Scotland Yard Game Coursework

Chris El Akoury iy21003@bristol.ac.uk
Emir Abou Zaki jk22756@bristol.ac.uk

March 13, 2024

1 Introduction

1.1 Scotland Yard Game

Scotland Yard is a board game where players take on the roles of detectives trying to catch a fugitive named Mr. X, who tries to avoid capture and escape from London. The game is played on a board that depicts the streets of London. Mr. X's location is kept secret to the detectives, but he has to share his location every few rounds. The detectives move their pieces on the board trying to catch Mr. X. On each turn, a detective can move to an adjacent space on the board. Mr. X can use secret moves and double moves to avoid capture. If a detective lands on the same space as Mr. X, they capture him and win the game. If Mr. X manages to avoid the detectives and survive all the rounds, he wins the game.

1.2 Project Overview

In this coursework, we were able to pass all tests in the CW-model and implement a Mr.X AI for cw-ai using Minimax algorithm with alpha-beta pruning and attempted to implement Monte Carlo Search Tree for Mr.x AI.

2 CW-Model

2.1 Completing MyGameStateFactory

We began by initialising the constructors in the MyGameState class which implements the GameState interface. This allowed us to pass the first few tests. Then we started completing the methods in this class. Some of the methods' implementations (getSetup, getPlayers, getDetectiveLocation, getPlayerTickets, getMrXTravelLog) were rather straightforward. However, for getAvailableMoves, we had to create helper methods. The first helper method we used is makeSingleMoves which returns an ImmutableSet of all the possible single moves a player can make from a certain position. Similarly, makeDoubleMoves returns an ImmustableSet of all possible double moves mrX can make from a given position on the game board. Both these methods use occupiedLocation which is a method we created that returns a Boolean value representing whether the given location is occupied by another player. Furthermore, getPlayerMoves calls both makeDoubleMoves and makeSingleMoves and returns an ImmutableSet of all possible moves a given player can make. Finally, getMoves loops over the remaining players calling getPLayerMoves and returns a set of all possible moves given a GameState.

2.2 Visitor Pattern

We used the visitor pattern in our implementation of the advance method in MyGameStateFactory. We utilized anonymous vistor classes that override advance for both double moves and single moves. These methods modify the player constructor, updating his position, travel log and the number of tickets he possesses based on the move he chose to make.

2.3 Completing MyModelFactory

MyModelFactory is the class that represents the concrete observer and implements the Factory¡Model¿ interface. It contains the chooseMove method which updates the state of the game and notifies the observers after move has been chosen.

3 CW-AI

3.1 State Class

We created a State class as a wrapper for the model's GameState class of the game. This provides helper functions that are used for the AIs. This made it easier to score states and have the required methods used by the AIs.

3.2 Scoring Evaluation

We used Dijkstra's shortest path algorithm to calculate the shortest path from this potential location of MrX to the locations of each detective. The sum of all the distances are added to the number of unoccupied nodes that are adjacent to this location (multiplied by a constant) to make up a score that is attributed to this potential location. The score is also affected by the current round. Moreover, secret moves are encouraged on the subsequent rounds of the rounds where MrX's location is revealed.

For our implementation of the Minimax AI, we needed a way to predict which moves are most likely to be played by the detectives. To score the detective moves, we used Dijkstra's shortest path algorithm. Moreover, the location which brought the detective closer to MrX was chosen.

3.3 Look-Ahead-One AI

This AI looks at all the possible moves that MrX can make in the next round and uses our scoring heuristic above to score each possible move. The AI then plays the move with the highest score.

3.4 Minimax AI

The minimax algorithm is a popular approach in artificial intelligence for decision-making in two-player games, where the player has access to all the information. It is used to determine the best possible move for a player, assuming that the opponent will also make optimal moves. This is comparable to the Scotland Yard game conditions considering MrX is playing against all the detectives (2 teams) and he has access to all the information on the game board.

The algorithm tries to minimize the maximum potential losses that MrX may incur from a given move, while at the same time maximizing the minimum potential gains he may achieve.

The minimax algorithm is ran on all possible moves of the current state, and returns the score of each move. The algorithm works by recursively evaluating the game tree, starting from the current game state and moving down to the leaf nodes of the tree, to a certain depth that we determine. At each level of the tree, the algorithm takes turns maximizing (mrX) and minimizing (Detectives) the scores. It assumes that both players will make optimal moves.

