Scotland Yard Model

Our first step was to set up the initial game state in class ScotlandYardModel(). This mainly involved creating the player list and running tests to ensure that all players had the necessary tickets, no two players had the same starting locations, and so on.

Finding Valid Moves

The main portion of validMoves() looks at all paths leaving the node occupied by the current player. It then looks to see if the destinations of those paths are unoccupied, and if the current player has the necessary tickets to take that route. If both of these are true, a TicketMove object is created with the player, the ticket and the destination as parameters. The moves are added to a hash set which is returned. If no moves were possible and valid then a pass move is simply returned.

Before returning the set, we check to see if any double moves are possible. This is only possible if there are at least two rounds left in the game, and that the player has a double move ticket - also inferring that the player is MrX. We loop through the set of valid moves that we just created and we find all of the subsequent possible moves that could follow. The second move and its corresponding first move are put into a new double move object, which is added to the set.

Accepting Moves

The first action our accept() method takes is to check that the move is valid, using the model's attribute 'Set<Move> validMoves'. We store the valid moves in this attribute to save us from having to call the validMoves() method as frequently, which would be more expensive.

We then check if the move is a double move. In this case we separate the move into two ticket moves, alter the destinations of those ticket moves depending on whether it is a reveal round and whether the next round is a reveal round, and then we accept each move separately. The first move is recursively passed into a second run-through of accept(), but with the attribute 'boolean doubling' set as True so that certain features are bypassed. Once the first move is processed, 'move' of type Move which holds a double move is reallocated to the second ticket move. accept() continues on and evaluates the second move as normal.

The next check is to see if the move is an instance of ticket move. This will conveniently trigger for the second move of the double move, as it would any ticket move. The following code implements the effect of the ticket, changing the player's location and transferring the used ticket to MrX.

The move has been made and a new stage of the game begins. If the move was the first move of the round then we increment the round and notify the spectators. We notify the spectators that the move has been made and - if the game isn't yet over - we rotate to the next player, telling the spectators that the round is over if necessary.

We wanted to implement visitor patterns in our accept() method in order to reduce its size and to improve modularity. In the end it proved to be unusable as visitor patterns use dynamic dispatch to separate the different ticket types. This means that when visit() receives the a double move, it has type DoubleMove rather than Move. This caused problems in our model as we recursively call accept() on the first ticket move in the double, and then continued with the current

instance of accept() but with the second ticket move as 'move'. With 'move' having type DoubleMove in the visitor pattern, the reference could not be reallocated.

Identifying a Game Over

There are various ways the game can end, and so this function essentially checks each of them one after another, and then if one of the checks is true it notifies the spectators that the game is over.

AI Development

MrX AI

To begin with we started working on a scoring system, this allowed us to decide whether the game board was good for MrX. We decided that the scores will be based on two different factors.

Firstly, we look how many routes there are from MrX's position. When we explore this, we value edges that can take you further more than others; as such we value boats at 40, trains at 8, bus at 3 and taxis at 1. If MrX cannot leave the node we score this as -9998 to stop MrX trapping himself.

Secondly, we look at how far the detectives are away. We work out an average distance from the detectives and combine this with the edges score in a weighted manner to create the overall score. The distances from the detectives are more heavily weighted as we believe these are more important. During this step if any detective is on top of MrX or just one away we score this very negatively and this indicates MrX has lost or will lose soon. To work out distances we needed to use Dijkstra's algorithm. We did this in a separate class, meaning we could create an object and call it wherever we needed it.

Following this we started to develop versions of our AI; the first being a system which just scored all the moves passed in and chose the best one. We expanded this by looking x moves ahead and choosing the move which led to the best location in 3 moves time. All these versions took no regard of where the detectives moved. We kept versioning and compiling regularly so we always would have a running version we could test. This meant as we made it more complex we always knew that the basics worked.

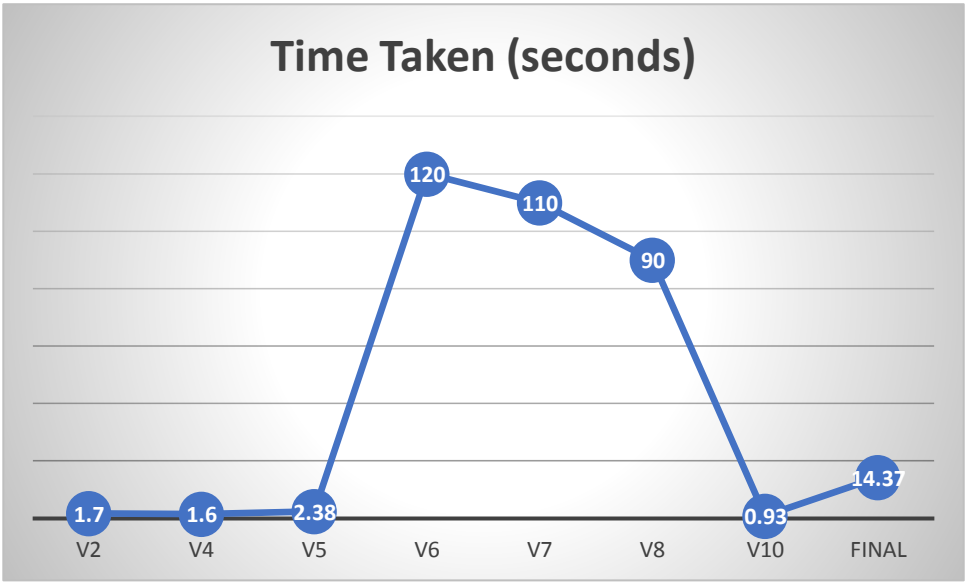
Next came the task of building the tree. We designed the 'MoveNode' class which contained a link to previous nodes and a list of all the nodes which followed it. It also contained the move that was just made. From this we built a tree of nodes representing what could happen. We quickly realised our MoveNode wasn't sufficient which led to us developing 'DataNode', this was the same as MoveNode but had the feature of storing a list of PlayerData as well. PlayerData was another class we had made up, similar to ScotlandYardPlayer. It allowed us to store players locations, colour and tickets. This meant as we moved through the tree we could keep track of everything and everyone's tickets.

