Scotland Yard Coursework Report

1. Introduction

Scotland Yard is a board game where player(s) called *Detectives* are tasked with capturing another player called *MrX* as they move around the board. Within this report, we review our implementation of a digital version of this board game explaining and verifying our development model and evaluating the ability of our implementation.

NOTE: We found a previous year's solution to the same problem on GitHub as it was made public and for *cw-model* we took a fair amount of inspiration from their implementation. Although, we find that their solution was inefficient and lacking optimisation in certain parts.

Link to Credited Party's GitHub: https://www.github.com/armandcismaru/ScotlandYard

2. Model

cw-model is the first part of the implementation, responsible for carrying out the majority of the back-end functionality for the game. We were supplied with a testing framework consisting of 83 tests that would test against the model we create; in which all 83 tests were **passed**. Below, we showcase our implementation for this part of our project, which consists of two files, *MyGameStateFactory.java* and *MyModelFactory.java* .

NOTE: We define 'Getter' methods as methods which simply return an attribute of a class, hence some methods may be labelled in similar notation to 'Getter' methods, however they have not been listed as such. This is because those methods perform a fair amount of computation and do not simply return the value of a private attribute in a few lines of code.

a. MyGameStateFactory

`MyGameStateFactory` is an implementation of `Factory` with a single method: `build`, which itself returns a new `MyGameState` object, a subclass of `MyGameStateFactory` that implements `GameState`.

i. Attributes

The attributes of `MyGameState` are as follow:

- `setup`, (of type `GameSetup`) holds the setup of the current game board,
- `remaining`, (of type `ImmutableList<Piece>`) holds the pieces left to move in the current round; it alternates between the detective pieces and the MrX piece,
- `log`, (of type `ImmutableList<LogEntry>`) holds MrX's travel log,
- `mrX`, (of type `Player`),
- `detectives`, (of type `List<Player>`),
- `everyone`, (of type `ImmutableSet<Piece>`) is a set of all the pieces present in the current game,
- `moves`, (of type `ImmutableSet<Move>`) holds all the legal moves that the pieces can make in their current positions on the game map,
- `remainingRounds`, (of type `ImmutableList<Boolean>`) lists the truth values of the reveal state of the remaining rounds in relation to MrX's travel log.

All the attributes are initialised in `MyGameState`'s constructor, as they are passed as arguments when the object is created, or they are derived from the passed arguments.

ii. Methods

- `advance`

- takes as argument of type `Move` and then it calls in turn its helper methods to advance the game to the next game state if the move is legal.

`advance`'s helper methods are:

  - `updateRemaining` - updates the `remaining` attribute to contain all the detectives present in the game if the move was commenced by MrX. If the move was commenced by a detective, `remaining` is updated to contain either the detectives who have not yet made a move and have at least one ticket, or, if the detective was the last one in `remaining` in the previous game state, `remaining` is updated to contain only MrX;
  - `updateLocations` – updates the location of the player who commenced the move. This method makes use of the visitor pattern to get the final destination of the move, so there is no different case for single or double moves;
  - `updateTickets` – updates the tickets that each player holds. This method also makes use of the visitor pattern as the `ImmutableList<Ticket> tickets` holds all the tickets used in the move. If the move was commenced by a detective, then MrX receives said tickets;
  - `updateLog` – updates MrX's travel log. The visitor pattern is used twice, to get the destinations of the move and the tickets used as immutable lists. For each destination of the move, a new log entry is added to `log` in accordance to the reveal state of the round.

- `getDetectiveLocation`

- takes a detective as argument and returns their location as an `Optional<Integer>`

- `getPlayerTickets`

- has a subclass `MyTicketBoard` that implements `TicketBoard`. The function then returns an `Optional<MyTicketBoard>` in relation to the player which is passed as an argument.

- `getWinner`

- returns an immutable set of pieces when one of the winning states is reached. The method covers all the possible winning scenarios, both for the detectives and for MrX. If a winning state is not reached in the current game state, an empty set is returned.

Winning states:

detective winning scenarios:
- ❖ MrX is captured
- ❖ MrX is stuck
- ❖ MrX has no tickets left

MrX winning scenarios:
- ❖ the last remaining round was played and MrX was not captured
- ❖ no detective has any tickets left
- ❖ all detectives are stuck

The 'stuck' and 'no tickets' scenarios for both the detectives and MrX are determined by the state of the set returned by `getMoves`. If the set is empty, then the conditions for these scenarios are met and the winners are returned.

- `getAvailableMoves`

- returns a immutable set of all possible moves if the game has not ended. The state of the game is determined by the state of the set returned by the `getWinners` method; if the set is empty, then a set of moves is returned; if the list is not empty, then an empty set is returned.

