éviter des actions incohérents.Par exemple une méthode remplirReservoir() n'a pas de sens si elle est appelée sur un robot à pattes.

De ces deux sous-classes héritent les classes qui définissent tous les types de robots que nous utilisons : Drone, RobotChenille et RobotRoue pour les robots à réservoir, et RobotPattes pour les robots sans réservoir.

2.2 La carte

Pour la représentation de la carte et de ses propriétés, nous utilisons trois classes : Carte, Case et Incendie. Ainsi que deux types énumérés : NatureTerrain et Direction.

Une instance de la classe Case représente une case de la carte, elle est définie par sa ligne et sa colonne ainsi que la nature de son terrain.

Une instance de la classe Carte représente une carte, elle est définie par sa hauteur, sa longueur et pour modéliser le plan de la carte, nous utilisons un array à deux dimensions de Case, qui paraît être le choix évident puisqu'il permet l'accès en temps constant aux différentes cases. L'attribut tailleCase définit la taille des cases. Nous stockons également les sources d'eau disponibles dans la carte dans un Set de cases; ceci nous permet d'optimiser la recherche des robots de la source d'eau la plus proche.

Finalement, les incendies sont représentés par la classe Incendie et contiennent les données relatives à la case où ils se trouvent (position), à l'eau nécéssaire pour les éteindre (eauNecessaire) et à leur état (atrribué, éteint).

2.3 Lecture des données et affichage

Nous utilisons le GUI fourni ainsi que le lecteur de données pour pouvoir initier et afficher les entités que nous utilisons à partir de cartes sous le format décrit dans le sujet. Pour mieux visualiser notre simulation, nous représentons chaque type de robot par une image de robot spécifique, de même pour les types de terrains. Pour les incendies, suivant leur intensité, c'est à dire la quantité d'eau nécéssaire pour les éteindre, ils sont representés par une image animée spécifique.

La figure ci dessous montre un exemple d'affichage du simulateur pour la carte Desert of Death.

