I. Introduction

[Connor’s job]

II. Model

[Connor’s job - introduction]

a. MyGameStateFactory

MyGameStateFactory is an implementation of Factory with a single method: build, which itself returns a new MyGameState object, a subclass of MyGameStateFactory that implements GameState. Most of the operations are done in the MyGameState class and as such this section of the report will be focusing on it.

i. Atributes

The atributes of MyGameState are as follow:

* setup, (of type GameSetup) holds the setup of the current game board,
* remaining, (of type ImmutableList<Piece>) holds the pieces left to move in the current round; it alternates between the detective pices and the mrX piece,
* log, (of type ImmutableList<LogEntry>) holds mrX’s travell log,
* mrX, (of type Player),
* detectives, (of type List<Player>),
* everyone, (of type ImmutableSet<Piece>) is a set of all the pieces present in the current game,
* moves, (of type ImmutableSet<Move>) holds all the legal moves that the pieces can make in their current positions on the game map,
* remainingRounds, (of type ImmutableList<Boolean>) lists the truth values of the reveal state of the remaining rounds in relation to mrX’s travell log.

All the atributes are initialised in MyGameState’s constructor, as they are passed as arguments when the object is created, or they are derived from the passed arguments.

ii.Methods

advance

- takes as argument a move and then it calls in turn it’s helper methods to advance the game to the next game state if the move is legal.

advance’s helper methods are:

updateRemaining - updates the remaining atribute to contain all the detectives present in the game if the move was commenced by MrX. If the move was commenced by a detective, remaining is updated to contain either the detectives who have not yet made a move and have at least one ticket, or, if the detective was the last one in remaining in the previous game state, remaining is updated to contain only MrX;

updateLocations – updates the location of the player who commenced the move. This method makes use of the visitor pattern to get the final destination of the move, so there is no different case for single or double moves;

updateTickets – updates the ticket that each player holds. This method also makes use of the visitor pattern as the ImmutableList<Ticket> tickets holds all the tickets used in the move. The tickets are all used and if the move was commenced by a detective, then mrX receives said tickets;

updateLog – updates mrX’s travell log. The visitor pattern is used twice, to get the destinations of the move and the tickets used as immutable lists. For each destination of the move, a new log entry is added to log in accordance to the reveal state of the round.

getDetectiveLocation

- takes a detective as argument and returns they’re loction as an Optional<Integer>

getPlayerTickets

- has a subclass MyTicketBoard that implements TicketBoard. The function then returns an Optional of MyTicketBoard with the player’s piece (which is passed as an argument to getPlayerTickets) as an argument is returned.

getWinner

- returns a immutable set of pieces when one of the winning states is reached. The method cover all the poosible winning scenearios, both for the detectives and for MrX. In a winning state is not reached in the current game state, an empty set is returned.

Winning states:

detective winning scenarios:

* MrX is captured
* MrX is stuck
* MrX has no tickets left

MrX winning scenarious:

* the last remaining round was played and MrX was not captured
* no detective any tickets left
* all detectives are stuck

The ‘stuck’ and ‘no tickets’ scenarios for both the detectives and MrX are determined by the state of the set returned by the getMoves. If the set is empty then the conditions for these scenarios are met and the winners are returned.

getAvailableMoves

- return a immutable set of all possible moves if the game has ended. The state of the game is determined on the state of the list returned by the getWinners method; if the list is empty, then a list of moves is returnd; if the list is not empty, then an empty list is returned.

getAvailableMoves makes use of getMoves to get the list of all possible moves.

getMoves

- is a helper function for both getAvailableMoves and getWinner. It returns a immutable set of all possible moves of the pieces that are present in the remaining attribute.

If remaining contains detectives, then getSingleMoves is called for each detective and the set of all possible single moves for all detectives is returned. Otherwise, if remaining contains mrX, getSingleMoves and getDoubleMoves are called and the a set of all possible moves is returned.

getMove’s helper methods are:

getSingleMoves – goes through all adjacent nodes where a detective is not already present and if the player (which is passed as an argument) has the required ticket to travel to that location, the move is added to a list. If the player has a ‘secret’ ticket, than they can travel to any non-occupied adjacent node regardless of the required ticket type.

getDoubleMove – calls getSingleMoves twice, once on to establish all possible single distance moves and one to determine all distance two moves. The method also checks for the pressence of a ‘double’ ticket.

iii. Getters

getSetup, getPlayers, getDetectiveLocations, and getMrXTravellLog all retrun atributes of MyGameState.

iv. Evaluation

Our code for MyGameStateFactory was largely structured around GitHub user armandcismaru’s code, however we consider that our solution is a significant improvement in terms of readability and modularity of the code, as well in terms of efficientcy. When ran on the same machine, our code passed all 83 tests ~20 ms faster on average than their’s, and the t-test indicates that the difference in runtime is statisticaly significant, with a p value of ~0.025 when run a 20 sample data set.