Having built the tree we moved onto minimaxing. We developed an algorithm which scored the end nodes and then worked back to the start node of the tree giving us the best possible path. At this point we realised running our AI at any depth over 2 caused it to take way over the 15 seconds allotted. We moved onto adding alphabeta pruning to see if this would solve it. This was quicker than minimaxing but the time was still too long due to the time spent building the tree.

At this point we redesigned our tree-building code so it only does one node at a time. This meant we could call it from our alphabeta method meaning we wouldn't waste time building parts of the tree which had been pruned off and deemed to be bad routes. This greatly improved speed but still wasn't good enough. It was at this point we realised we couldn't include all possibilities in our tree. Hence we limited it to only including detective moves that were staying the same distance or moving one closer to MrX and MrX moves that had a score of 0 or above. This meant no moves where he went next to a detective would be included as these would be near certain death.

Now we have designed a system where our code tries to build the game tree to the end of the game. But after 14 seconds it returns the score of the node it is on so that the AI always makes the decision within the 15 second time limit. We believe this is more dynamic and makes use of all the available time. Also it obfuscates what our AI is doing/ where he is, as we have noticed when an AI is quicker it means it has fewer options allowing people to work out where it is more easily.

Throughout the development of the MrX AI we were developing a detective AI, this allowed us to run them against each other quickly seeing how they were behaving and what we needed to work on.



This graph shows the times taken for each version of the AI. They each were tested with depth 3 (where applicable) and with 3 players at the same starting locations every time. The significant increase was when the detective's moves were introduced into the game tree. After that the code was repeatedly optimised. The large decrease to V10 corresponds to only considering detective moves towards MrX. The final version runs in between 14-14.5 seconds every time regardless of initial setup as it tries to go as deep as it can in the allotted time.

Detective AI

The first iteration of our detective AI was very simple. It looked at the most recent known location of MrX – called 'lastKnownMrX' - then used Dijkstras to calculate distances from that location. It then loops through all of the detective's valid moves and chooses the move which will result in the detective being closer to MrX. Though potentially enough to tackle a new player, this is not sophisticated enough to run against a MrX AI.

The next iteration involved the addition of a custom spectator which we called 'MrXFinder'. When it received a makeMove() call from MrX it would look at the ticket he used and calculate a list of locations he could now be in. We then used this to calculate the scores of the detective's next moves by finding an average distance from each of the possible locations of MrX.

This was a lot better but the detectives still seemed a little clueless, particularly in the first few moves where MrX's location was not known at all and the detectives were just taking random moves. Our next improvement to the AI was to make the detectives head towards nodes with good transport links when they have nowhere else to go. This means that when MrX does become visible they are faster to get to him. We valued taxi edges at 5, bus edges at 7 and underground edges at 25, as they allow MrX to travel the furthest.

A problem we noticed with the AI was that they would start to trap MrX but then let him slip. This was because once a detective had occupied a potential MrX location, the location wasn't being removed from the list, meaning that the detectives were moving around the map quite blindly. By removing a node each time it is landed on by a detective, the possible locations MrX could be in drastically decreased, and the detectives became much better at surrounding MrX.

Another change which improved the AI's ability to surround MrX was an alteration in the move choice. We added an exception where if a move will land the detective on a possible MrX location, the detective should definitely take that route, rather than going to a node which is good on average. This is not only a chance to catch MrX but will make the moves of the next detectives more accurate, as they have less possibilities to aim at.

We tried implementing a different scoring system where the detective simply went towards the nearest possible MrX location, however a few rounds after a reveal round these locations were so spread apart that the detectives diverted away from each other and became ineffective.

Reflections

If we were to implement another detective AI I think I would have utilised a game tree, even only to three depths. I think this because our detective is most weak when MrX takes a boat, as it simply can't keep up. The detectives chase along the slow taxi routes, only for MrX to then take another boat and make the detectives quite powerless. A game tree would be able to predict this behaviour and the detectives would be move more smartly to catch him at either end.

If we were to do the MrX AI again, we would research into where the time is being spent especially during the tree building process. This would allow us to consider more nodes and do more computations in the time, ultimately allowing us to perform better.