GRENOBLE INP UGA Ensimag

https://ensimag.grenoble-inp.fr/

Object Oriented Programming : Project report

Fire robots

Object Oriented Programming
Academic year 2022 - 2023

Table des matières

Introduction

Our project concerns the development of an application in Java allowing to simulate a team of Fire robots operating autonomously in a natural environment. In this report, we will detail our approach, our design choices and the strategies implemented to meet the expectations of the application. Thus, we will present the main results thanks to the different tests we developed.

1 Design choices

1.1 Package `Carte`

Our `Map` is represented by a two-dimensional array of `Square`s. Each `Square` is defined by its row and column. In addition, the uniqueness of each `Square` is guaranteed because the creation of `Square`s is only carried out during the generation of the `Map`, and no method subsequently allows for the modification of `Square`s or the creation of new ones. A `Map` contains methods to check the existence of the neighbors of a `Square` and to iterate through them. The `Fires` are in this package and are defined by their `Square` position and their intensity. The `Fires` as well as the `Maps` have an attribute `SimulationData` so that each of the two classes has access to the other. Thus a `Map` does not manipulate the `Fires` and vice versa, only `SimulationData` centralizes the management of data, while `Fire` and `Map` are only data structures.

1.2 Package `Robot`

`Robot` is an abstract class defined by `SimulationData`, a position and a type that gives it specific characteristics : a speed, a water volume, a linear intervention time and a filling time. They also have a potential `Fire` to which they could be assigned and an availability. The child classes consist of the 4 types of `Robots` : `LEGS, TRACKS, WHEELS, DRONE`. Each method related to a `Robot` action is located in this package and respects the principle of inheritance as much as possible.

1.3 Package `Events`

Each `Event` is defined by the `Simulation` to which it belongs, the `Robot` that will perform this task and the date on which this `Event` will be executed. The abstract class `Event` has an abstract method `execute()` specific to each class of `Event`. It may happen that during the execution of an `Event`, a similar `Event` is added to the scenario to repeat this action in a loop at regular intervals.

1.4 Package `Simulation`

1.4.1 Class `SimulationData`

A `SimulationData` class centralizes all the data of the `Simulation`. The `Fires` are in the form of a Hashmap<`Square`,`Fire`>, allowing us to check if a `Fire` is on a given `Square` and access it with constant complexity. Due to the uniqueness of the `Square`s and the use of the Hashmap, we guarantee only one `Fire` per `Square`.

The `Robots` are stored in a Queue. In fact, it was impossible to put our `Robots` in a Hashmap < `Square`,`Robot` >, because in this case there could only be one `Robot` per `Square`.

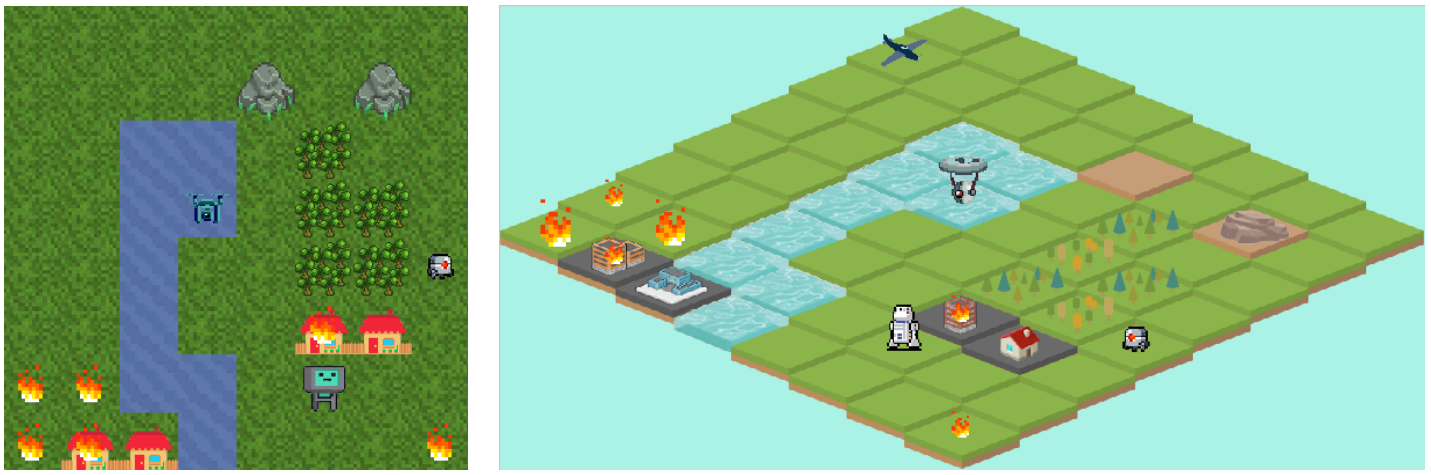
We also implement a method returning an iterator on the `Fires`. As deleting elements from an iterator poses a problem, we chose to leave an extinguished `Fire` in the data structure but completely ignore it (we no longer draw it and interact with it). `SimulationData` also has an attribute `Map`, as well as an attribute `dataFile` to restart the `Simulation` with the Start button (Restart).