We belive that our solution can be further improved upon by optimising the getDoubleMoves method by using dynamic programming principles.

Our code presents a warning in the InteliJ IDE stating Condition 'mrX == null' is always 'false'. This can not be avoided due to the nature of one of the tests that requires for a null pointer exception to be thrown in the case that the mrX object is null.

b. MyModelFactory

[Connor’s job]

III. AI

[Introduction to ScotFish – Connor’s job]

i. Atributes

* board, (of type Board) holds the current status of the game board,
* scoreMap, (of type Map<Integer, Integer>) maps each node to a score value,
* detectives, (of type ImmutableList<Detective>) holds all detectives that take part in the game,
* mrX, (of type Piece) holds MrX’s piece.

Methods used:

ii. initialiseScoreMap

- initialises all the nodes with the initial score value of 0.

iii. setPlayers

- initialises the detectives and the mrX attributes.

iv. getDetectiveLocations

- returns the current location of all the detectives presesnt in the game as immutable list of integers.

v. setScoreMap

- sets the actual score values by calling helper methods.

setScoreMap’s helper methods:

setDetectiveAdjacentNodeScore(N, F)

– parameters: N of type int, score modifier;

F of type int, modifying factor.

- calls setAdjacentNodeScore and decreases the score br N \* 5 points for each detective location and

setAdjacentNodeScore(origin, N, F)

-parameters: origin of type Integer, the origin node in relation to witch the scores for the adjacent nodes is determined;

N of type int, score modifyier;

F of type int, modifying factor.

-creates a list distanceOneLocations of all the nodes of distance one and decreases the score by N points. Then for each node in distanceOneLocations, all adjacent nodes’ scores are decreased by N / F points.

setDetectiveAdjacentNodeScore andsetAdjacentNodeScore create a zone of decreased score value around the detective pices. This result in MrX ‘running away’ when the other pieces get close to him and also insures that picking a move that places him in proximity of the detectives is less likely. During play-testing we have determined that a value of N = 100 and F = 2 result in desireable behaviour.

setFerryNodeScore(N)

- parameter: N of type int, score modifyier.

- increases the score of the nodes that have a ferry connection by N points if MrX has at least one ‘secret’ ticket. If MrX is already standing on a ferry node, then that node’s score is reduced by N points, to avoid the case of a double move to the same location.

setFerryNodeScore makes ferry movments more likely to be picked, as we found they often are harder to track by detectives. We have observed that a value of N = 125 gives the best behaviour. The score of a node that has distance 2 to a detective but is a ferry node has a value of 25, which is slightly higher than 0, sometimes makes for unexpected and riscky moves by the AI.

vi. getBestMove

- picks the move that the AI makes out of the list returned by getHighestValueMoves. It separates the moves into three lists: normalMoves, secretMove, and doubleMoves. As the name implies, secretMoves holds all the moves that use a ‘secret’ ticket and doubleMoves holds all the double moves. Normal moves holds all the moves that are do not meet the conditions of the other lists. As such secretMoves and doubleMoves are not mutually exclusive, so a double move that uses at least one ‘secret’ ticket is present in both lists.

The method picks at random a move from one of the three lists. The chance of a normal move to be picked is 80%, whereas a chance for one of the special moves to be picked is 20% (10% for a double move and 10% percent for a secret move). This makes sure that MrX does not use his special tickets too quick while retaining the element of surprise when he does.

GetBestMove’s helper method:

getHighestValueMoves

- returns all moves that have the same highest score as an immutable list of integers. It iterates thgrough scoreMap twice; once to determine the maximum score of the move’s destination node, and once more to add the moves that have the destination node’s score equal to the maximum score to the return list.

vii. Evaluation

There are features that we would have liked to have implemented but unfortunetly we couldn’t beacouse of time constraints. An implementation of Digkstra’s Algorithm to compute the optimal paths that the detectives would take to reach MrX and decrease the score of the nodes along the paths was planned. We would have also liked to implement a look ahead algorithm that would use the optimal paths to return the best line of MrX moves, similar to chess engiens.

IV. Partner’s Contributions

TO DO

V. Workflow

TO DO (if there is time and space for it)

[Closing statement – Connor’s job]

h h h h h h