COMS10017 – SCOTLAND YARD PROJECT

Patrick Lee

**PART 1 – MODEL**

Outline

The first task was to build the ‘Game State’ and ‘Model’ Factory classes such that it would pass all the given tests to ensure they would act as foundations to the Scotland Yard game, essentially creating implementation for an object “game state” which would provide all details about a particular game state including the arrangement of pieces on the board and the moves available to them. This was linked with the “model” implementation which could register and notify any external observers about game events such as moves and if the game has ended.

Our implementation was completed successfully, passing all 83 of the given tests and making use of a variety of techniques and OOP concepts learned in the past few months.

Helper Functions

In addition to implementing overridden methods from the “GameState” interface, we developed several helper functions within the surrounding “MyGameStateFactory” class to break down some of the more complex problems and reduce the size of the “MyGameState” class and its constructor. Helper functions were created for the following specific tasks:

* Given a list of detectives, determines if they are all distinct and valid. Returns Boolean.
* Given a location and list of detectives, determines if the location is occupied by a detective. Returns Boolean.
* Given a list of Player objects, returns an Immutable set containing the each Piece object linked to each of the Player objects.
* Given a Move object, uses the Visitor Pattern to return an Immutable Set of the destination(s) reached/passed through in that move.
* Given the MrX Log, a GameSetup object and a Move object, returns a new MrX Log (As an Immutable List of Log Entries) which contains an entry for the most recent move made my MrX.
* Given a Player and Move object, returns the same Player object but updated so that their location and tickets reflect the move they have just made.
* Given a MrX Player object and list of detectives, returns an Immutable Set containing all detective Piece objects and the MrX Piece object.
* Given a list of Player objects, list of detectives and a GameSetup object, determines if the the given players can move or not. Returns Boolean.
* Given a GameSetup object, list of detectives, Player object and Integer representing a location value, returns an Immutable Set of all Single Moves the Player can make.
* Given a GameSetup object, list of detectives, Player object and Immutable Set of single Moves, returns an Immutable Set of all Double Moves the Player can make.

OOP Techniques

* Classes, objects, attributes, methods and encapsulation – Attributes of objects instantiated from the “MyModel” and “MyGameState” are initialised in the constructors and only accessible via the getters and setters which we implemented based on the interfaces for each class, thereby ensuring attributes can only be modified and retrieved via specific methods.

Evaluation

**PART 2 - AI**

Scoring Function

We implement a basic scoring function which would provide a rating for each move explored in the game tree as well as giving a base case when either the tree has reached a maximum depth or a winning/losing position is found. In the case of a MrX win, the score for the board configuration would be 100, and -100 in the case of a loss. Otherwise, the score would be the weighted sum of the average distance between mr x and the detectives and the number of moves available to him (this would be subtracted if a detective move was being evaluated). Whilst exploring the game tree, the score for each move is the weighted sum of this function being applied to whatever board configuration the move leads to, and the maximum score which can be achieved from a subsequent move, generated from the next layer down in the tree.

Reducing the move list

It is often the case that more than one of the moves available to a player will lead to the same destination due to being able to use different means of transport. As these will lead to the same board configuration, these moves become redundant and can be removed from the list of moves to evaluate, speeding up the execution time of minimax. In our implementation, excess moves are removed from the list based on the scarceness of the required ticket(s) i.e. rarer tickets, such as double tickets and secret tickets are preserved unless they are the only way to get to a certain destination

Ordering the moves list

The moves available to the agent are initially sorted by heuristic score, i.e. the scoring function mentioned earlier is applied to the board state reached by each move, and then each move is sorted using merge sort such that the moves with the highest heuristic appear first in the list. This sorting process is also done at each layer in the game tree, massively increasing the efficiency of alpha-beta pruning. MLLO