1.4.2 Class `Simulator`

It is in the `Simulator` class that the `Events` are added to a PriorityQueue representing the scenario. The `Simulator` will execute all the `Events` whose date attribute is equal to the current date, then draw the `Map`, the `Robots` and the `Fires`. Once finished, it increments the current date.

1.5 Graphical aspect

It is possible to display the `Map` for small basic tests in a command terminal using the `toString()` method of `Map` (cf 3.1). We first implemented a 2D version of the `Simulation`, then an isometric 3D version. The `Robots` and `Fires` are animated and the size of the `Fires` adapts to their intensity. Each `NatureTerrain` `Square` has different possible assets that are chosen randomly, so for each `Simulation` the map is unique.



(a

FIGURE 1 – Graphical aspect of the `Simulation`

2 Implemented strategies

2.1 Calculation of the shortest path

Operations and methods related to the calculation of the shortest path are located in the `Strategy` package and not with the `Robots`. This separation principle that we have applied throughout the project facilitates the maintenance of the code, such as changing strategy quickly. We chose to implement the Dijkstra algorithm although the A* algorithm could have been considered. A Dijkstra function takes as arguments a starting `Square`, destination `Square`, and a boolean searchWater.

There are 3 possible cases :

- **Case 1** : knowing the destination `Square`, searchWater is defined to `false` and the shortest path is returned.
- **Case 2** : searchWater is equal to `true`, the shortest path to the nearest WATER square is returned.
- **Case 3** : searchWater is set to false and no destination is specified (null) : the shortest path to all accessible `Squares` is then calculated.

These 3 cases define the 3 functions `calculatePath`, `destinationPath`, and `fillPath`.

2.2 Strategy

2.2.1 Basic strategy

We iterate over the `Fires` :

1. If there is an available `Robot`, we assign it the `Fire` and calculate the shortest path.
2. The `Robot` will be available once the `Fire` is extinguished.
3. If there are no more fires to assign, the `Robots` will remain stationary.

2.2.2 Final strategy

We iterate over each `Robot` :

1. We ask if it is available, if not then we do nothing.
2. If it is available, we calculate its shortest paths to the `Fires` and assign it the closest one.
3. The `Robot` will then head towards the fire and will be unavailable until that fire is extinguished. The `Robot` automatically refills its water tank.

Optimisations de la stratégie
Strategy Optimizations

(a) If all the `Fires` are assigned, we send an available `Robot` to a `Fire` that is already being taken care of by another `Robot`.

(b) If all the remaining `Fires` are unreachable for a `Robot`, then that `Robot` heads towards a reachable `Fire` that is already assigned.

