Scotland Yard Coursework Report

Lucy Randewich (xm20246), Callum Ward (ke19697)

Overview:
We have completed the cw-model part with all tests passing, and also have a working AI for mrX; an AI for detectives has not been attempted.
Due to circumstances, we found it better to work on different parts of the coursework individually, such that Lucy worked mainly on the model and Callum worked mostly on the AI.

CW-Model:
The class MyGameStateFactory begins with three helper methods used by the constructor of myGameState, “makeSingleMoves”, “MakeDoubleMoves” and “determineWinner”. The myGameState constructor first completes all validity checks for passing the first set of tests, then initialises its attributes using the parameters passed to it. It then does some other initialisation tasks such as checking for a winner, and determining which players are remaining in the round. The subsequent getters and setters have been implemented trivially, and the advance function makes use of the visitor pattern for single and double moves.
MyModelFactory was completed by implementing Model as an inner class, with all its abstract methods implemented including implementation of the observer pattern.

CW-AI:

Overview:
For the AI we chose to implement a recursively generated Minimax game tree with Alpha-Beta pruning. This was done using a main GameTree class in combination with helper classes to store each nodes data and score them. Starting with the AI interface, the onStart method is overridden to import shortest path data from a text file to a 2d array. The code for this has been left in the ShortestPath class.

GameTree:
An instance of this class is created in the pickMove method, the constructor takes the Board, shortest paths array and the number of rounds to look ahead. The game tree starts by creating a GameState instance in order to get access to the advance method. From this state you can simulate a game by creating deep clones then advancing and storing an instance per node. Originally each node stored a list of the previous moves made and used that to advance the root state, however the time to do this increases exponentially compared to storing a state in each node. Pruning happens on every layer by choosing the highest/lowest scoring child nodes of a node depending on whether it’s MrX’s turn or a detectives. Once the desired depth or a leaf node has been met, that branch of the tree stops and sums the scores of its child nodes to its own. Once completed the highest scoring node from the root’s child nodes is picked as the best move.

Node:
A node instance represents a move made in the game tree, allowing every move to be scored individually. To do this each node stores position data on MrX and any detective that just made a move. Most importantly the current GameState is stored so that it can be cloned and advanced for the next move. I cloned a deep cloning library (01/04/2021 https://github.com/kostaskougios/cloning.git) as I didn't have access to the source code of the GameState class to implement a deep cloning method. This was necessary so advancing individual nodes didn't just alter one instance.

ScoreGen:
The ScoreGen class has one instance used throughout the lifetime of the pickMove call, its constructor takes the shortest path array. The class contains one method per scoring technique, with only 4 being used currently. Win/Losses result in the biggest effect to the score, next being how many moves MrX can make if it's his turn. Shortest distance to detectives is scored not linearly to prioritise escaping close detectives. Distance from a ferry is also scored not linearly but at a lower degree for the same reason.

ShortestPath:
This class isn't used in the final implementation however I left the code in so you can see how the text file was made. I used a capped recursive approach to find shortest paths between every position on the board. Starting by finding the longest path between any 2 nodes and capping the recursion depth to that (10), so that it would run quicker for testing. The text file contains data in this order, start position, final position and shortest number of moves required, each separated by a space.

Improvements:
- Generating the tree more efficiently so a greater depth can be used to predict better moves.
- Lower memory usage, this could have been done by dereferencing the GameState once its child nodes had been generated.
- Creating a detective AI by changing a few parameters of the current model.
- Adding more scoring techniques to give MrX a specific play style, for example prioritising double moves for when detectives are close.
- Adding a variable depth function to allow for greater depth when possible