Programming and Algorithms II - Scotland Yard Report
Finn Hobson and Harvey Robinson

CW-MODEL

For our CW-MODEL section of the Scotland Yard Project we have managed to pass all 114 tests and create a playable implementation of the Scotland Yard board game.

After playing the physical version of Scotland Yard and understanding the rules, we started by creating the constructor of our ScotlandYardModel class. Our constructor creates a few essential fields such as the game graph, a list of rounds and the list of ScotlandYardPlayer objects as well as checking that the players and the game have been initialised correctly.

We then went on to implementing the view methods - the behaviour specified in the ScotlandYardView interface that our model implements. Most of these methods are getter methods that return an attribute or information about the current game. For the methods that return a collection, we had to make sure that these collections are immutable so that other classes cannot break the game by adding or removing elements.

Next, we began implementing the methods that allow each player to make their move every round. We created a method called validMoves which takes a ScotlandYardPlayer as a parameter (the player whose turn it is to make a move) and returns a Set<Move> of the possible moves they could make. To generate this set of moves, the method checks if each node is free to move to and if the player has the necessary tickets. This method is nearly 50 lines long so it may have been possible to increase readability by putting each of these checks into individual methods and getting validMoves to call these methods to generate the set of moves. The startRotate method is called at the beginning of every round and the accept method updates the status of the current player and the game based on what move is chosen and then rotates to the next player, allowing them to make their move.

When a player selects a move, it can either be a single move, a double move (only for Mr X) or a pass (only if no move is possible for a detective). Move is an abstract class meaning it cannot be instantiated. The sub-classes TicketMove, DoubleMove and PassMove extend Move so they inherit all of the features of the Move parent class. We have used a visitor design pattern to update the attributes of the current player depending whether the selected move is an instance of PassMove, TicketMove or DoubleMove. Inside our ScotlandYardModel class, we created a nested class called Visitor that implements MoveVisitor - a class given to us in the skeleton code that knows how to visit each type of Move object. Each visit method within Visitor updates the current player's location and ticket count and if it is Mr X's turn, the current round and Mr X's last known location are also updated. Inside our accept method, we instantiate a Visitor object

and then ask the selected move to accept our visitor and dynamic dispatch on the move sub-type.

Once we had implemented methods to add and remove spectators, we created methods to notify all of the spectators about events during the game such as when a move is made, when a round is complete or when the game is over.

CW-AI

For the CW-AI section of the Scotland Yard Project, we have managed to create a working AI for Mr X that selects the best move from a list of valid moves based on a scoring system. We have created a method called score that takes a view of the game, Mr X's current location and a list of valid moves, and returns the best possible move. Within the makeMove method of the MyPlayer class, our code returns this best move back to the game by calling the accept method of the parameter object callback.

The score method that we have implemented cycles through each move in the list of valid moves and gives each move a score based on several factors. These factors include:

1. The number of paths coming from the new location. We increase the score of the move depending on how many exit routes there are from its destination node. This makes it less likely that Mr X will be cornered or get stuck.

2. Proximity to the detectives. For every detective one node away from the move's destination we subtract fifty from the score. For every detective two nodes away from the move's destination we subtract 20 from the score. This moves Mr X away from the detectives.

3. Saving double and secret tickets for later in the game. We subtract a large amount from the score of double and secret moves at the beginning of the game and make this amount smaller and smaller as the round number increases. Double and secret moves will usually have the best score so this makes Mr X save these tickets for the later stages of the game when he is more likely to need them.

4. Using secret tickets tactically. We prevent Mr X from using a Secret ticket on a reveal round by subtracting a large amount from the score in these situations. This prevents Mr X from using a secret ticket to hide his transport type but then having his location revealed anyway.

Once every move in the list of valid moves has been given a score, the move with the best score is returned to the game. Similar to our CW-MODEL, we have used a visitor design pattern to visit each Move object to update the destination of the current move and check if a secret ticket is being used.