The Report of Scotland Yard

Group members: Ruoxin Chen and Shuaili Zhu

CW Model

We have successfully passed all the tests, and we introduced many other methods like argus-check for dealing with conditions which are not illegal, and update methods in advance. We also write many streams to make the code more efficient.

My Game state factory:

VisitorPattern:
From the interface of Move, we know that it accepts visitor. In this way, the runtime dynamic type of the move can be automatically obtained, and there is no need to manually judge whether the move is singlemove or doublemove.

singleMove: this method is for dealing with the single movement, which is the basis for doublemove, there is two judgement, one is for normal tickets, and the other one is for secret tickets.

doubleMove: the doubleMove is for mrx, in the method of doubleMove, there are mainly four cases for mrx, the two tickets are the same, the two tickets are not same, and the case that mrx uses one secret ticket or using two secret ticket.

advance: in the method of advance, we provide four sub-small methods for dealing with different parameters.
updateLocationAndTravelLog : for updating the detectives' location and travel log ,addTravelLog: update mrx's travel log, updateTickets: update tickets,updateAvailablePlayer: update the players who still can move.

The following are the details of how we implement these four methods.

1.**updateLocationAndTravelLog**: the method will have a judgement at first, If the move is not commenced by mrX, the detectives' location and travel log will be updated through the method updateLocation and TravelLog. If the move is commenced by mrX, it would use the method addTravelLog for mrx.

2.**addTravelLog:**
Determined whether there is a double move at the beginning through visitor pattern.
use isDouble, isReavel and is revealnext these three boolean variables to add travel log, whether the destination should be hidden or not.

3.**updateTickets:**
If the detectives use tickets, the number of tickets will be given to mrx through the method given, but if the tickets are used by mrx, the detectives won't get the tickets.

4.**updateAvailablePlayer:** in that method, it will update the available player, they should have at least one ticket in any of the transportation: TAXI, BUS or UNDERGROUND.

My Model Factory:

in the class of MyModelFactory, it uses the observer design pattern with the registeredObserver, uniregisteredObserver and chooseMove method

registeredObserver uses a HashSet for keeping the observers, and unregisteredObserver is for deleting the observers. The chooseMove method uses the enumeration class of Event which in the Observer interface gets an event after making the judgement, the onModelChanged will be called when the game state changes.

CW- ai

In this part, I choose to use the Dijkstra algorithm, and I successfully implement the Ai for detectives and mrx.

The following is the introduction of the ai for detectives and mrx.

AiForDetectives:

When I write ai for detectives,I mainly consider the freedom of the point, greater freedom means more options after moving. In most rounds, the position of the mrx is not visible, so the chances of being able to catch the mrx are greater when there is more freedom.
In the method of getBestMove, it is the specific process of making detectives choices, using adjacentNodes.size() to represent the degree of freedom of the point, and thus to make a comparative judgment.

Ai for mrx

MiniMaxForMrx:

this class uses the MiniMax algorithm for mrx to analyze which point is more beneficial.

minimax:
This is the MiniMax algorithm itself, the main purpose is to maximize mrx's benefit and minimize detectives' benefit. The method first checks the base case whether there is a winner or the depth equals zero. The method mainly contains two situations, One is mrx's turn, and the other one is detectives' turn. If it's mrx's turn, the method keeps track of maxEval and updates the alpha value, if the beta value is less than or equal to alpha, the leaf is cut off because detectives won't choose. If it's the detectives' turn, it keeps track of minEval and updates the beta value. If the beta value is less than or equal to the alpha value, the leaf will be cut off because mrx won't choose.

score:
The method of scoring is quite subjective, I found a relatively balanced method in terms of freedom and distance, I assigned different rates to different ranges of distances and then calculated the score through freedom+distance + rate.

applyMinMax:
This method is the entry method for mrx's ai, it starts by initialization the maxEval to negative infinity and the alpha and beta values to negative and positive infinity respectively, which are used in the tree structure. It then obtains the move for the current board state and iterates over them.
For each move, it applies the move to the board and then evaluates the resulting board state by recursively calling the MiniMax function with the new board state, a decreased depth and alpha beta values. If eval is greater than the current maxEval, or if it has the same value but has more adjacent nodes in the graph, it becomes the new best move. If the eval is the same as the current maxEval and the same number of adjacent nodes, it is added to a list of possible moves. Finally, the method returns a randomly selected move from the list.

pickMove:
This is the main method for picking a move, dijkstra will be initialized here, and the best move can get from applyminimax with a certain depth.

DijkstraScoreFactory:

syncBuild: this method uses the Singleton design pattern, which is also a lazy initialization, using the build method to initialize. It first checks whether Dijkstra is null, if it is, it enters a synchronized block to ensure only one thread at a time can access the section. Within the synchronized block, it checks again if Dijkstra is null, and if it is, it calls the build method to initialize it.

build: this method uses a factory design pattern, it builds a graph of the game board and calculates the shortest path between nodes using Dijkstra's algorithm. For each node in graph, it adds a evrtex to Dijkstra object through the addVertex method. Next, it iterates over each node and its destinations, then calculates the weight of the edge between them, finally the edge is added to the Dijkstra object using the addEdge method.

getWeight: I assigned different values to the three genres of transportation based on the scarcity, which is also the number of tickets provided by the system by default.

class Dijkstra:
use the constructor to initialize vertexList and edgeMatrix, it sets all edges to infinite weight(except for self-loops, which have weight 0). This allows the Dijkstra algorithm to correctly compute the shortest paths between all pairs of vertices in the graph. Besides, the method dijkstra finds the shortest path between v0 and vi. In the score function here, the method will determine which is the nearest point to mrx among all detectives.

Possible Improvement:

ai for detectives:
The detective's ai can then apply the Dijkstra algorithm on top of the degrees of freedom, it can be applied in the rounds when mrx's location is revealed. As a result, the movement of detectives may seem to be more intelligent.

ai for mrx
Like the getWeight method in the class DijkstraScoreFactory. I define the weight by scacity, but it is totally subjective, people may have distinct opinions, for instance, we can consider the tickets that we have left sothat the weight would be different for every detective.

In the MiniMaxAiForMrX class, the score method is also subjective. I supposed that when distance is low, freedom is not important, so I assign the highest rate to the lowest distance. Maybe there are better ways to define the score method.

In terms of depth, I tried to figure out a specific relation between the level and runtime, but the difference would be really big because the freedom is highly related, when the freedom is high, the calculation would be too complex.

Reflection:

In the first part, a lot of code at the beginning is less logical and less reusable, so to write dry code in java I have to be familiar with OO concept and the use of different design patterns.

In the ai part, I made a lot of changes which is really time-consuming, so it is especially important to structure the program before starting. Besides, I should also organize the logic in pseudo-code before writing the algorithm and then implement it in the programming language.

Contribution: The first part is done by both of us, and Ruoxin Chen modified the code by using more streams, and second part is written by Ruoxin Chen.