Programmation Project 2
Part 3 Report

Ardoin, Lichtlé, Riou

May 23, 2018

1 Introduction

For this last part of the project, we focused on the ability of saving and loading a game and on the making of a small artificial intelligence. We also added, as requested, the ability for the player to see statistics about the game and how well they are doing economically speaking, as well as the statistics of the artificial intelligence. Finally, we added a bit of dynamism to the network by adding acceleration and deceleration, loading times and random breakdowns.

2 Charts

We have added a new tab whose purpose is to display various economics statistics to the player. They do not update in real-time as the different variables responsible of storing the data to be displayed are not properties; however, the statistics are always up to date and it is possible to see them from the start of the game up until the current moment.

A first button, `Stats`, displays some textual data. It includes the profits made by each type of vehicle (trains, boats, planes) by transporting passengers or goods, and the expenses created by these same vehicles, mainly coming from fuel consumption. Are also displayed the spendings the player has made and whether it is by buying structures, roads or vehicles.

Next to that are three other buttons to display charts. There are 3 pie charts showing a graphical representation of the data introduced above, that is the profits, expenses and spendings. As it is a pie chart, it only shows relative data, so we also added a bar chart displaying the profits and expenses one next to the other so as to be able to compare them.

Finally, the chart containing the most information (and being the most beautiful) is probably the line chart. It display the total net income of the player and breaks it into the incomes from the trains, boats and planes. Moreover, it is displayed from the start of the game to the current moment, which allows to see the evolution of the ingame economics and which type of vehicle is the most - or the less - profitable to the player. The data is saved every one second, which means every hour in-game (in case the game is not accelerated).

The charts are drawn in the class `Charts` and make good use of the class `Player`. To make it work, we adapted the functions `earn` and `pay` of the class `Player`, which respectively allowed us to make the player gain or lose money, so that they get some information about where is the payment coming from. It could be that the player has bought a structure, or maybe he has earned money from plane passenger tickets. In any case, it is stored in separate variable so the charts can show all these data separately.

Last but not least, we had already added in part two arguments of the form `owner: Player` to our structure and vehicle classes. Thanks to that, the statistics created above are independant for each player. As a result, it was very easy to add a tab for the stats of the AI: we just had to create a new instance of the class `Chart`, which takes a player as a parameter, and voilà.

3 Dynamic network

The first change we added is to let trains have loading and unloading times. In part two, when the engine arrived into a town, all its carriage instantly disappeared and the train was back on tracks towards its next stop. Now, the carriages will finish their journey without disappearing and the

train will wait for the last carriage to have arrivied before leaving the town, after having taken a short unloading break.

Then we added an acceleration feature. Vehicles progressively gain speed when leaving a town, or their departure infrastructure, up to a certain maximal speed. They continue their journey, and finally they start losing speed when eventually reaching their destination. Also, trains may have technical issues at some point, which forces them to brake and go to their destination at low speed.

4 Loading & Saving

For saving the map, we used an `xml` file, like we did for the map loading in the previous part of the project. Each entity is saved along with its properties such as its position, its name, its population for a town... Some general information about the game is also saved, such as the current time or the player's money. Besides, the map, which was generated automatically when we pressed *New Game*, is now also saved; each tile is individually saved with its type (grass, water, sand...). This is probably not the most efficient way to save the map (just for a 200×200 map, $80,000$ lines of `xml` are generated), yet it is quite fast. Some information though we didn't manage to save, like the vehicules trips or the goods.

We handled the loading of the `xml` map in the same way we did in the second part of the project. Notice that we use a different sytax that does not interfere with the previous format of map we had to load. The map background is first loaded, tile by tile. Then comes the buildings, followed by the rails, and finally the vehicles. The structure in which a vehicule should be created is given with by coordinates coresponding to the last structure the vehicle has visited. Besides, all the entities that are loaded are created using the same functions that create entities when the player is playing, that is why it is imperative for the different entities to be created in the good order when loading the map, so everything is well connected.

5 Artificial intelligence

We decided to create an artificial intelligence which can play and interact with the real player. To implement its decisions, we use a structure of N-ary trees and a stack called `Waiting actions` (actually it is a list used as a stack). The leaves of the trees represent the different actions the AI can do, whereas the stack represents the actions the AI has planned to do but has not done yet.

The artificial intelligence thus works thanks to two functions: `chooseAction` which fills the stack by finding a path in the tree, and `executeAction` which empties the stack by executing the chosen actions.

Using a stack to save the following actions is necessary because our vision of the Tycoon Game is very restricted in terms of possible actions. For instance, let us suppose that the AI would link two cities by a plane trip: this is possible if and only if both cities do have an airport (and if the two cities exist of course...). As a result, the AI cannot just execute this action directly; therefore it pushes it on the stack and then checks if there is an airport in both cities. If there is, the link will be created at the next iteration by popping the stack, otherwise the action "create an airport in the cities to connect" will be pushed further on the stack. These choices of actions also take into account every pre-required constraint, thanks to a tail-recursive algorithm, which takes advantage of lists and trees structural specificities.

The gameplay is just a cohabitation between the AI and the player. A flag is displayed on each structure and its color indicates its owner (that is, its creator). However, the structures created by the AI can very well be used by the player, and vice versa.

6 Conclusion

For this last conclusion, we think the project was very interesting and we have learnt a lot of different concepts, from Scala itself and documentation search to XML parsing, and even Brownian motion for the generation of random maps... We even learned about genetic algorithms although we did not manage to implement one for our AI.

All in all, we may not have done so much with the graphics of our game had we knew what we would have to do. It always made us be one step behind when it came to implenting the features

you were expecting. However we put a lot of OOP into it and learned some key concepts so it was worth taking the time.