OOP Coursework - Scotland Yard

Kiran Steward, Michael Poluektov

I. OVERVIEW

In this coursework we have successfully completed the Scotland Yard model with all tests passed. Furthermore we have worked on a MrX AI and successfully implemented the MiniMax algorithm with Alpha and Beta pruning, as well as a version of the Monte-Carlo tree search algorithm with light, heavy playouts and tree parallelisation.

II. CW-MODEL

We have successfully implemented all methods of the `GameState` interface with all the appropriate checks.

1. Limitations

- Our `getRound()` helper method relies on counting the amount of double move tickets used by Mr X, which uses the fact Mr X starts with two of those tickets, making our implementation non scalable to any other number of double move tickets.

2. Merits

- Passed all tests & produced a playable game.
- Combining single and double move generation into a single `makeMoves()` method decreases the size of our code as well as the runtime of MrX's move generation (if he has double tickets).

III. CW-AI OVERVIEW & MINIMAX

Below you can find a brief description of our achievements in the open-ended task of the coursework. For more information on the implementation, check our comments and JavaDocs. A good place to start examining our code is in the `...ai/mrx/` folder. Disclaimer: to ease configuration, we store some of our constants in a Toml file located in `/src/main/resources`. That file is parsed using the `Toml4j` external library.

1. Main interfaces & MiniBoard

To reduce runtime and ease access to data required by location pickers, we have created a `MiniBoard` class which stores only the necessary information needed to score a location. The class also contains simplified methods from the Board class such as `advanceMrX()` and `advanceDetective()`. The main limitation of the `MiniBoard` class is the lack of ticket tracking, as well as it's inability to differentiate between the detectives and MrX's move when it comes to using a ferry.
Each of our AIs is split into 3 blocks, with the functionality of each block described by a separate interface, allowing us create each agent in a modular way by combining these key functionalities:

- A `LocationPicker` provides the method `getScoringMap(...)` which a map containing the provided destinations and their respective scores.
- A `TicketPicker` provides the method `getBestMoveByTicket(...)` which outputs the best move out of a set of moves with the same destination based on the tickets used in that move.
- A `MovePicker` provides the method `pickMove(...)` which outputs the best move based on a `LocationPicker` and `TicketPicker` and limits the use of double moves.
- `IntermediateScore`s are objects used by the `MiniBoard`'s `getMrXBoardScore(...)` method. They are used to provide a heuristic evaluation of MrX's position.

2. Default implementations & limitations

One limitation of that structure is that ticket selection and destination selection are handled separately, which limits our ability to pick a move where one factor is influenced by the other (for example, being less inclined to use a ferry requiring a secret ticket, if the destination score is similar to another destination). We have chosen to give up that flexibility in favour of a more modular way to construct an AI. Here is a brief explanation of our implementations of that structure:

- Our default ticket picker picks whichever ticket MrX has the most of, unless the last round was a reveal, in which case MrX picks a secret.
- Our move picker first considers single moves, and if and only if their scores are below a threshold, it adds double moves. In the case of MCTS we use the Observer pattern to add double moves dynamically.
- Our location pickers pick the best location with the assumption that the detectives are always aware of it.

3. MiniMax algorithm

One of our location pickers picks the location based on the MiniMax algorithm, which is optimised with alpha-beta pruning. One of it's biggest limitations is the lack of a cut-off point when the timer runs low, so by running the algorithm we are either risking low accuracy from a small amount of calculated steps or MrX losing by timeout. The algorithm iterates through every position of the game after N moves (except for those pruned out, which can not be optimal) and compares these positions' heuristic scores, then

picks whichever move location guarantees the best score for MrX. Another limitation of this approach is that we are giving a much higher priority to exploration over exploitation, which means that we're spending too much time finding redeeming qualities of seemingly bad moves, rather than aiming to find the best move out of multiple seemingly good moves.

IV. MONTE CARLO TREE SEARCH ALGORITHM

The Monte Carlo Tree search algorithm (MCTS for short) aims to resolve the above limitations of the MiniMax algorithm. MCTS efficiently samples a game tree with a high branching factor by using reinforcement learning. It is *asymetric*, which means that it will explore more promising options first, without completely discarding alternative paths. It is also *anytime*, which means it can be easily stopped at any point and return it's best guess at what the optimal move is. As a bonus, by running it with a variable amount of iterations dictated by hardware and by utilising randomness in some parts of MCTS, it can be considered *non-deterministic*, making it harder for detectives to predict MrX moves even with knowledge of it's move picking algorithm. In order to further understand the advantages and drawbacks of different implementations, see the brief explanation of the default version of MCTS below.

