**Scotland Yard Coursework**

Within the MyGameState factory file, we implemented the GameState interface using the class MyGameState. To accomplish this, we first implemented the build() method, which included validation such as checking for null references and correctly initializing the game state. We ensured that the game state was returned with Mr. X as the first player to move. After this, we implemented the basic interface getter methods to return values of attributes and other necessary information.

Next, we focused on move generation and updating the game state. To start, we fully implemented the getAvailableMoves() method, which creates an immutable set of all possible moves given the current game state. To accomplish this, we used helper functions for single and double moves. In these helper functions, we checked the ValueGraph to determine valid moves based on other player locations and the quantity of tickets available.

We then moved on to implementing the advance() method, which is vital for processing a move and returning a new game state. In this method, we utilised the Visitor design pattern to update the game state. This involved handling single and double moves separately. In each of these methods, we first validated whether the chosen move was legal for the player making it. Once validation was complete, we performed the move by updating the player's location, using the corresponding ticket, and modifying the relevant player attributes before returning a new game state.

**Additional considerations included:**

- If Mr. X was moving, we needed to update the move log.
- Ensuring the remaining attribute never contained the wrong players.
- Checking whether the move occurred in a Mr. X reveal or hide round when adding it to the log.

Next, we implemented the getWinner() method, which initially checks if the game is over.

**To accomplish this, we considered several possible outcomes:**

- If any detective has captured Mr. X.
- If Mr. X has no valid moves remaining.
- If all detectives are stuck.
- If Mr. X has used all his moves.

Finally, to complete the Scotland Yard closed-ended section and finalise the game, we implemented the Observer design pattern in MyModelFactory. Here, we completed the Model interface to create game model instances. Within this, we built the game state, registered and unregistered observers, and checked whether the game was over when moves were made.

After completing all of this, we successfully passed all tests and proceeded to the open-ended AI section of the coursework.

The first challenge was to obtain Mr. X's current location, determine where he would move next, and identify which ticket he would use. Since we were using the non downcasting version, we utilised the Visitor design pattern to achieve this.

The first part of the AI we implemented was the scoring function, which rated all possible locations Mr. X could move to based on a formula. This helped determine the best move given the available options. To achieve this, we initially needed to calculate the shortest distance between each detective and Mr. X's potential destination.

To do this, we first retrieved the location of every detective and the possible move locations for Mr. X using the Visitor design pattern. We then created a helper function to find the shortest distance, for which we used a breadth-first search (BFS) approach. We chose BFS because every edge in the graph had equal weight. To implement this, we used the ValueGraph and our own queue.

Another key factor we considered was freedom after the move that is, the number of available moves Mr. X would have after reaching a destination. To evaluate this, we created another helper function.

**In the score calculation function, we applied the following logic:**

We added the square root of each detective's distance to the destination. This introduced a heavier bias against locations near detectives, steering Mr. X away from them.

We included the freedom factor, dividing it by 25 to ensure it didn’t have an overwhelming influence on the score.

Finally, we implemented a check to determine whether a detective was adjacent to the destination. If so, we assigned it the minimum possible score to prevent Mr. X from choosing it.

This approach allowed us to intelligently guide Mr. X’s moves, maximising his survival while avoiding close encounters with detectives.

To start our game tree, we implemented a simple minimax tree with a depth of 2. To achieve this, we created a class called MinMaxTree, within which we defined a Node class for each node in the tree. Each node had attributes for its value, children, and whether it was a findMax node. Each layer of the tree was designated as either a findMax or not findMax layer. Depending on the layer type, the move with the highest possible score outcome was selected.

The computeMinMax method recursively evaluated all node values in the game tree, starting from the root nodes and working down to the leaves.

After implementing this, we introduced alpha-beta pruning, as the AI was taking an excessive amount of time to calculate the best move, even with just three detectives. Alpha-beta pruning significantly reduced computation times by pruning irrelevant branches, thereby making the AI much more efficient.

**To implement this, we used two values:**

Alpha represents the best value for Mr. X.

Beta represents the best value for the detectives.

If beta was less than or equal to alpha, the rest of the branch was pruned, as it no longer needed to be explored.

Once we completed this, we had developed a two-move look-ahead AI that could handle up to three detectives within a reasonable timeframe. However, for four or more detectives, computation times became impractical. As a result, we used a one-move look-ahead function in those cases.

**Strengths of the Approach**

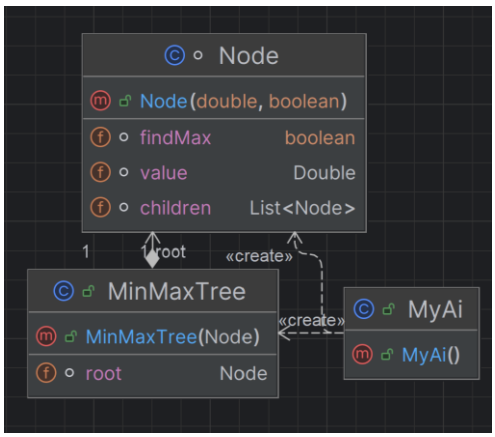
Choosing breadth-first search over Dijkstra's algorithm was preferable, as the graph was unweighted, and breadth-first search was simpler to implement.

Implementing alpha-beta pruning significantly reduced calculation times, improving the playing experience.

**Weaknesses of the Approach**

Even with pruning, minimax remained impractical for four or more detectives. To address this, we should have optimised our code further.

We were limited to a depth of 2 for our trees. A greater depth would have resulted in a stronger Mr. X player.




**Here are some resources we utilised to better learn and write out code!**

https://www.youtube.com/watch?v=l-hh51ncgDI&t=520s

https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/