Scotland Yard Report

ty24512 and in24486

Amaan Raazi and Aayush Bhalerao

We managed to implement the game passing all 83 tests in CW-Model and implemented an AI for MrX in CW-AI.

CW-Model: Starting from build, we conduct some basic checks ensuring such MrX is non null, ensuring no player has an invalid ticket and all detectives have unique locations, throwing the appropriate exception in case any of these tests fail. We construct a set of all of the game pieces. Then we call getMoves and return a newly constructed MyGameState using all the above-mentioned arguments.

In getAvailableMoves we call the constructAvailableMoves method and return the results. In constructMoves, we construct a set of all possible mrX single and double moves or all possible detective single moves depending on turn using helper function such as calculatePlayerSingleMoves and calculateMrXDoubleMoves.
This is a deliberate design choice to allow the public getter to remain simple while delegating the calculation to the private helper function.

In MyGameState constructor we again do some basic checks like whether the set of moves or detectives are empty etc. and handle them accordingly.

The checkGameOver function checks for a variety of game end conditions and returns the winner or an empty set accordingly.

Then comes the advance method. Here we have used the visitor design pattern. If a Single Move is passed in call either handleMrXSingleMove or handleDetectiveSingleMove. And if Double Move is passed, we call handleMrXDoubleMove.
In the handleMrXSingleMove, handleDetectiveSingleMove and handleMrXDoubleMove, we update the locations of the player, update the logs and update tickets accordingly.

Then we have updateMrXTravelLog which updates the travel log, using helper functions updateMrXTravelLogforDouble and updateMrXTravelLog, these functions being self-descriptive.

Then we have two methods to update MrX location for Single Move and Double Moves and detective locations for single moves.

getSetup, getPlayers, getDetectiveLocation and getPlayerTickets, getMrXTravelLog and getWinner are simple getters.

CW-AI: We managed to build a working AI for MrX by implementing MiniMax, Alpha Beta Pruning, parallelization, Game trees and effective scoring functions based on distance using Floyd-Warshall Algorithm, tickets and freedom of movement.

We first precompute the shortest distance between each node in the graph. We do this using Floyd-Warshall Algorithm. This is quite space inefficient but allows for O(1) lookup. This is a deliberate design choice as Scotland Yard's graph has a limited number of nodes (<200) but a large number of calls are made while picking moves and calculating distance from MrX to detectives.

We also store an array of all pieces and all occupied detective positions on any given moves. This minimizes calls to the board's getters, thus saving time.

In pickMove, we first get candidate moves. These are sorted shallowly to help in pruning. We then implement parallelization to consider all these moves deeply using minimax.

In Minimax, we use a max depth of 7, this allows us to simulate 2 MrX moves and 1 move each for all 5 detectives. This helps us look ahead and make MrX make optimal decisions. MiniMax is implemented with alpha-beta pruning in a textbook fashion with Moves being stored in a priority queue to allow faster sorting and thus make pruning efficient.

We employ the visitor design pattern in the getDestination and updateVisitedLocations helper functions. They are used to get the final destination of a move and update the array of already visited locations respectively.

The evaluateBoard function is key to our strategy. It uses the getMinDistance helper to get the minimum distance between mrX and a detective. It also uses calculateFreedomScore to get the number of unoccupied neighbors and subsequently returns them after combining them with different weightages.

The evaluateMove function discourages SECRET and DOUBLE ticket usage in non-critical scenarios (i.e. non reveal rounds and early rounds) by adding a penalty to such moves. It also helps avoid backtracking and repetition by penalizing such moves.

The simulateMrX and simulateDetectiveMoves are self-explanatory. They return a ProxyBoard with updated locations and newly constructed move set (according to turn order). The ProxyBoard class is a class that implements a board by taking an existing board and returning all of its methods. This allows us to override specific methods without implementing others from the abstract interface board.

The constructMrXMoves, constructMrDetectiveMoves, getMrXMoves, getMrDetectiveMoves are simple constructors and getters borrowed from CW-Model. They help us to construct moves for the next player based on the turn order while simulating moves.

We only construct single moves for MrX in constructMrXMoves. This is to make minimax faster at high depth. Double Moves are however fully considered in the first move selection.