Scotland Yard Report

Summary of getting all Valid Moves

We created 4 main functions for getting all possible valid moves.
These were:

- *getMoves*;
- *getDoubleMoves*;
- *getMrXValidMoves*; and,
- *getDetectiveValidMoves*.

The getMoves and getDoubleMoves would be used as helper functions to produce a final set of moves within the getMrXValidMoves and getDetectiveValidMoves functions. getMoves returns a set of Moves for a particular Ticket and Transport type. It iterates through all the edges of the current node of a detective and returns moves that correspond to the same Ticket and Transport type given to its arguments. getDoubleMoves was a little more complicated, we had to nest a for-loop inside of a for-loop to look at edges of edges, and as this function was only for MrX we had to account for a new Transport type (Ferry) and a new Ticket type (Secret). We also had to consider all possible combinations of 2 ticket types. We also had to make sure that all detectives could legitimately make the move by considering the tickets they currently have. We did this by using data structures such as Maps. This is where Java has advantage over other programming languages such as C, as it is a lot easier for you to use these data structures within your code. We did this by accessing the Map that was given to each ScotlandYardPlayer upon instantiation, and for each edge data checking they had the correct amount of tickets.

The getMrXValidMoves and getDetectiveValidMoves functions were used to create the final set of moves for each type of player (MrX, Detective). Within these functions we got an idea of how to use classes within the ScotlandYardModel package, and gave us a good understanding of concepts such as encapsulation with packages.

Accept Method

In order to execute the data returned by the *makeMove* method, we had to implement the *Consumer* interface for type *Move* and produce an *accept* method.

In *accept* we needed to validate the move, and then execute the move if it was valid. To check if the move was valid we first checked it wasn't *null*. If this was the case we then fetched all the valid moves the player could make using *getDetectiveValidMoves* or *getMrXValidMoves*, depending on the player type. If the passed move was not in the set return my these methods then the passed move is invalid and an *IllegalArgumentException* is thrown.

Should the move be valid we then needed to execute the move, and update the game accordingly. Here there were three different techniques which would have to be used, depending upon whether the move was a *TicketMove, DoubleMove* or *PassMove*. We created a separate private method for each case.

If the move is a *TicketMove* we would update the player's location and remove one ticket, of the type used to make the move. Here it is possible for the move to be made by either Mr X or

a detective. If a detective made the move then their ticket would be given to Mr X. Otherwise, if Mr X made the move, this would signal the start of a new round and spectators would need to be informed. We need to tell spectators that a move has been made, again there are two possible situations here, either Mr X made the move and it is not a reveal round, or not. If it is not a reveal round for Mr X then the move we tell specators has been made is given with Mr X's last known location as the destination. In all other cases we passed the move with the players new location.

If the move is a *DoubleMove* we know Mr X has made the move and so will have to alert spectators of the start of two new rounds and of two moves being made. We stated by removing a double ticket from Mr X and updating his location to the final location of the move. There is an increased complication here over whether one of the rounds is a reveal round. We decided to parse the passed *DoubleMove* first by producing two *TicketMoves* which make up the passed move, and set the destinations of them depending upon whether the round the move is made on is a reveal round. We created a variable called *MrXSafe* to store Mr X's last known location while we parsed the new information. After parsing the move we returned *MrXLastLocation* to the value it had before parsing. We then altered spectators that a double move had been made, and then executed a *startRound* method. This method checks if it is a reveal round, and updates *MrXLastLocation* accordingly. Then removes the transport ticket that was used to make the move from Mr X, increments the round number and tells spectators that a new round has been started and then that a move has been made.

If the move is a *PassMove* then it can only have been made by a detective and thus all we need to do is tell the spectators that this move has been made.

Once the move has been executed then either the game is over, or the next player should be told to play. To check if the game is over we called our *isGameOver* method. If this returned to then we told the spectators that the game was over. Otherwise, the next decision depends upon who made the last move. If the last detective made the move, so if Mr X was due to make the next move, then a rotation has been completed and we altered spectators to this. Otherwise, the next player would need to be told to make their move. We did this by fetching the valid moves for this player, as done previously, and then calling *makeMove* on the player.

isGameOver

There are four situations which mean the game is over:

- Mr X has been captured;
- Mr X cannot move;
- All the detectives cannot move; or
- Mr X has escaped.

We created a separate method to check each possibility.

To check if Mr X has been captured we check the locations of all the detectives, if one of them is on the same location as Mr X then he has been captured and this method returns true. Otherwise, it returns false.

To check if Mr X cannot move we fetched al the valid moves he could make using *getMrXValidMoves*. If the returned set is empty then Mr X cannot move, hence this method returns true.

To check if the detectives are all stuck we fetched all the valid moves of each detective, in turn, if we found one which had a valid move which was not a *PassMove* then our method

returned false. If the method managed to check every detective without finding one who can make a move over than a *PassMove* it returns true, as no detective can move.