`getAvailableMoves` makes use of `getMoves` to get the set of all possible moves.

- `getMoves`

- is a helper function for both `getAvailableMoves` and `getWinner`. It returns an immutable set of all possible moves of the pieces that are present in the `remaining` attribute.

If `remaining` contains detectives, then `getSingleMoves` is called for each detective and the set of all possible single moves for all detectives is returned. Otherwise, if remaining contains MrX, `getSingleMoves` and `getDoubleMoves` are called and a set of all possible moves is returned.

`getMove`'s helper methods are:

  - `getSingleMoves` – goes through all adjacent nodes where a detective is not already present and if the player has the required ticket to travel to that location, the move is added to a list. If the player has a 'secret' ticket, then they can travel to any non-occupied adjacent node regardless of the required ticket type.
  - `getDoubleMove` – calls `getSingleMoves` twice; once on to establish all possible single distance moves, and once to determine all distance two moves. The method also checks for the presence of a 'double' ticket.

iii. Getters

`getSetup`, `getPlayers`, `getDetectiveLocations`, and `getMrXTravellLog` all return attributes of `MyGameState`.

iv. Evaluation

Our code for `MyGameStateFactory` was largely structured around GitHub user armandcismaru's code, however we consider that our solution is a significant improvement in terms of readability and modularity of the code, as well in terms of efficiency. When ran on the same machine, our code passed all 83 tests ~20 ms faster on average than theirs, and the t-test indicates that the difference in runtime is statistically significant, with a p value of ~0.025 when ran a 20 sample data set.

Due to our findings, we can reject the *null hypothesis* H_0 that theorised our *cw-model* runtime to be more than or equal to the runtime of the credited party and instead accept our *alternative hypothesis* H_1 that speculated our runtime to be less; allowing us to infer that our implementation of this part of the project is indeed more efficient.

We believe that our solution can be further improved upon by optimising the `getDoubleMoves` method by using dynamic programming principles.

Our code presents a warning in the IntelliJ IDE stating `Condition 'mrX == null' is always 'false'`. This cannot be avoided due to the nature of one of the tests that requires for a null pointer exception to be thrown in the case that the MrX object is null.

b. MyModelFactory

This part is responsible for constructing and executing the *Observer Pattern* technique in OOP, which is essential and very efficient for use in this project. Below is shown the different parts of its implementation:

i. Attributes

Below are the attributes for `MyModelFactory`:

- observers, (of type`ImmutableSet`) holds all current observers,
- state,(of type `GameState`) stores the new `GameState` when built with `build`.

ii. Methods

- `registerObserver`

- Register method for *Observer Pattern*, takes an `Observer observer` assigns a new `Observer` to the `observers` list while still maintaining immutability.

- `unregisterObserver`

- Unregister method for *Observer Pattern*, takes an `Observer observer`, removes the specified `Observer` from the list while still maintaining immutability.

- `chooseMove`

- Processes the move for each player by calling `advance`, updates the observer for each player and declares whether the game is over or not.

- `build`

- Instantiates the new game state and observers list for the new `GameState`, returns *Observer Pattern*

iii. Getters

`getCurrentBoard`, `getObservers`

iv. Evaluation

Unlike in the previous section (*MyGameStateFactory*), we did not take much inspiration from the previously credited party as we felt this section was more intuitive and as such very little referring was done to their implementation. We did come across an impending error, where the framework issued to us imported an unused library feature called *NotNull* alongside *Nonnull*, which threw an error because of a missing library and the fact that *NotNull* was used instead of *Nonnull*. We believed for a while this was necessary, but after consulting a TA, we rectified the issue and removed the library entirely.

We would discuss possible future improvements; however, we believe there is not much to improve because the solution is already short, concise, modular and efficient. We believe our solution is in an optimal position and will represent an across-the-board solution like others due to the small magnitude of the problem in the first place.

3. AI

The second half of the project is reserved for an implementation of AI for *MrX* and the *Detectives* which must autonomously make moves for the player. '*ScotFish* ' (a play on words of the famous chess engine *'Stockfish'*), our implementation of this AI, is a scoring-based model for *MrX* **exclusively** that makes moves depending on a scorevalue assigned to *Mrx*'s moves. In this section, we will discuss our implementation approach and scoring system and will evaluate its efficiency, identifying any weaknesses and improvements. Below we showcase our AI:

i. Attributes

- `board`, (of type `Board`) holds the current status of the game board,
- `scoreMap`, (of type `Map<Integer, Integer>`) maps each node to a score value,
- `detectives`, (of type `ImmutableList<Detective>`) holds all detectives that take part in the game,
- `mrX`, (of type `Piece`) holds MrX's piece.