1. Basic functionality

To use MCTS, we first initialise a *search tree* containing a root node, representing the agent's current game state and we initialise it with *child nodes*, that represent all the outcomes of the agent's available moves. Each node has 2 variables associated with it: a score, and a number of plays, both initialised to 0 upon creation. We then run the following 4 steps until a time limit has been reached:

1. **Selection –** We start from the root node and recursively select a child that maximises the *UCT* function, defined below, until we reach a *leaf node*: a node without any children added to it.

$$UCT(i) = \frac{w_i}{s_i} + c \cdot \sqrt{\frac{\ln s_p}{s_i}}$$

With w_i the total score, s_i the amount of plays, c being the exploration constant, and s_p being the amount of plays of the parent.

2. **Expansion –** After selection has been finished, we randomly chose an unvisited move and create a child node corresponding to the outcome of that move. If the current node is *terminal* (the game state has a winner), this step is skipped.
3. **Rollout –** The during the rollout phase, (aka *simulation* or *playout*) we are simulating a playout of the game until we reach a terminal state and save the result. The default version of algorithm uses a *light* playout, which means that the selected moves are random. These randomly generated states are not added to the search tree.
4. **Backpropagation –** The result of the rollout is then recursively propagated up the tree until we reach the root node. It is important to note that that the score of each node represents it's *parent node*'s agent's likelihood of picking that node.

2. Standard implementation

The classes `LightNode` and `TreeSimulation` represent a default MCTS implementation, without any game-specific improvements. In that class, we are continuously running the 4 above steps. The rollout returns the round at which the game has terminated. Here's how that result of the rollout of a node affect it's ancestors:

- The score of the root node is never changed or accessed.
- The amount of plays is incremented by 1 in any case.
- For a node representing the outcome of a move by MrX, we backpropagate the amount of rounds MrX has survived for after the initial position.
- For a node representing the outcome of a move by a detective, we backpropagate the total amount of rounds minus the result of the rollout.

When the time limit is reached, we output the move with the highest average score. This approach produces good results, usually predicting the amount of rounds MrX can survive for, if it is less than or equal to 3. The performance of this algorithm may be compared to a 3-8 step lookahead AI, depending on the hardware and the branching factor. It's main limitation seems to be positions where MrX already has a good position with the detectives being far away.

3. Scotland Yard specific optimisations

One optimisation we can make is replace the random "light" rollout with a deterministic "heavy" rollout, which would assume that at each state the agent would make the best move based on one step lookahead. This would then utilise a heuristic function such as our location scoring class based on the Breadth First Search algorithm, and greatly improve the precision of each rollout. However due to the increased complexity of each simulation, we were unable to run a sufficient number of simulations to noticeably improve the quality of the predictions.

4. Parallelisation

Unlike the default implementation of the algorithm, the performance of MCTS with heavy rollout can be greatly improved by utilising multiple cores. Since the time it takes to execute each cycle is taken up mostly by the rollout phase, which does not need to access or modify the rest of the tree, we can easily execute this stage in parallel by slightly delaying backpropagation. This would mean that the selection phase may be slightly less accurate, however on the scale of thousands of simulations this doesn't appear to have a significant impact. The results of this method seem similar to the results of the default MCTS implementation, with the heavy rollout version giving better predictions in states where the detectives are far from MrX. Unfortunately we didn't have enough time to perform proper statistical tests on the relative performance of the 2 versions of the algorithm.

V. MAIN AI LIMITATIONS

1. Structural limitations

Despite our AI's producing good results, there are still many potential improvements to be made. Some limitations of our `MiniBoard` data structure prevent certain strategies to be taken into account, such as the detectives trying to ”starve” MrX by not using a given type of ticket, MrX not considering nodes adjacent to a ”stuck” detective as dangerous, or MrX taking advantage of the detectives' unability to use a ferry.

2. Hidden information

It is also important to note that while our algorithms treat Scotland Yard as a game with perfect information, the detectives do not actually know MrX's location at all times. Despite MCTS being originally intended to be used in games with full information, it was proven to be a viable solution to solving game with imperfect information as well. For example, we could simulate the detectives' move choices more realistically in the rollout phase, or even apply it in creating a detectives AI which aims to reduce the amount of positions MrX can be at.

3. Optimising our heuristics

Many of the functions used by MrX to pick a move contain arbitrary constants, the values of which influence our agent's decisions. We could optimise those values by either spending more time on play testing, or by self-play with a detective AI, using gradient descent with a loss function based on the outcome of the game to optimise both players' heuristic functions. Alternatively, we could attempt to replicate the approach used by Alpha Go Zero, and replace the rollout phase completely with a machine learning model trained to predict the round MrX is expected to lose at in a given board.