The Scotland Yard Project Report

By Nuno Alberto and Abdulrahman Abdulaal

Part 1 of the coursework (CW-MODEL)

GetAvailableMoves Method: Returns all the available moves in the current game state. We used two helper methods in order to get this behaviour and keep our code structurally correct. MakeSingleMoves returns all the possible single moves for a specified player in the current game state and MakeDoubleMoves returns all the possible double ones. They are quite similar in the way that they go through all the possible destinations of the player (in the case of MakeSingleMoves, the adjacent nodes; in the case of MakeDoubleMoves, the nodes up to 2 edges away) and they try to find all the available means of transportation to get the player from its location to the destination.

Advance Method: Returns a new game state according to the move passed as argument. Generally speaking, first we use the tickets required to execute the move and then we place the player in its new position, i.e., the destination of the move. After these two steps we have two different scenarios: if the player moved is MrX, we need to update MrX's travel log; and if the player moved is a detective, we need to give the tickets used to MrX. We then check if there are any players left to play in the current round and we deal with the edge cases accordingly. Finally, we create a new game state with all this updated information and we return it.

Visitor Pattern: We used this pattern to facilitate the access to information from both types of Move (SingleMove and DoubleMove). We created a class called SpecVisitor that implements the Move.Visitor interface and stores the information about the move in its fields. We can then call the method visit from the Move class passing an Move.Visitor (instance of the SpecVisitor class) as an argument and we will get the necessary information about the move itself (single or double). This works because we make use of Double Dispatch and Overloading: in the first call it will dynamically call the right visit method from SingleMove or DoubleMove (Single Dispatch), then it will again call the right visit method (Overloading) from SpecVisitor (Single Dispatch).

Observer Pattern: We created the ModelFac class to implement both the Model and the Model.Observer interfaces. The trickier method is chooseMove, the other ones are quite straightforward. Inside the try block we advance the state of the game using the move passed as argument and then we notify all the observers that a move has been made if there is not a winner. If there is a winner, then we notify all the observers that the game is over. Inside the catch block we catch all the illegal argument exceptions and we notify all the observers that the game is over.

Part 2 of the coursework (CW-AI)

Scoring Method: We implemented a scoring method that receives the location of MrX and the detectives in a given game state and returns a score based on that. It takes into consideration at most 4 factors that vary in importance according to the game situation, but, at least roughly, they have this order of importance (most important first): the minimum distance to the detectives; the number of available nodes adjacent to MrX's location; the number of available nodes adjacent to the available adjacent nodes to MrX's location (only if we have double moves); and, finally, the average distance to all other detectives (all but the closest to MrX). To compute the distance from MrX to the detectives we implemented an algorithm called Breadth-first search which is a variant of Dijkstra's algorithm. We can use it because the graph is unweighted (every edge has a value of 1) and we used it because it is faster than Dijkstra's.

MiniMax Algorithm: We implemented this algorithm in order to choose the move that maximizes the minimum gain resulting from that move. Alongside with the game tree we constructed by advancing the game state accordingly, it allows us to have an AI that looks further than only one round ahead. We also used Alpha-Beta Pruning and short-circuiting to improve the runtime of our program. The initial call to MiniMax is with depth of 2 but we are actually computing with depth of 3 because when the initial call happens we are already one level down the tree and then the recursive calls happen until we reach depth of 0 (exclusively). Therefore, we are looking 3 moves ahead.

GameStateSubstitute Class: We created a new class that replaces Board.GameState in order to improve the runtime of our program and to have more flexibility in general. We only store what we absolutely need to advance the game and keep track of the state of the game. This way, we get a lightweight version of GameState but with the exact same behaviour. Most of the methods we copied from CW-MODEL because we wanted to have the same behaviour as before. We made several changes in the advance method because we wanted to mutate the current GameStateSubstitute instance instead of creating a new one every time we advanced the game.

Reflection: We are pleased with how far we have come because we played the game several times and we found out that the probability of MrX winning (with the travel log exposed every round and with us controlling the detectives) is around 35%. Under normal conditions we do not have access to MrX's location every round, so the winning percentage in these conditions will dramatically increase reaching close to 100% we believe. Nevertheless, we feel like we could make some improvements. We consider that a dynamic tree depth and width management would be beneficial because in the beginning of the game we might want to search with a lot of depth and a low width (reduced set of evaluated moves) but closer to the end of the game we might want to reduce the depth and evaluate more moves to make sure we do not overlook anything. We had to find a middle ground between depth and width but the ideal solution would have been a dynamic management of the game tree.