Scotland Yard Coursework Report

ii. Methods

- `initialiseScoreMap`

- initialises all the nodes with the initial score value of 0.

- `setPlayers`

- initialises the `detectives` and the `mrX` attributes.

- `getDetectiveLocations`

- returns the current location of all the detectives present in the game as immutable list of integers.

- `setScoreMap`

- sets the actual score values by calling helper methods.

`setScoreMap`'s helper methods:

- `setDetectiveAdjacentNodeScore(N, F)`
  – parameters: `N` of type `int`, score modifier;
  `F` of type `int`, modifying factor.
  - calls `setAdjacentNodeScore` and decreases the score by `N * 5` points for each detective location.
- `setAdjacentNodeScore(origin, N, F)`
  -parameters: `origin` of type `Integer`, the origin node in relation to witch the scores for the adjacent nodes is determined;
  `N` of type `int`, score modifier;
  `F` of type `int`, modifying factor.
  -creates a list `distanceOneLocations` of all the nodes of distance one and decreases the score by `N` points. Then for each node in `distanceOneLocations`, all adjacent nodes' scores are decreased by `N / F` points.

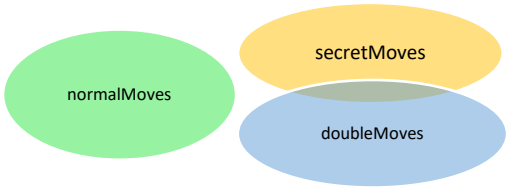
`setDetectiveAdjacentNodeScore` and `setAdjacentNodeScore` create a zone of decreased score values around the detective pieces. This results in MrX 'running away' when the other pieces get close to him and also ensures that picking a move that places him in proximity of the detectives is less likely. During play-testing we have determined that a value of `N` = 100 and `F` = 2 result in desirable behavior.

- `setFerryNodeScore(N)`
  - parameter: `N` of type `int`, score modifier.
  - increases the score of the nodes that have a ferry connection by `N` points if MrX has at least one 'secret' ticket. If MrX is already standing on a ferry node, then that node's score is reduced by `N` points to avoid the case of a double move to the same location.

`setFerryNodeScore` makes ferry movements more likely to be picked, as we found they often are harder to track by detectives. We have observed that a value of `N` = 125 gives the best behavior. The score of a node that has distance 2 to a detective but is a ferry node has a value of 25, which is slightly higher than initial value 0, sometimes makes for unexpected and risky moves by the AI.

- `getBestMove`

- picks a move out of the list returned by `getHighestValueMoves`. It separates the moves into three lists: `normalMoves`, `secretMove`, and `doubleMoves`. As the name implies, `secretMoves` holds all the moves that use a 'secret' ticket and `doubleMoves` holds all the double moves. `normalMoves` holds all the moves that are do not meet the conditions of the other lists. As such `secretMoves` and `doubleMoves` are not mutually exclusive, so a double move that uses at least one 'secret' ticket is present in both lists.



The method picks at random a move from one of the three lists. The chance of a normal move to be picked is 80%, whereas the chance for one of the special moves to be picked is 20% (10% for a double move and 10% percent for a secret move). This makes sure that MrX does not use his special tickets too quickly while retaining the element of surprise when he does.

`GetBestMove`'s helper method:

- `getHighestValueMoves`
  - returns all moves that have the same highest score as an immutable list. It iterates through `scoreMap` twice; once to determine the maximum score of the move's destination node, and once more to add the moves that have the destination node's score equal to the maximum score to the return list.

vii. Evaluation

Given that there was no instructional guide to implementing this section of the project, development was done intuitively and hence we came across more issues. One notable task was implementing a way of retrieving a list of players that could be used for future iteration, because we did not have access to a `Player` list but a `Piece` list due to the framework supplied. Hence, we were forced to code a 'work-around' that provided much of the functionality that was already implemented in *cw-model*, hence this issue could have been avoided and we could have recycled code if the framework was optimised to provide a `Player` list.

There are features that we would have liked to have implemented but unfortunately we couldn't because of time constraints. An implementation of Dijkstra's Algorithm to compute the optimal paths that the detectives would take to reach MrX and decrease the score of the nodes along the paths was planned. We would have also liked to implement a look ahead algorithm that would use the optimal paths to return the best line of MrX moves, like chess engines.

We also would have liked to have implemented an automatic testing framework, but time was a factor so we did not.

To summarise our development into the AI feature of *Scotland Yard*, we implemented a fully working AI that can make decisive, correct decisions of which move to make during gameplay. We have played many games against our AI even with move tracing on to give us an advantage against *MrX* and there were games that we in fact lost, due to *MrX* making the most optimal move away from detectives based on our scoring method.