Scotland Yard

Razvan Dan David, Stefan Pruna

May 2, 2019

1 Model

The first part of the coursework was *cw-model.* Our task was to pass all tests by completing the implementation of the ScotlandYardModel class. This involved implementing the constructor, the *ScotlandYardGame*, *Consumer* and the *MoveVisitor* interfaces, but also some auxiliary methods.

The constructor

The constructor for the *ScotlandYardModel* class ensures that all data that was passed in as arguments is valid and can be processed and copied into the class data structures. There are many types of validity checks, including checks for null pointers, valid colour and locations and appropriate tickets. *PlayerConfigurations* are converted to *ScotlandYardPlayers* and added to a list of players that is used later on in the class.

The ScotlandYardGame interface

There are many methods that we wrote in order for our class to implement the *ScotlandYardGame* interface, but two that stand out as being very important for the gameplay are *startRotate* and *isGameOver* (or *getWinningPlayers*). In the methods used to check if the game is over or to get the colours of winning players, we checked four game-ending conditions: if there are more rounds to be played, if mr. X or the detectives are stuck and if mr. X has been captured.

Valid move generation

In order to correctly implement *startRotate*, we needed to generate a collection of all valid moves a player can make from their positions and with their ticket configuration. We did this by iterating over all the graph edges available from the player's current location and adding the simple moves that the player has the tickets to make to a HashSet. To generate the double moves, we checked if the player has a double move ticket, and iterated over the available edges for all the previously generated simple moves, finding new available moves and combining them into a double move.

Move accepting callback, visitors

Players are given a collection of valid moves that they can make, and a method to call when they decided which move to make. This method, *accept*, is included in the *Consumer* interface, which we implemented. It takes a generic *Move* object as argument, checks if the selected move is valid and executes game logic. In order to get the type of move the player selected and to be able to execute game logic, we made use of the visitor pattern by implementing the *MoveVisitor* interface. This meant creating three overloaded methods in which we had access to the type of move used by the player. In these three methods, we removed the used tickets, added tickets to mr. X, moved players, incremented rounds and current player turns, and notified spectators.

2 AI

The second part of the coursework was *cw-ai*. Our task was to implement an artificial intelligence algorithm that can choose the best move for mr. X. We had to create a scoring method and implement the MinMax algorithm, but also come up with creative ideas to make our program more efficient.

The scoring method

We decided to use three scoring criteria involving the available moves of mr. X, the average distance to the detectives and the minimum distance to them. To compute the distances from mr. X to the detectives, we used a Breadth First Search algorithm which we started from every detective position, taking their ticket configuration into account. We normalized the three scores such that each of them would range between 0 and 100. After some trial and error, we ended up combining these three scores as follows: 70% of the final score is the minimum distance to a detective score, 20% is the available moves score, and 10% is the average distance to a detective score.

MinMax

In order to make our A.I. more powerful we implemented the MinMax algorithm and optimized it with Alpha-Beta pruning for more performance. We generate the game tree recursively with a maximum depth of three and choose the best move mr. X can make using the scoring method we created. In order to model the game state, we created two new classes, *ScotlandYardAIModel* and *ScotlandYardAIPlayer*, inspired from *ScotlandYardView* and *ScotlandYardPlayer*, but which support modifying the game state in an easier way.

Caching

Because starting Breadth First Search algorithms for each possible move MinMax generates is computationally costly, we came up with a Caching system. We generated a 5D matrix (200*200*12*9*5), with the following mapping: (start node, end node, number of taxi tickets, number of bus tickets, number of underground tickets). We stored the minimum distance in which a player with a certain 3-tuple configuration of tickets (last three dimensions of the matrix) can go from one node to another (the first two dimensions). This matrix takes about 60 seconds to be computed. As such, we saved it as a binary file (21.4 MB of size), that we can easily load when the A.I. is first started.

3 Reflection

BFS After implementing the Breadth First Search, we figured out that there were other ways in which nodes could be reached in the same amount of steps, but with a better future ticket configuration. We modified our algorithm such that it also considers the cases in which a player can move using two different types of transportation to the same node, generating different states in which that node can be reached.

Caching We also implemented a caching system for the number of valid moves a player can make from each node, similarly to the minimum distance cache explained earlier. Later on, we realised that the data stored in this cache was flawed, as we didn't take other players standing in the way into account. We decided to leave the source code, but generate the number of valid moves for each MinMax call.

Further performance improvements We think that to further improve the performance of our A.I., we could simulate some of the configurations generated by the MinMax algorithm on different threads. We decided not to tackle this further optimisation, as we lack the experience in parallel computation.

Conclusion By doing this project, we have learned valuable lessons about object-oriented programming paradigms and team work. These include the usage of github or other version control software, the practicality of the visitor design pattern in big software projects like this one, and the importance of planning ahead when writing code.