wr15339 tj15299

Scotland Yard Report

CW5

Started with the initialisation of the Scotland Yard board. Firstly we created the constructor. In the constructor we declared two array lists to hold the data of the spectators and the players. We also created two integer variables, one to store the location of mrX and another to store a count of the number of players who had joined the game. We then implemented the method for players to join the game by adding each player to the playerList and returning false if the player was already added to the game. We then implemented the isReady() method by checking if all of the detectives had been added to the playerList. Inside Scotland Yard we also created several methods that retrieve a player's locations, tickets and also the list of rounds.

Next we implemented the methods that allow the players to interact with Scotland Yard. These included a method to return the current player, change to the next player and a notify method that tells a player that it is their turn and gives them a list of playable moves. These moves are worked out in our valid moves functions that calls generateMoves in Scotland Yard Graph. GenerateMoves creates a move for each outgoing edge from the player's current location and then checks each of these against the player's tickets, removing it if unplayable. The play method then allows a player to play their chosen move, remove the required tickets for that move and also increment the round counter.

We then implemented the methods that allow spectators to view the game as if they were a detective i.e. not being able to see Mr X's current location. These methods added spectators to a list and notified them when a detective made a move.

Finally we needed to create methods that checked whether the game had finished and whether the detectives or Mr X had won. The isGameOver and getWinning Players methods return detectives as winners if any of the detectives occupy the same node as Mr X or if Mr X has no valid moves or no tickets. They return Mr X if the detectives have no valid moves, no tickets or if the round limit has been reached and Mr X has not been found.

CW6

For the second part of the coursework we firstly had to implement a messaging service for the client in JavaScript. We began by creating an interpretReady function that takes a ready message, stores the information (current round, player locations and player tickets) as variables and then sends this information to the GUI connector. The next method was interpretNotify which takes a notify message sent when a move is played and updates the user interface with the new tickets and what type of move was just played. It also tells the GUI to animate the player to their new location.

The interpretGameOver method tells the UI that the game is over and also notifies the AI players (if there are any and they are connected) that the game is over. interpretConnection takes a connection message and sends the information within to the judge. The final method that we created was interpretNotifyTurn which takes a notify message from an AI player and sends it to the AI server.

We started to implement some basic functions for the AI including a scoring heuristic. This takes in a game board and scores it based on distance to detectives, number of outgoing transport routes, the types of these routes. After reading some strategy guides we also decided to weight the nodes in Regent's Park less because of the low number of outgoing routes and how it tends to trap Mr X.

Next we decided to implement the MiniMax algorithm. This required us to create a new graph structure to simulate a game tree. The nodes in the graph each represent a game state and the children of each node represent all possible new game states after the next move is made. The algorithm then searches each ply for the node with the maximum score if it is Mr X's turn or the minimum score if it is a detective's turn. The algorithm returns the best move Mr X can make to maximise his score if the detectives always make the best move available to them.

We created a class called GameBoardSim to hold the player locations and tickets of a game state. Inside it's constructor it would receive a List <Move> parameter which would either contain the single move MrX made in that turn or the moves each detective made. The constructor would also receive the player locations and tickets from the previous state. It would then use the move made to update the tickets and locations. Unfortunately we ran out of time before we could implement that correctly and the ticket numbers are being altered incorrectly. We then have methods inside GameBoardSim which generate possible detective and MrX moves based on the simulated game state. It also contained methods that checked whether MrX or the detectives had won in this game state. Inside the minimax algorithm we had 2 methods.

The first (max) would take the current game state and check if we had reached the maximum depth or if either the detectives or MrX had won the game in that state and assigned a score to that state accordingly. It would then return that game state. If those conditions were not met, it would instead generate a list of possible MrX moves and create a new game state for each move which simulated the board as if that move was made. It would then set move node to contain the result of the min function run on the new state with a depth decremented by 1. It would go through all of these new move nodes and find the one with the highest score. It would then return this node.

Min was a very similar algorithm which also returned the state with the correct score assigned if the game had ended or the depth was reached. Otherwise it would create nodes which simulated the moves the detectives could make. Max was run on each of these states to find the best node resulting from each state. The min function then found the worst node of all those results as that was the node the detective would be most likely to choose. It then returned that node.

Given more time we would have liked to finish the MiniMax algorithm. We would have liked to firstly increase the number of possible detective moves generated as at the moment we assumed they would move as close as possible to MrX. We would also have liked to perfect out GameBoardSim so that it created an accurate simulation of the locations and tickets. Furthermore, we would have implemented alpha-beta pruning to reduce the number of redundant branches (where the move already has a lower score than a previously evaluated one) that are searched. We would also like to have completed our basic AI for the detectives which used the Page Rank algorithm to move towards the most connected nodes for the first few turns and once Mr X is revealed use Dijkstra's algorithm to move towards Mr X's last known location.

Our implementation using MiniMax, even with alpha-beta pruning, would still fall victim to the horizon effect as it can only search through so many plies before running out of time. This means that even for a move evaluated as bad the next ply may give very favourable scores or vice versa. This could be countered by the use of quiescence search which checks whether child nodes are "noisy" or "quiet" (say if detective's get very close to Mr X or there is a direct route from a detective to Mr X).