(c) We have implemented the propagation of `Fires`. Each new child `Fire` has an intensity equal to half of its parent's. When a `Fire` has an intensity lower than a limit value, it can no longer propagate. There can be no `Fire` propagation on water.

3 Tests and main results

In order to make our code more robust and clearer, we created new exceptions : `TerrainIncorrectException`, `VitesseIncorecteException` and `VolumeEauIncorrectException`.

3.1 TestCarte

TestCarte is the first test developed in our project. It allows to display a `Carte` and to verify the implementation of `LecteurDonnees`.

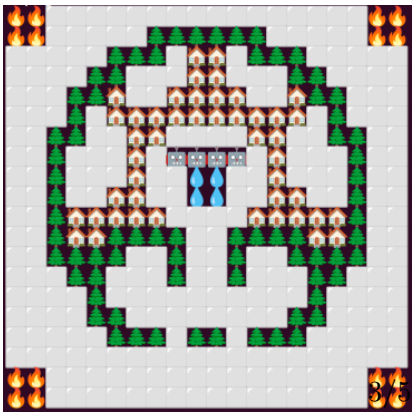


FIGURE 2 – Display in shell

3.2 TestRobot

TestRobot allows us to ensure the stability of our different implemented entities, before approaching the notion of `Simulation` and `Evenements`. The test generates a simplistic `Carte` various interactions are possible with the user in order to test the different methods. A single `Robot PATTES` is present on the `Carte` and can evolve according to the specified inputs.

3.3 TestDijkstra

TestDijkstra allows us to test our implementation of the shortest path from a `Robot` to an `Incendie`. The test displays the Mushrooms of Hell map (in a simplistic version). We specify the `Case` where we want to move and we can follow our `Robot` step by step.

3.4 TestSimulateurOK

TestSimulateur is the first test implementing the notion of `Evenement`. The test creates a `Simulateur` where events are added manually. The test therefore displays the sequence of `Evenements` according to the predefined scenario, which moves the `Robot`, fills its tank and pours water on a fire (whose position was previously known).

3.5 TestChemin

TestChemin follows the principle of TestDijkstra, by adding the management of successive event additions automatically through the creation of a path.

3.6 TestSimulation

TestSimulation shows the final test of our project. It displays the different parts of the project on our final graphical interface. Thus, we can observe a `Map` with `Fires` of varying intensity and different `Robots`. Once the `Simulation` is launched, we can see the `Robots` evolving within the `Map`, implementing the strategy implemented to extinguish each `Fire` as quickly as possible. The `Simulation` ends when all the `Fires` are extinguished.

3.7 TestStrategie

TestStrategie is initially an interactive debugging test, but it seemed relevant to make it available. Through this test, we can generate a `Map` with a specified number of `Fires`, each at a chosen location and at a fixed intensity, a specified number of `Robots`, each at a location and according to the specified type. Once the `Map` is generated, we can observe the `Robots` implementing the strategy calculated by our `Simulation`, until all the fires are completely extinguished.

3.8 Test of optimizations

— Test of optimization (a) (several `Robots` on the same `Fire`) : visible on the testSimulation.

— `exeOptimisationB` : Test of optimization (b) (ignoring unreachable `Fires`).

— `exeSimulateurPropagation` : Test of optimization (c) (test of the propagation of `Fires`).

— `./testCarte.sh <n> <PATHTOMAP>` : shell script to run the simulation with or without propagation for any map. Put n to 1 for a simulation with propagation, 0 otherwise. Test with the map `desertOfDeath-20x20.map` to see that the propagation of Fires beats the `Robots`.

This report is a translation of the report originally written in French, as is the code. In this report, some French words, such as packages or robots name, have been translated for clarity. Here are the words you will find in the code and their corresponding translation used in this report :
Map → Carte
Square → Case
Fires → Incendies
SimulationData → DonneesSimulation
Evenement → Event
LEGS, TRACKS, WHEELS, DRONE → PATTES, CHENILLES, ROUES, DRONE