For Mr X to have escaped we need him not to have been captured, and for there to be no roudns left. Having previously established that Mr X had not been captured, we just needed to check if there are any rounds left. We did this by comparing the value of *currentRound*, if it is less than the number of elements in *rounds* then there are still more rounds to play and thus the game is not over.

Our *isGameOver* method runs through each of these methods in turn. As some are the last three should only checked at the end of the round we added a condition that the current player must be Mr X for these methods to be run. If any of these methods returns true then *isGameOver* returns true, only if all return false does *isGameOver* return false.

Pruning

We run the pruning method first to remove moves that would never be the best, and therefore reducing the processing time to score all these methods. This allows for a more complex scoring scheme.

First we pruned out expensive moves, these being moves which could be achieved with a more plentiful ticket supply. So if we could move to the same location using either a bus or taxi ticket, but had more taxi tickets then the bus move would be removed from our list of possible moves. We achieved this by creating a map between possible locations and a collection of the different transports which can be used to get to that location. We fill this map by running through all possible moves and reading what ticket type they require. We then check every location in the map and create a new move using the most plentiful ticket type which can get to there, and adding this move to a list of moves which is returned by the method.

Then we pruned out double moves which could be achieved by a single move. We did this by first creating a list of all locations which can be reached by a single *TicketMove* and then checking if the final destination of each *DoubleMove* was in this list, if it was then that *DoubleMove* was removed from consideration.

We then decided to implement some tactical pruning, which we felt would make Mr X play better. We chose that Mr X should make a double move whenever it was a reveal round, thus he would not be in the location that was revealed. Preferably the second move of this *DoubleMove* should use a secret ticket so it is harder to predict where he went. To prevent him running out of double tickets we have prevented him from making a double move unless it is a reveal round.

Scoring Scheme

We created a method called *movesToLocation* we takes a location as a parameter and then returns how far away each detective is from that location, in a map. This was done by checking all edges away from the given node, if a detective was on any of these nodes then they scored 1. If not all detectives have be located then the edges from each of these edges is checked, with the score incrementing by 1 each time. This continues until all detectives have been scored.

This methods forms the basis of our scoring scheme. Our first scoring scheme simply gave each move the score of the closest detective to the end location. This was then updated to

be the sum of all the detectives to that location. Then the highest scoring move would be found and chosen, if multiple moves had the same high score then a random number is generated and the corresponding move is chosen.

Improvements to AI

For our pruning method we feel the tactical choices could be improved. Both in terms of what choices are made and when they are made. We believe a risk assessment method could be good here. This method would assess the risk to Mr X in his current location, if the risk is low then we will pick a single move however if the risk is high he will pick a double move.

Our current scoring scheme has an issue whereby if one detective is very close to a location but the rest are very far away then the score will still by high, despite the risk. We think using a scaled score when instead of scoring the sum of how far each detective is from the location, and choosing the highest scoring, we could score 1 over how far each detective is from a location and then choosing the lowest scoring move.

Currently our AI only looks one move ahead and doesn't attempt to predict the moves of the detectives. We have developed a method called *possibleLocation* which returns a list of each location a detective can move to in one turn. We chose this to be a list, rather than a set, so it can hold duplicates and then we would count the frequency of each location and use this to establish the probability of a detective moving to that location. We think implementing this with a tree structure to find all the possible moves the detectives can make in one, two or more rounds and then giving each of Mr X's moves a risk value which can be incorporated into the score. By looking multiple moves ahead we could choose the best move for a future round and then using dijkstra's algorithm to find the best path to it. This can then be updated each round.

Object Oriented Concepts

In Java one of the main concepts is creating classes with particular methods and attributes. One advantage of this is that you can design a class separate to any other class, and you can specialise the class to complete certain tasks. This makes it a lot easier to write compact and efficient code. We saw first hand how this can be very useful when working on ScotlandYardModel, and how we used the classes within the package such as ScotlandYardPlayer to tie many attributes of a player such as tickets, colour and location to one object which made it a lot quicker and easier to access these.

We also learnt about concepts such as Inheritance, and how creating families of objects that all inherit from one superclass can be very beneficial, as you would want to create a relations between classes that are very similar. We saw this with Move, which was the super to TicketMove, DoubleMove and PassMove. All of these are a type of Move but differ with what they are used for. But as they all inherit from Move, we could create one Set<Move> instead of creating multiple Sets when returning our set of ValidMoves.

Another key concept of object-orientation are interfaces. Our ScotlandYardModel implements Consumer<Move> and ScotlandYardGame which extends ScotlandYardView. This was very helpful when designing our AI, as makeMove contained a ScotlandYardView parameter, and we could obtain all the required information from the view to then design the AI.

A concept that we used throughout our code was generics, this allowed us to use multiple data structures such as Maps, lists and sets that use and store objects that we specify.

Paired Programming

It was a very good experience to be able to collaborate with another person in order to complete this assignment. It gave us a good insight into the good practices in code development that would be used in the workplace, such as using git to exchange files. It was also helpful to discuss how we would tackle certain tasks and problems, as at some points both of us had different ideas but from it we were able to gain an appreciation of alternate ways to