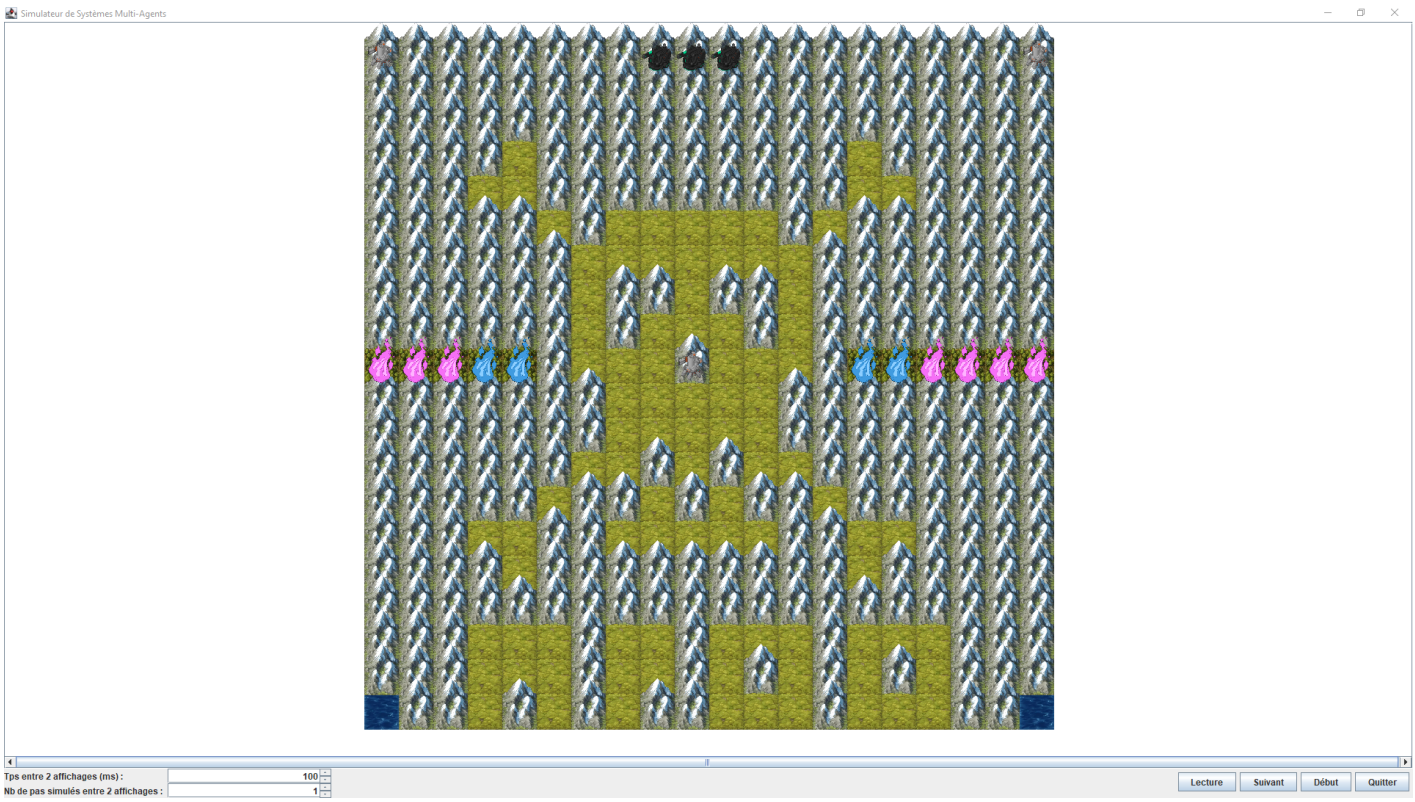

Figure 1: Simulateur carte Desert of Death

3 Résolution du problème

Nous présentons dans cette section les classes et méthodes utilisées pour la résolution du problème.

3.1 Calcul du plus court chemin

Pour le calcul du plus court chemin, l'algorithme utilisé est celui de Dijkstra, étant le plus performant pour des graphes ne contenant pas des poids négatifs.

Pour ce calcul, nous introduisons deux classes : Itineraires et Classe.

Une instance de la classe Itineraire est associée à un Robot, elle contient tous les plus courts chemins de la case du Robot à toutes les autres cases de la carte dans laquelle évolue le robot, ceux-ci sont contenus dans une Map<Case, Sommet>, qui pour chaque case associe le sommet correspondant, et un Set<Sommet> permettant de stocker les sommets non visités durant l'algorithme de Dijkstra.

Une instance de la classe Sommet est définie par la case à laquelle elle correspond, un coût correspondant au temps nécessaire pour le robot pour y accéder à partir de la case où il se trouve. Les coûts sont de type double afin de permettre des valeurs infinies, correspondant aux cases innaccessibles.

L'utilisation des collections disponibles en Java permet d'optimiser l'éxecution de l'algorithme de Dijkstra. En utilisant par exemple un TreeSet de sommets contenant les voisins des cases marquées, et en redéfinissant la méthode compareTo, nous définissons un ordre sur les sommets suivant le temps qu'il faut pour les atteindre et parvenant ainsi à selectionner en temps constant à chaque itération le sommet le plus proche. De même, stocker les directions dans une liste chainée permet d'insérer en temps constant au fur et à mesure les directions menant à une certaine case et à pouvoir les parcourir aisément.

3.2 Stratégies et événements

Nous définissons une classe abstraite Evenement correspondant aux évènements à éxecuter dans le simulateur et possédant comme attributs la date à laquelle s'exécute l'évènement et le simulateur sur lequel sont générés les évènements. La méthode abstraite execute permet de décrire l'exécution de l'évènement.

Événement

Les événements fonctionnent sous forme de cascade d'évènements c'est à dire que chaque robot ne possède qu'un seul événement futur, qui créera à son exécution un nouvel événement à une date ultérieure. Ainsi cela nous permet d'éviter le calcul de la date d'exécution des événements et simplifie la gestion de l'éxtinction des feux dans le cas de multiples robots sur le même feu.

