What We've Done

In this project, we have been working to complete the implementation of a digital version of the Scotland Yard board game, and to implement a simple AI that can select a valid move that is available to it.

The first part of this project involved the completion of a class in the Scotland Yard project to enable the game to run correctly. The class completed was the ScotlandYardModel class which stored and handled the games current state, the state of all the players in the game, providing notifications on events in the game to spectator objects, and general game logic.

To ensure that we had completed this part of the project correctly, we were provided with an implementation guide including general expected behaviours of the game, UML sequence diagrams that showed the order in which certain operations in the program should occur, and a series of tests that could be run on our implementation to ensure everything we created worked as it was meant to.

To actually achieve the provided goals we had to implement a number of different methods including: a constructor method to safely initialise the game model with valid data, a series of getter methods that returned either values or immutable data structures containing data used in the class, methods to handle aspects of the games logic such as win conditions and finding valid moves, and also methods to handle the progression of a game round allowing all players in the game to submit their moves. Also necessary to the completion of this part of the project was the use of some classes making use of the visitor design pattern to allow a simple way to process the different kinds of moves that could be provided by a player. In our implementation we made use of two visitor objects: a visitor to process the moves, and a visitor to hide the details of a move before notifying all of the model's spectators as to what move had been made.

The second part of the project involved the creation of an AI for MrX so that when playing against detectives the AI will be able to look ahead a few moves and select a move that can be considered good.

This is accomplished through the implementation of a minimax algorithm with alpha-beta pruning to speed up the building of the game tree. The MrXAI class will return a MyPlayer object when createPlayer is called on it, and said object handles the move deciding logic. Upon first being called the class will initialize several variables based on various factors and build a Map which stores the distance from each node of the game map to every other node. This is implemented as a map of maps, where the parent map uses each node as a key, and the child map uses every other node as its key and the distance between the parent node and each other node as the values. This map is built upon the first makeMove call to the MyPlayer object and stored so as to avoid redundant rebuilding.

The actual decision making is entirely handled by the minimax function. It is called with an AIBoard (a helper data storage class created for the purpose which stores a board state which also holds reference to AIPlayer objects, created for a similar purpose), a current tree depth, and alpha and beta values. It will initially be called by makeMove, and thus the first call passes in the current board state. From there, it makes a recursive call to itself with a board state updated to reflect every possible valid move for the current player, up until the specified tree depth is reached. The tree depth is set to the number of players + 1, such that the function will consider the possibilities of the move after Mr. X's next one regardless of the number of players, making it a significant

improvement of our initial implementation of a look-ahead-one AI. A higher tree depth leads to too significant of a slowdown for viable play, and thus was avoided.

Achievements

Over the course of this project we feel that we have generally improved as programmers, but also feel that there are aspects of the project that we feel were implemented well.

In the first part of the project we feel that there are a number of areas where either the introduction of new concepts has improved our code, or we are happy with how we eventually implemented a particular method.

In the first part of the project, there are two new concepts that we have made use of that we feel have drastically improved our code from what it would have been without these concepts. The first concept being functional operations to handle the processing of certain data sets in our program, this has in the cases where it has been used allowed the shortening of some of the code and allowed us to use the process of function composition to simplify the method getting the results we require. While the benefits of using this concept were not quite as large as the other concept we used, we still feel that it has improved our code nonetheless. The second concept we made use of was design patterns, in particular the visitor design pattern which made the process of handling the three different types of possible moves far simpler than it initially seemed it would be. Using this design pattern, instead of needing to try and treat moves entirely generically, or down casting, we could instead pass our visitor objects to the moves visit method and easily be able to process the particular type of move we received.

Also, we are happy with two methods in this part of the project in particular, the method to get all valid moves for a player, and the method to accept a move for the player. The method to get valid moves I feel we have designed in a fairly simple way that retrieves all the valid moves for any player it is provided with, and also allowed a fairly easy way to add the functionality to get the special moves that are only available to MrX. We are also happy with the movement accept method, as through many iterations of the design of this method we feel that our current version is concise and fairly easy to follow in its basic flow, I believe that this is primarily because of how we separated each step of accepting a move from a player into a separate function.

In the second part of our project, we were provided the opportunity to complete the task using a variety of different methods, and we are quite happy with our implementation of the method we chose to use to complete this part of the task.

To implement the AI, we first created a scoring function, that primarily scores moves based on the distance of MrX from the detectives if the proposed move is made, this is then used in a minimax algorithm to determine the best move for MrX to take next. We think that we have implemented both the scoring algorithm and the minimax algorithm quite well. In the scoring algorithm, in order to speed up processing, the distance from any given space to any other space has been pre-processed and can just be accessed immediately speeding up processing. In addition to this we also feel that our minimax algorithm is implemented quite well, as it has been designed so that it will only generate and search a game-tree of a desired depth. This could allow for future optimisation, and also implements techniques to allow for alpha beta pruning which, in a game like Scotland Yard with so many possible moves, is quite important as it can save a substantial amount of run-time as the AI selects their new move.

Critical Reflection

Looking back over the course of the project however, we can see ways in which our end product could have been improved, both in terms of the final code, and in terms of our workflow throughout.

In terms of our workflow, we feel that there were points throughout development where, when faced with a difficult task, we would simply be coding a solution, taking no time to plan our code out. This usually resulted in us having to re-do at least parts of the written code later on, taking time that could have been used on other aspects of the system.

We also feel that in the first part of the project, in the move accept method, it can become difficult to follow the logic of the code when trying to consider how more complex moves, such as double moves, work in the system. In the future, if this code were to be maintained or altered, while the basic move logic is easy to follow, altering the logic for the double move method could result in issues that the person looking at the code may not have expected. This is an example of an issue that would have been solved if we had placed more time into the planning stages of the project.

There are some aspects we also would have liked to improve with regards to our AI but could not do during the time allotted for the project. While the AI will select a move quite quickly, this is mainly due to a limitation we had to implement which only allows the AI to look one move ahead, as otherwise the AI would begin to take a substantial amount of time to select a move. If we were to try and improve the abilities of our AI to look ahead further ahead in the game, we would need to improve the runtime of our minimax algorithm. In this case, if we were to try and achieve this improvement in run-time we could have potentially used a tool typically used by chess AIs in the form of transposition tables, which would allow the system to remember when board states have been previously generated by the minimax algorithm. This allows for time to be saved during game tree generation, as the program can simply lookup pre-generated board state scores rather than calculating them from scratch.

On testing it also seems that our AI will tend to stick to a similar area as it moves. unless the detectives start approaching. This leads to the AI wasting moves it could have spent moving away from its last known location. The scoring algorithm in this case may mean that occasionally moves are judged in such a way that a cycle begins to occur. To avoid this in future

We also could have made use of parallel computation in our AI. This would allow the game tree generation to take advantage of additional processors, speeding up the program and allowing for a greater tree depth to be reached in the same amount of time. Were we to have implemented this, we would have gone with an approach wherein each processor simply explores a different subtree of the game tree, starting from the same root node. However, this fairly basic approach to parallel searching would probably have not saved a huge amount of time.