Scotland Yard Project

Our main goal was to achieve a working digital version of the popular board game Scotland Yard. After carefully reading the documentation within the interfaces and the instructions given on this coursework, we managed to successfully implement both cw-model and cw-ai.

CW - MODEL:

We started by verifying the building game state of all possible errors and forbidden states. After we implemented the GameState interface with all its methods, the most important and complex of all being **advance()** which would receive a move as parameter and return a whole new game state.

Within the project we used specific **OOP techniques**, material which we found in the lectures:

1. **Lambdas and Streams** to make the code more readable and cleaner, within the **getPlayers()** method.

2. **Visitor Pattern** to get both the **destination** (for **makeSingleMove()** method) and **destination2** (for **makeDoubleMove()** method).

3. **Anonymous Inner Class** to implement and instantiate a class at the same time, within **MyGameStateFactory** class at **getPlayerTickets()** method to implement **TicketBoard()** interface, at **advance()** method to obtain the list of destinations and within **MyModelFactory** class to implement the **Model** interface.

The first problem we encountered was related to the mutually exclusiveness of **getWinner()** and **getAvailableMoves()** methods within **MyGameStateFactory** class. After a little bit of documentation and tests we managed to overcome the problem by testing if the **winner** list was empty every time **getAvailableMoves()** was called.

The second problem we encountered was related to the **Visitor Pattern** concept. This pushed us to read and study everything we found on the internet and in the lectures related to this. After a few discussions and tests we understood how it works and managed to adapt it to our code.

The third and last problem we faced was the concept of **Anonymous Inner Class** which we needed to fully understand in order to finish the **MyModelFactory** class. We understood its efficiency and readability in no time and managed to pass all the tests.

CW - AI:

theCrane  vs. **smartDetectives** 

theCrane - mrX's AI

In the beginning we started to think about how to make the **scoring function** (which helped us to decide the best possible next move). We have taken in consideration several different variables which we considered the most important ones in choosing a next move while playing the game:

1. **The distance between mrX's next location and the detectives =>** we made use of Dijkstra's algorithm for finding the shortest path between two nodes (which was actually a Breadth First Search (BFS) using the weight 1 on an edge. We also made some basic assertions for our dijkstra to gain some confidence and be sure that the algorithm is implemented correctly (see **OurDijkstra** class and **TestDijkstra** class). We used a variable called **movementRatio** which gave a **very big score** if **a detective was unable to catch mrX** (it was impossible because he didn't have enough tickets to reach that location) or a score which was calculated using the **division between the distance between the players and the detective's number of tickets**.
2. The second factor taken in the consideration was the number of stations that a node is connected with. We calculated a **station score** for each future move. For example a node which is connected with a bus and a taxi lane is better than a node which is connected with just a taxi lane. Moreover, **we used different weights** for the different types of stations (a train station weights a lot more than a taxi station because with the train you have the opportunity to **travel as far as possible** and the train stations are **more rarely** than taxi stations and you may miss the opportunity to use your train tickets). We chose the weights based on the % of each type of station that appears on the map. We didn't neglect the fact that calculating the station scores should be based on mrX's remaining tickets.

3. The third variable is straightforward and was added in the late part of developing mrX's AI, after doing a lot of testing. We needed to be sure that the **next location would be as far as possible from the previous one**, because in case that is a round when mrX shows up it means that the detectives will start to hunt him.
4. The fourth variable is called **danger** which is considered to be an improvement to the first variable. In case that **a detective is 1 position away from mrX** then the scoring function should **return a very bad score**.

Each of these variables weights differently on the final score which is returned after calling this method.

PlayerInfo class & OurNewBoard class

We created a class called **PlayerInfo** (which is similar to Player class already made in cw-model) for storing all the data we have about a player and query information by implementing some useful methods (examples: **getPiece, changeLocation** -> used when computing future moves).

OurNewBoard is a class that we use for updating the board when computing mrX's future moves. We keep all the information we need in it: a list of players (a list of PlayerInfo objects), setup, graph. This class has implemented in it the *most important method which is similar to the one in cw-model: **getAvailableMoves.***

How we calculate mrX's next move ?

We built a **game tree** and used the **MiniMax algorithm** for looking into the **future** of the game with the purpose of finding the best possible move for mrX. We started by looking just 3 moves into the future, but the algorithm was moving too slow. We optimized it by adding **alpha-beta pruning** in our MiniMax algorithm. In our algorithm very useful classes for simulation where PlayerInfo & OurNewBoard which were mentioned above.

We felt that we can do something more to make our AI **more powerful**. In an **extreme case** it was vulnerable if mrX would **remain without time to move** (too many possible cases, too many operations to compute). As a solution to this, we've done some research on the internet and we made use of a method from the System class, which helped us getting the current time in milliseconds. By knowing that information, we could know when is the time to **force the algorithm** to return a move. After this, we

forced the algorithm to compute **the biggest number of anticipated moves** (for example: it is round 1, with a game with 24 rounds, the algorithm would calculate 24 future positions in depth).

smartDetectives - detectives' AI

After finishing mrX's AI, we started with developing **smartDetectives**. They are not as complex as theCrane but they are pretty good and maybe with some luck they can catch mrX (depending on **different factors** like: starting position, available tickets, when mrX's shows up, the number of rounds).

Our algorithm is based on 2 simple things:

- If mrX reveals his position the detectives should get as close as possible to that location **(Dijkstra's algorithm).**
- In the case that detectives don't know mrX location they should choose a location with a **good score** (more stations) -> because staying on good nodes gives you the opportunity to **travel easier** when mrX reveals again.

Some reflections

theCrane (mrX's AI) can be vulnerable: he might not be able to compute enough operations and if he remains without time it is possible that he would not return the best move, because he did not get the chance to analyze all the outcomes. One way to improve it is to decide a strategy for how to use his special tickets (secret and x2) more efficiently. He should keep track if he is revealing his position, trying to leave the area as soon as possible after that.

smartDetectives can be improved in different ways. Building a game tree with even only 3 depths will help detectives to predict mrX's behavior. Another way of making them better is to analyze mrX's used tickets after revealing himself and follow a possible path that he could have taken.

After finishing the project, we are glad to say that we both improved our object oriented programming skills, gained confidence and learnt more about how it is like to work within a team, to contribute to the same project and at the same time learning the basics of Git.