Scotland Yard Report

Introduction

In our project, we **passed all tests** in Model part and have attempted AI part. In this report, the sketch of Model and the implementation of AI will be introduced. After that, possible limitations of both parts will be discussed.

The Sketch of Model

In this section, getAvailableMove(.) and advance(.) methods in MyGameStateFactory will be explained. Also, build(.) method in MyModelFactory will be explained.

getAvaliableMove(.) method

We have created two helper functions which are makeSingleMoves(.) and makeDoubleMoves(.). In getAvaliableMoves(.) method, we create allPossibleMoves to hold every possible move for players that have not made a move in the current round yet. We loop through all remaining pieces and generate moves they can make with those helper functions and add them to allPossibleMoves. If the winner set is not empty, an empty set is returned, such that no moves are available.

Advance Method

We create an anonymous inner class for Visitor when using visitor pattern. So, we have our elements (i.e. SingleMove and DoubleMove) and our Visitor (i.e we choose void, we update the attribute of in gamestate). Visitor pattern uses double dispatch to let moves to single dispatch first of the reference of the visitor and the void visitor to dispatch again to determine whether it is a SingleMove or DoubleMove. In the visit method for Single Move, we check if this move is commenced by detective, then update that detective location and give that ticket to mrX, update mrX state and update the detectives list.

We create some helper functions and use it in advance method like updateMovesAndRemaining(.), addMrXLog(.), checkWinner(.) and revealMovesThisRound(.) to update gamestate that moves are commenced by that specific player (i.e. mrX or detectives).

Build(.) in MyModelFactory class

We implemented the build method via observer pattern. We have our concrete observer and our concrete observerable (model). We created a notify method to update all observers about the events. In chooseMove method, we update the gameState with advance(.) and use notify(.) to update all observers with the events occured. By using the observer pattern, our model does not need to know anything about the observers so observer and observerable are independent to each other. Also, observers can choose to subscribe or unsubscribe to the observable dynamically at runtime without changing the state of model.

Implementation of AI

In this section, the scoring algorithm of the AI will be introduced. Next, two AI, namely MrXAI and DetectiveAI will be introduced. Also, one external library, GraphStream[1] is used in ScoreUtils class for Dijkstra's algorithm.

Structure of Score

Firstly, a Score interface is implemented, which ensures all Score implementations have the same structure. Moreover, Strategy pattern is used in Score interface with ScoreContext as concrete strategy. ScoreContext can be implemented and contains different sets of attributes, allowing different implemented methods accepting different number of arguments. In this project, two Score classes, which are GameTreeScore and DetectiveScore are implemented. The former scores all available moves for MrX and the latter scores for detectives.

Secondly, ScoreUtils class is created with all useful scoring and helper functions. For example, in calculateDistance(.), Dijkstra's algorithm[2] is used to find the shortest path between the source and destination. In calculateTicket(.), different types of tickets will be mapped to a well-defined score and the average score of all tickets will be returned. In calculateFreedom(.), we use the degree of source node as the scoring scheme but if an adjacent node is occupied by detective, the score will be 0.

MrX AI

In MrXAI, a GameTree instance of the current board is created. In the GameTree constructor, all child GameTree, which is basically all possible GameState involving detectives after mrX moved, is built using the build(.) method recursively. The number of levels is decremented each time when building the List of child GameTree until

[1] GraphStream - A Dynamic Graph Library. (n.d.). *GraphStream - A Dynamic Graph Library*. [online] Available at: https://graphstream-project.org/.
[2] Imported from GraphStream [1]

levels reaches 0. In our AI, levels are defined as 2, meaning that MrXAI will predict 1 step of detectives' move after MrXAI moved.

After that, the constructed GameTree is passed into GameTreeScore constructor. GameTreeScore inherits ScoreUtils and implements Score, which built itself into a general structure. In there, minimax algorithm is used to return the best move for mrX in getBestMove(.). Specifically, all child GameTree are scored and the minimum score will become the score of the root. The maximum score of all roots will be consequently obtained so that the optimal move for mrX can be found.

Detective AI

In DetectiveAI, a DetectiveScore instance of the current board is constructed and getBestMove(.) is called consequently. In getBestMove(.), all possible detectives' moves are scored with algorithms like how we scored mrX moves. The difference here is that only distanceScore, which is the minimum distance to mrX's last reveal location, is accounted as the scoring scheme.

Limitations

There are a few limitations in our AI implementations. Firstly, a weighted scoring algorithm should be introduced. As the total score of a move is generally the sum of three different scores, the sum needed to be weighted to provide a more optimal estimation of the best move. To a higher extent, the AI could be able to adjust the weight itself, meaning that it can "learn" how to play the game.

Secondly, GameTreeScore only supports levels 1 or 2. As if it is greater than 2, the program runtime will be exponentially increased, causing the AI cannot move within 15 seconds. Instead of building the whole GameTree, pruning like alpha-beta-pruning needs to be utilized to reduce the tree size, ultimately boosting the runtime.

In more general, there is still room for improvements for our coding. For example, we need to learn how to make comments on our code in a more precise way to improve readability. We also need to reflect our code regularly even though it works perfectly, as there might be a much better way to do it, making the code cleaner and more efficient.

References

[1] GraphStream - A Dynamic Graph Library. (n.d.). *GraphStream - A Dynamic Graph Library*. [online] Available at: https://graphstream-project.org [Accessed 01 April. 2024].