Scotland Yard Report

Laura Pozzetti, Vardhini Bhasker

//Abstract

For this assignment we implemented a model of the Scotland Yard game following the model specification and the given guide. As an extension, we created two versions of a Mr X AI using different algorithms. We also wrote a simple AI for the detectives to play against the two MrX AI components, in order to compare the effectiveness of the algorithms used.

// About Scotland Yard Model

We developed the model for the game using a test driven approach, keeping in mind the agile development practices mentioned in lectures.

We started by implementing the constructor `ScotlandYardModel`, completing field assignment and input validation for all the attributes `rounds, graph, view.` In order to make the code in the constructor more object oriented, we created smaller methods outside of the constructor (e.g. `isMissingTickets`, `hasSingleTickets`).

In order to check for consistency in the players, we added the attributes mrX and list of detectives as instances of the class `ScotlandYardPlayer`. In our implementation, we decided to use array lists when holding `ScotlandYardPlayer` objects. We used this data structure in order to ensure that players are ordered in the same way as they are in the constructor so that there were no errors in `MatchesSupplied` tests. We also decided to use hash sets to hold objects such as colours and location and check that there were no duplicate members of a set in our constructor.

Next we implemented all the methods in the `ScotlandYardModel` class that arise from the interface `ScotlandYardView` (e.g. `getRound`, `getPlayer`). These were instance methods with the same signature so they override the methods in the superclasses.

We proceeded by creating a set of valid moves and implementing the `startRotate` method, which starts each playing rotation and calls `makeMove` for each player.

To create the set `validMoves` we decided to create two sets `validTicketMoves` and `validDoubleMoves`. This allowed us to implement the two different validation logic separately, and to have a more object oriented code.

We created an inner class ScotlandYardVisitor which implements `MoveVisitor` to use the visitor design pattern. This enabled us to perform operations on `ScotlandYard` using a visitor class so that it would not be necessary to modify source files. By using this behavioural pattern, we were able to implement the play logic by creating visit methods depending on the type of move chosen. The

accept method in `ScotlandYardModel` received this visitor object and called the visit method on the visitor object, thus implementing the play logic through double dispatch.

Finally, we implemented the `spectator`-related features and notifications using the behavioural design pattern observer. Upon completion of this, we passed all of the given tests.

Once we had completed and played the game, we decided to further improve our code by making it more object oriented. We substituted sequential logic and long conditional statements with new methods in the class. For example, we created the method `noDetectiveIsAtDestination` to make our `validMoves` code shorter and more elegant.

// About the extensions

We implemented two versions of the AI component for mrX (named `MrXAI` and `MrXAIminimax`) and a simple AI for a detective. Both of them use the creational Factory Method design pattern by calling the `createPlayer` method given in the `PlayerFactory` interface without having to specify the class of the `MyPlayer` object.

`MrXAI` uses dijkstra's algorithm to calculate minimum distance to nodes from Mr X's node and puts these distance values into a hashmap with the node itself being the key. In the `getBestMove` method, we create a scoring function giving importance to distance to detectives and number of detectives that were 1,2 or 3 moves away.

`MrXAIminimax` uses dijkstra to calculate the distance of the players from every node on the map, in terms of number of moves and tickets. If there are more than two players the best move is chosen through a scoring system that takes into account a safe distance from the detectives, the availbility of tickets, and the round. If there are only two players `MrXAIminimax` uses the algorithm Minimax to evaluate the evolution of the game over a larger number of moves. Alpha-beta pruning has been applied to the minimax tree for efficiency purposes.

In order to compare the two AIs and reflect on the effectiveness of these algorithms in projects like ours, we made them play against the detective AI 100 times. Despite `MrXAIminimax` winning slightly more, the Mr X success rates that we got are similar, so we plan on taking part in the AI competition to properly test our AIs.

// Improvements and further development

Given more time we felt that we could have improved our `ScotlandYardModel` (perhaps by using more advanced interfaces and data structures) and especially our AIs. None of the AIs that we created checks that the opponent is suitable. For example, it is possible to select a Mr X AI for all 6 players, and the game goes on in most cases without any apparent problem until the end. We could implement some checks and make it impossible to select more than one MrX per game.