Report

CW-Model

The code in CW-Model has passed all 83 tests. Most exceptional circumstances are identified with the help of the TestModule. There are 15 helper functions for methods required by the interface Board. Which will be explained later in the Implementation section.

Implementation

How Available Moves Are Produced

The single moves are produced by a given player and a given ticket map. To generate a move, the function firstly iterates through the adjacent node of the location of the given player. Then it checks whether a detective has occupied the node and if the player has the required ticket. If so the move will be made by the given player, node, and required ticket, then added to the result list. Finally, the secret move will be made and added as well if the player has any secret card.

All single moves are needed to produce double moves. An intermediate destination must be one of the destinations of available single moves. Hence the function iterates through the available single moves, finds adjacent nodes of the intermediate destination, and produces a new ticket board. Next, it iterates through the adjacent nodes, passing it and the new ticket board to the CheckSingleMove function to check its feasibility. A helper function will ingrate two single moves together to produce a double move.

How to get TicketBoard in an elegant way : Lambda

The implementation of function:

```
getPlayerTickets(Piece piece)
```

require to return an instance of TicketBoard of type Optional by given piece.

Generally, this function can be accomplished by implementing an anonymous class of TicketBoard. Such as:

```
new TicketBoard() {
    @Override
    public int getCount(@Nonnull Ticket ticket) {
        return player.tickets().get(ticket);
    }
}
```

However, powerful JAVA alow us to arm ourselves with style of functional programming by passing a function as a parameter, which simpler and more efficient. After find the player object by given piece, we can use

```
ticket -> player.tickets().get(ticket);
```

to implementing the TicketBoard instead of anonymous class. Finally, return it back with wrapping of Optional.

Reflection

Almost all of our biggest struggles in the development process come from our Improvidence. When we passed a test that has been bothering us for a long time, we suddenly found that the test we could have passed did not pass. We had to stop the development process to sort if-else and loop logics. We had to review our understanding of every attribute.

CW-AI

Summary

Our AI stands on the Minimax Algorithm, which is a wildly used algorithm for turn-based games. The detective's turn is slightly handy so we use the CartesianProduct function provided by Guava to get all possible combinations of moves for their turn. The scoring function is based on Dijkstra Algorithm, which will be explained in detail later.

Scoring Function

The scoring function considers the distances from detectives to Mr.X. It also weighs each distance because the closer the detective is the more dangerous the game state will be. The function reviews the log entry too. The score will be higher if the detective sees a secret move when revealed. It also saves double moves for more emergency circumstances otherwise the AI is going to consume them at the very beginning of the game.

The general score is calculated by the following function

$$S = base - \sum_{i=0}^{n} \frac{\alpha}{d_i^2}$$

where *base* is the base score and *alpha* is an argument. The value of *alpha* is on the basis of the base score.

Minimax and Alpha-Beta Pruning

Simulation in the minimax algorithm uses a lite version of MyGameState class we implemented in CW-Model. A new instance will be initialized for each move and passed into the deeper recursion until the recursive function reaches the base. A tree will be created in this process, and a higher node will select the biggest or smallest score from its children nodes. Scores are passed from the bottom way to the top layer of nodes. Therefore, the helper function can select the correct optimal move by comparing the score of the root node and that of its children.

Limitation of AI

The biggest limitation of our AI is its efficiency. We have cut the unnecessary simulation and used Alpha-Beta Pruning. Even so the workload is still huge for a low performance laptop.

The scoring function has another important issue. It cannot analysis the density of detectives in an area. If the four detectives are at equal distances from the suspect, but two are in the east and the rest in the other direction. The AI cannot tell that the two detectives on the east are more dangerous than the others.

Reflection

We have divided the entire project into several components at the very beginning. This includes distance algorithms, scoring function, the minimax algorithm, and trees. However, instead of writing a unit test, we write the entire project all at once and then debug it. We found that this made it difficult to find and fix bugs, which slowed the overall efficiency of development. One example is that a null pointer error kept occurring because we forgot to remove empty lists when calculating Cartesian Products. We can fix bugs such as these much more quickly and easily if we run unit tests.

Some of the highlights that make us proud

The Alpha-Beta Pruning we implemented improves performance hugely by reducing unnecessary calculations. We have derived a feasible and flexible function to quantify the current GameState. It has considered not only the distances between detectives and Mr.X but also general strategies. The tree structure can be set to any depth. This means that if the performance is optimized enough, Mr. X will become invincible with a deep enough minimization algorithm.