Scotland Yard Report

Daniel Adigamov (zf23328), Mykola Takun (xq23942)

cw-model

Overview

- **All of the 83 tests have been successfully passed.**
- Implemented MyGameState fully, including declaring all attributes, completing the constructor for full initialization of game state, and implementing all getter methods.
- Implemented the advance and getAvailableMoves method successfully.
- Created several helper functions for getter methods in particular to implement and aid functionality.
- Created helper update functions to aid in the update of game state in advance, and implement functionality that would be too convoluted in a single advance method.
- Implemented functionality to determine winner.
- Implemented the **Visitor design** pattern, a behavioral pattern letting us define a new operation without changing the classes of the elements on which it operates. It was used for applying different behaviours to single and double moves based on their type, using single dispatch in this case, as the behavior is determined only by the type of move.
- Included the **Factory design pattern**, a creational pattern letting us defer instantiation to subclasses for different implementations of TicketBoard for MrX and detectives. When a player is initialized, the corresponding ticket board factory is used to create the appropriate type of ticket board based on the player type. Each player is then provided with their respective ticket board instance, allowing them to interact with and use tickets during the game. It allows for the creation of objects without specifying their concrete classes, enabling flexibility and enhancing code readability.
- Implemented **Observer design pattern**, a behavioral pattern used to notify subscribed observers based on event changes such as game over or move made.
- **Upholding encapsulation:** we kept encapsulation in the project in several ways, this includes setting all the MyGameState attributes as private for data hiding, implementing getter and setter functions for access control, and setting classes

and methods to private, when possible, to hide unnecessary implementation details for information hiding.

Reflection and Limitations:

As with any project, there are several improvements that could be made, such as:

- Creating more files for classes to create more clean, readable and modular code.
- Creating more files for separate classes for organization, due to an abundance of helper methods, in turn making the final MyGameStateFactory file long and hard to navigate.
- Although several design patterns were used (visitor, observer and factory), more design patterns could be used in order to reduce code repetition and create a good foundation for further additions and implementations.
- Although the code was well-commented, efforts could be made to improve code readability further by adhering to consistent naming conventions and reducing code complexity where possible.

cw-ai

Overview

- **Dijkstra** path finding algorithm implemented to find the shortest path between MrX and different detectives.
- **Minimax** algorithm implemented for MrX artificial intelligence, including alpha and beta pruning.
- **Scoring** method was successfully implemented to return the score of the board (the distance to the closest detective), where detectives try to minimize this, and MrX tries to maximize this to get away from the detectives.
- **Pickmove** method successfully implemented to pick the move with the best board score (ideally this is the optimal move). Each time the function is called, a new game state is created so that we can use the advanced method.
- **Visitor Design pattern** implemented for visiting moves based on their type, whether they are a single or double move.
- **Optimizations:** at first, the code was quite inefficient and the MrX ai would take a very long time to make a decision on what move to make. Once optimization we made was to create two different methods of Dijkstra path finding algorithm, one

finds the distance to the first detective it encounters, meanwhile the other finds the distance of every single detective, and returns the smallest one, the latter takes a lot more time and its only used when we are trying to find the distance to a specific detective set as the target. While the optimized one is used in getScore.

- Another optimization we made, used in minimax and implemented in the detectiveMoveCombination, instead of considering every possible move that each detective could take, we choose only those that are the worst-case scenario for MrX and get closer to him, and take only three of those moves, because regardless of the combinations found, all of them are going to get closer to MrX. This saves us from a lot of needless processing and usage of computer resources in the program.
- At first, MrX would only choose double moves and use up all his double move tickets at the start, as they were marked as the most optimal with the highest board score as they moved the furthest distance from the detectives, we fixed this by MrX ai only consider double moves if any detective can catch MrX in their next move, ensuring that the ai does not waste all the double move tickets and only uses them when necessary. This also turned out to be a massive optimization for performance and the time taken for MrX to decide his move.

Reflection and Limitations

- Certain areas could be improved, such as optimizing the Dijkstra path finding algorithm by replacing it with A* for faster performance.
- Due to time constraints, certain implementations could not be realized, such as a working ai for the detectives. As well as more efficient and smart use of MrX's tickets (as for the meantime MrX mostly only uses taxi tickets).
- More elaborate and detailed comments and documentation could've been included, as well as more modular and well-organized code to improve code readability and scalability. As well as more consistent naming conventions.
- There were some bugs with double moves where MrX would incorrectly choose double moves either in the wrong scenario, or double moves that end up in the same place, due to time constraints we were not able to fix this.
- At first, we were able to implement correct use of secret moves, so that MrX is more likely to use them after a reveal move, so that the detectives have a harder time guessing where he went, however, after some further development there were too many bugs related to double moves, and we had to backtrack development and we

were not able to reimplement correct use of secret moves, again due to time constraints.

- We considered implementing the **Decorator** design pattern to build upon the Move class and add extra functionality (a score attribute to store score so that it was directly linked to every possible move), however this was deemed redundant further into implementation of the ai.