The minimax algorithm uses our heuristic scoring function to evaluate the leaf nodes of the game tree, assigning a score to each node based on the desirability of the outcome for MrX. The algorithm then works backward up the tree, propagating these scores up to the root node, where it selects the move that leads to the best possible outcome for MrX.

3.5 Alpha-Beta Pruning

Alpha-beta pruning is a method used to optimize the Minimax algorithm. It eliminates the need to evaluate all possible moves, leading to a faster search. We maintain two variables, alpha, representing the minimum score that mrX can achieve, and beta, representing the maximum score that mrX can achieve. During the search, if a move leads to a score lower than the current value of alpha, or a score higher than the current value of beta, that move is pruned (not evaluated further).

To confirm the efficiency of alpha-beta pruning, we conducted some tests on the same state of the game. One with alpha-beta pruning and one with just the original Minimax algorithm, the following results were observed:

Minimax (depth = 6)	Number of Recursive Calls	Run Time (ms)
With Pruning	970	1,301 ms
Without Pruning	21,984	5,997 ms

Table 1: Table showing the number of recursive calls and run time taken by Minimax with and without pruning

Table 1 proves the efficiency of alpha-beta pruning with the Minimax algorithm, as that the number of recursive calls with alpha-beta pruning has decreased by over 95%.

3.6 Monte Carlo Search Tree

Monte Carlo Search Tree (MCST) algorithm is commonly used for building board game AIs. MCST works by creating a tree of possible moves and their outcomes. In each iteration of the algorithm, MCST traverses the tree from the root node to a leaf node using a selection formula, expands it by adding all possible legal moves as children to that node, and randomly simulates a game from the newly expanded node. After the search is complete, the algorithm chooses the move with the highest win rate as the best move to make.

We tried a basic implementation of MCST by utilizing MonteCarloNode class for the nodes of the tree and MonteCarlo class that is responsible to search through the nodes and apply the 4 steps of MCST for a number of iterations.

`The 4 steps of Monte Carlo Search Tree:`

1. **Selection**
   This step starts from the root node and successively selects child nodes until it reaches a leaf node that doesnt contain any children. This selection is done by applying the following formula to the nodes and selecting the child node with the maximum value:

$$UCT(x) = \frac{w_i}{n_i} + c\sqrt{\frac{\ln(N_i)}{n_i}} \tag{1}$$

   - w_i stands for the total reward for a certain node considered after the i-th move
   - n_i stands for the number of simulations/plays for the node considered after the i-th move
   - $N_i$stands for the total number of simulations/plays after the i-th move run by the parent node of the one considered
   - c is the exploration parameter set equal to 3; which is the balance between exploration and exploitation.

2. **Expansion**
   The selected node is then expanded by adding all possible child nodes to the tree (all the possible moves from that state)

3. **Simulation**
   The algorithm then performs a simulation by playing out a roll-out policy, in our case we have 2 policies, one which plays random move until the game is terminated and one which plays the optimal moves until the game is terminated. The second policy plays out very slow and is not the most efficient to use in MCST. After the simulations are done, a reward/score is returned depending the on the winning player of the game.

4. **Back Propagation**
   The results of the simulation are backpropagated up the tree, updating the statistics of the nodes that were visited during the simulation. This includes the number of visits and the total reward obtained from each node.

4 Limitations and Reflections

4.1 Minimax algorithm Limitations

The Minimax algorithm assumes that both players (or teams) will choose the best move at every turn. However, this is not always the case in practice. Humans are prone to making mistakes and sub-optimal decisions.

The time complexity of the Minimax algorithm with alpha-beta pruning is still exponential in the worst case, which can become a problem when we're dealing with very large game trees.

4.2 Scoring Function Limiations

The scoring function is good at determining how advantageous a certain state is for mrX. However, it isn't as good at determining if the move used to reach this destination is the most efficient one. We were able to encourage the use of secret tickets after reveal rounds, and double moves only when single moves have low scores or when mrX is trapped and needs to escape. However, when a double move is the optimal move during a reveal round, it plays a double secret move which is sub-optimal considering his location will be revealed anyway after the first secret move. This is due to the way in which the scoring of secret and double move tickets are defined and could be improved by utilizing move filtering or by manually implementing checks for double moves when using secret tickets.