Scotland Yard Project Report

The cw-model passes all tests and both *MyGameStateFactory* and *MyModelFactory* are fully implanted and functional, below is a summary of the code.

- A set of getters implementing the *GameState* **interface** itself, and its methods. The method *getPlayerTickets()* features a **lambda** to define the behaviour of *getCount()* from the *TicketBoard* interface without explicitly creating a class; this method is there to obtain the amount of tickets from the type of ticket.

- The method *getAvailableMoves()* is called every time a new *GameState* is created to obtain the set of moves available to the players, checking whether a winner has been declared. This is followed by some helper functions to utilise returning the available single or double moves, considering whether the move is valid.

- The next set of methods are called by the **constructor**, in the variable *this.winner* every time a new *GameState* is created to calculate whether a winner has been found, the 4 different ways for the 2 sides to win have their own specific methods. It has a neat structure in order to make feel easier to read.

- The utility helper functions are used to gather specific attributes or to learn specific properties from the code that normally cannot be obtained. Within the *updateLog()* method is the **visitor pattern** in practice used when updating MrX's log based on the move he has made.

- A **named inner class**, *GetDestination*, is implemented as a **getter** as there was a lack of an easy way to obtain the destination from a move. It **implements the object Visitor** as the Move can be either Single or Double move.

- The method *advance()* distinguishes MrX and detectives, and makes two distinct action updating the *GameState*. Stuck players are also removed by using conditional method *removeIf()*. Here, a **method reference** is used which avoids the repetition of a **lambda expression**, where *this::isStuck* references the *piecesNotMoved* variable as the parameter of *isStuck()*.

- The *build()* method creates the *GameState* with *testBuild()*, **asserting** certain properties that must hold to validate the game.

- The **observer pattern** is used in *MyModelFactory* , in order to broadcast essential information (e.g. Game over) to every observer.

The cw-ai is a finished model of an algorithm to be used for MrX when deciding moves. Here's a summary about the module.

- Class *MyAi* implements the *AI* interface, picking the moves among the moves generated from *getAvailableMoves().*

- The outer class *ScoreMap is* created when given the board (*gameState*), it contains the list of *Score* objects, which contains (maps) every possible moves available for MrX with its own evaluated score, which is created from **the inner class.**

- The class *ScoreMap* is also has the authority to evaluate the scores based on its move, directed by *score()* which takes into account about :
  I) The distance between the detectives and MrX
  II) To evaluate the nodes in terms of having variety of selection of transport
  III) Evaluating the ticket's value
  IV) Evaluate the current situation, finding out how much MrX is on the cusp
  V) Changing the actions depending on how dangerous he is currently

- The outer class *Dijkstra* is created to calculate the distance between a detective and all the nodes by implementing **the Dijkstra algorithm**. We have referenced the outer library *PriorityQueue* in order to facilitate the order of adjusting the distance.

- To avoid manipulating or looking inside the inner data, the classes are protected via **encapsulation** and marking it as **immutable**, only letting the wanted output to be returned with a **getter** from outside.

- Testing functions of our program, which has a formal structure, based on **abstract** class of basic test function, and **inheritance** of the class in order to test in various phases.

To meet the most fundamental rule: staying away from detectives, counting distance was essential and we have gone through it via Dijkstra algorithm.

Since the graph provided had its weights in terms of transport, not the distance between the nodes, and it had Ferry which detectives could not use, we had to reconstruct the map structure, with 2D integer array with assigning 1 if the value is present. Also, we had made an inner class *Node* to store the node index and the distance.

In the beginning, we have gone through two nested for loops to update the shortest path from the source to destination. But we encountered the problem that we are getting repetitive calculation, so we have referenced the outer library priority queue in order to optimize the algorithm. We also had to implement the interface *Comparable<T>* , and **override** *compareTo()* method in order to compare our object *Node* via its distance, when making a priority. Furthermore, the algorithm was optimised to additionally ignore detectives who were stuck or did not have the required tickets for a move. We used **Iterator** for extracting a value from a set.

The score of each move is then evaluated based on the distance, the ticket used, and the node the AI is ending up at.

First, the score is adjusted based on the distance; originally, a *meanDistance* was calculated adjust the score, but this was theorised to lead to an issue where if 3 players were far away, and 1 was close, then the threat of the close one would not be big enough, this led to an optimisation and a change to calculating the mean danger of the given location, instead, an average danger is assigned from each detective and a mean is calculated from that, detectives too far away to matter (those who are further away than the rounds remaining) would be ignored. A literal adjustment was made to the score of a location depending on the imminent proximity to a detective, where locations right next to a detective were basically grounded in score to never be considered.

Secondly, the score is adjusted based on the type of transportation. Since there are two move types, Single and Double, and two emergency state (mrX is in emergency or not), so there are four basic situations. Since some tickets are more valuable in certain situations, we had to clarify these four cases and assign score differently. This was done by using a **functional visitor** differentiating two move types and **conditional statement** in each method. It also uses 2 **method references** for both parameters of the functional visitor when evaluating the ticket itself.

Thirdly, the AI evaluates the node it is ending up at, scoring nodes which have more variation higher than those which do not. It prioritises stations with more transport when it is exposed or in imminent danger.

Finally, the score is returned, and *bestMove()* picks the move with the highest score.

The functions made were tested by the classes in test folder, and it was in a structure that the *TestModel* **abstract class** provides the basic tool, including **overloading** *test()* function to receive various types of arguments, for testing the function, and we could simply **inherit** the class and test the functions. However, there was a limitation since we were not using a framework for the test, since we had made most of our functions private (**Encapsulation**), there was a restriction that we could not specifically test the inner methods since most of the data and methods were marked as private.

Of course, the AI is not perfect, from some play testing, it has been observed that the AI performs well in the early-game when detectives are quite a few distance away but begins to make errors in the mid-game while in the late-game it loses to human intuitiveness and lack of consideration of it. (e.g., if in imminent danger in all possible moves, consider which one the players will most likely not pick). Also, since we have not implemented the min max – like algorithm nor the game tree, the AI just evaluates from the current state, not anticipating opponents' move.

Overall, this AI would be most likely be suitable as a medium difficulty AI, it is not too hard for the players to catch the AI, but at the same time it is not too easy to catch it out in the early game. Furthermore, the AI does miss some obvious logical conclusions what move it should make as a result of being unable to consider the situation from the detective's perspective. Via this project, we can glad to state we had experienced the power of Object-Oriented Programming.