Scotland Yard Coursework Report

Summary

Our Scotland Yard game is fully implemented, with both user and AI players for Detectives and MrX. We pass all tests in cw-model and provide a look-ahead AI in cw-ai;

cw-model:

The cw-model is based on the two classes *MyGameStateFactory* and *MyModelFactory*. The former creates a *MyGameState* which carries out the game logic for each round, including checking for winning states and moving players. The latter produces a *MyModel* that fulfils the observer design pattern. Both classes include key getter and setter methods which modify private class attributes. This maintains encapsulation, preventing accidental editing of attributes that should not be accessed.

The visitor design pattern is used in *advance*. The twice overloaded function doMove is passed into a new *FunctionalVisitor*, an implementation of the *Visitor* interface. The move (element) then accepts the new *FunctionalVisitor* (visitor). Double dynamic dispatch allows the specific functions to be called in the case of double and single moves.

The observer design pattern implemented in MyModelFactory notifies observers of ‘GameOver’ or ‘MoveMade’ board states, after a game state is advanced. Methods are implemented here that cover registration of observers.

cw-ai:

“The Real Slim Shady”, is a MrX AI. A minimax algorithm is used recursively with a 3-move lookahead. It generates a game tree with alternating layers of MrX and detective moves. If the current game state contains a winner, that node is evaluated. At each layer, the score returned is the max score from the tree below on MrX’s turn, or the lowest score on the detective's turn. This model assumes the detectives will make the best possible moves each round. Alpha-Beta pruning is used to break off parts of the tree where moves will not be taken, due to more suitable moves being available at that level.

“The Fake Slim Shady“, is the Detectives AI. When MrX has not revealed his location yet we select random moves for the detectives. Otherwise it traverses backwards through the MrX travel log to find MrX’s location upon last reveal. Our AI chooses the move for each detective that minimises the distance between itself and MrX’s last known position. This increases the detectives’ chances of catching/surrounding MrX.

Achievements : cw-model

<u>All tests passed:</u>

- 83 of 83 tests passed

<u>Helper methods:</u>

- Repeated functionality implemented in methods (DRY)
- Improved readability by separating helper methods (e.g: *stuckRemover* and *logHelper*)
- An example is shown below in figure 3

<u>Strategy Pattern:</u>

- Methods were overloaded where possible, maximising readability and allowed for case dependent specialisation
- For example, *isStuck()*, *isWinner()* and *doMove()* are case specific, changing if called on MrX or detectives. *isStuck()* is shown below in figure 1

<u>Visitor Pattern:</u>

- In *advance* we opted for using the provided functional visitor
- For reference type *Move*, double dynamic dispatch revealed the underlying type of each move
- Methods *doSingleMove()* and *doDoubleMove()* are passed into visitor and called depending on underlying type, returning a new GameState
- This is shown below in figure 2

<u>Limitations:</u>

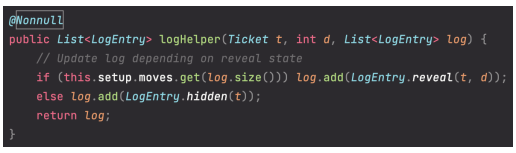
- Long constructor - Needed to validate game states
- Use of streams - Not fully within OOP principles but help simplify procedural code and allow for concise functional approach

<u>Figures 1, 2 and 3:</u>

```
// True if all detectives have no tickets
private boolean isStuck(Detective p) {
    for (Player det : this.detectives) {
        if (!det.tickets().values() ImmutableCollection<Integer>
                .stream() Stream<Integer>
                .filter(v -> v != 0)
                .collect(Collectors.toSet()) Set<Integer>
                .isEmpty())
            return false;
    }
    return true;
}

// True if mrX has no available moves
private boolean isStuck(MrX p) {
    if (this.remaining.contains(p)) {
        return (this.moves
                .stream() Stream<Move>
                .filter(x -> x.commencedBy().isMrX())
                .collect(Collectors.toSet()) Set<Move>
                .isEmpty());
    }
    return false;
}
```

```
@Nonnull
@Override
public GameState advance(Move move) {
    if(!moves.contains(move)) throw new IllegalArgumentException
    // Functions applied when visited
    Function<SingleMove, GameState> smf = (this::doSingleMove);
    Function<DoubleMove, GameState> dmf = (this::doDoubleMove);
    // Functional visitor to reveal underlying type
    Visitor MoveVisitor = new FunctionalVisitor(smf, dmf);
    return (GameState) move.accept(MoveVisitor);
}
```



```
@Nonnull
public List<LogEntry> logHelper(Ticket t, int d, List<LogEntry> log) {
    // Update log depending on reveal state
    if (this.setup.moves.get(log.size())) log.add(LogEntry.reveal(t, d));
    else log.add(LogEntry.hidden(t));
    return log;
}
```

Achievements : cw-ai

Minimax:

- MrX AI generates a game tree with alternating layers of MrX and detectives moves
- The maximum depth of this tree is 1 (MrX) + 5 (detectives) + 1 (MrX) = 6 total
- MrX plays as a maximising player and detectives minimising
- At the lowest depth the board score is evaluated and passed up the tree. At the source of the tree, for the maximum evaluation, the index in the original list of moves is returned

Dijkstras:

- Distance calculations included in *Distances.java* but not used in *pickMove()*
- Dijkstra's algorithm used Indexed Min Priority Queue to find the shortest distance between every node. List<List<Integer>> is indexed into find distances between nodes.

File Streams:

- File output stream used to write dijkstra's *List<List<Integer>> distances* to file *t.tmp*
- File input stream reads file *t.tmp* as *List<List<Integer>>* object

Convert Board:

- Method included in MyGameState copy to convert a board to MyGameState.
- Allows for finding MrX location from any state

Alpha-Beta pruning:

- Alpha-Beta pruning used to break of unnecessary parts of tree and speed up algorithm
- Stops looping over level if best evaluation could never come from remaining branches

Board Scoring:

- total – sum distance between MrX and detectives
- freedom – number of available nodes from MrX position
- closest – smallest distance between MrX and a detective
- score combines total and freedom. It is significantly decreased when the closest detective is 1 move away from MrX.
- MrX winning immediately returns 10000 and detectives -10000

Detectives AI:

- Selects random move in rounds before MrX is revealed
- Otherwise uses distances to find move that gets closest to MrX last reveal location