Nous définissons plusieurs sous classes correspondant aux différents évènements s'exécutant durant notre simulation :

- **AttributionTache**: Attribue un incendie à un robot, calcul le chemin du robot vers l'incendie et appelle **RobotSeDeplace** ayant pour prochain événement **EteindreFeu**.
- **RobotSeDeplace**: Demande au robot de se déplacer sur un chemin donné, à la fin du chemin créer nextEvent, l'événement donné en entrée symbolisant le prochain événement.
- **EteindreFeu**: Demande au robot de verser son réservoir sur le feu. Une fois que son réservoir est vide, crée un événement **SeRemplir**. Une fois que l'incendie est éteint libère le robot.
- **SeRemplir**: Demande au Robot de se déplacer à la source d'eau la plus proche et de s'y remplir. Pour celà, on calcule le nouveau chemin et on crée un évenement **RobotSeDeplace** ayant pour nextEvent l'événement **ReservoirRempli**.
- **ReservoirRempli**: Rempli le reservoir du robot. Dans le cas où l'incendie attribué est éteint, libère le robot. Le cas échéant, met à jour le chemin pour le diriger vers l'incendie et crée un événement RobotSeDeplace ayant pour nextevent **EteindreFeu**

Stratégie

ChefPompier est une classe modélisant le chef des pompiers dont le rôle est de distribuer les tâches et de coordonner entre les robots, la classe possède comme attributs le simulateur à laquelle elle est associée et des listes chainées des robots et des incendies présents sur la carte.

Nous avons organisé les stratégies en suivant le design pattern "strategy" qui permet d'être plus flexible dans le reste du code. Ainsi nous avons une classe mère ChefPompier avec uniquement deux méthodes publiques : son constructeur et execute(). Toutes les stratégies vont hériter de cette classe. On aura alors à choisir une stratégie en instanciant une des classes qui implémentent une stratégie (une des classes filles) avec un type statique ChefPompier et le reste du programme n'aura qu'à appeler sa fonction execute() indifféremment de la stratégie choisie.

Nous avons implémenté deux stratégies. Une première stratégie naïve (classe StrategieNaive) qui est une version améliorée de la « startégie simple » donnée par le sujet :

-Le chef pompier va chercher un robot libre. -S'il en trouve un , alors il lui assigne un feu. -S'il n'en trouve pas, il attend quelques instants et recommence sa recherche. -Le chef pompier fait cela tant que tous les feux ne sont pas éteints.

-Un robot qui se voit attribuer un feu, place son attribut occupee sur true et se déplace vers le feu pour l'éteindre. Tant que cet attribut est sur true, le chef pompier (classe ChefPompier) ne lui proposera pas d'incendie. Lors de l'intervention, s'il lui manque de l'eau, il va en chercher au point d'eau le plus proche, en toute autonomie. Une fois l'incendie éteint, il met son attribut sur false et redevient un robot libre du point de vue du chef pompier.

La seconde stratégie (classe StrategiePCC) est une légère modification de la première stratégie lors de l'attribution. Elle consiste à choisir systématiquement le couple (incendie, robot) les plus proches. Ainsi l'ordre de lecture et de stockage des incendies et des robots ne rentre plus en compte dans le choix des attributions. Pour cela, à chaque fois que le chef pompier souhaite attribuer un feu à un robot, il demande à chaque robot libre de lui donner le temps de parcours du chemin le plus court vers tous les incendies non attribués. Il recupère le robot qui a renvoyé le chemin le plus court, ainsi que l'incendie associé et émet un ordre d'attribution.

4 Tests réalisés

L'exécution de notre programme sur les cartes fournies permet de valider son fonctionnement, et ce, en utilisant les deux stratégies :

-La simulation se termine par l'extinction de tous les incendies présents sur la carte quand cela est possible. C'est à dire quand au moins un robot dans la carte parvient à avoir un réservoir rempli et à accéder à la case de l'incendie.

-Les robots prennent les plus courts chemins pour accéder aux incendies ou aux sources d'eau.

-Quand un robot éteint un incendie, il peut se voir réattribuer un nouvel incendie.

-Quand un robot ne peut atteindre aucun incendie, il reste immobile tout au long de la simulation.

-Un robot ne peut pas accéder aux cases dont le terrain n'est pas compatible avec son type, ainsi qu'aux cases dont il n'existe pas de chemin fait uniquement de cases de terrain compatible. Un robot ne se voit ainsi pas attribuer un incendie auquel il ne peut pas accéder.

-Les robots se ressourcent plusieurs fois quand cela est nécéssaire. Les robots n'ayant pas de réservoir ne se ressourcent jamais, ceux en possédant un le font uniquement quand leur réservoir est vide.

-Un incendie n'est jamais attribué à plus d'un robot à la fois, et un robot ne se voit jamais attribuer plus d'un incendie à la fois.