What has been done?

CW-MODEL – Completed all the tests. Double dispatch used during advance for dealing with single vs double moves when move given (via anonymous inner classes) and observer model implemented in the model factory. The only standalone visitor is the final destination visitor as this is also used elsewhere in the AI. The game state factory contains many, self-explanatory, helper functions, namely: the single and double move set creators; giving the set of winners; determining whether a player can move and a piece's player; and getting updated player and MrX tickets, positions and the travel log.

OVERALL for AI: Summary of added/changed classes and what they do: **Clive** – controls the AI and make the call for getting best move, **DijkstraShortestPath** – gets the shortest path from one given node to all other nodes as an array (based on the graph given by board). **EvaluatePlayers** – an interface that for player evaluation classes to follow; requires that classes implement and evaluate function. **EvaluateCloseIn** – provides evaluation for moves and state based on logic for players trying to capture mrX. **EvaluatePrepare** – provides evaluation for moves and state based on logic for players preparing for mrX's next reveal. **EvaluateTickets** – a class containing static methods for getting changes to evaluation based on logic surrounding tickets. **FinalDestinationVisitor** – an implementation of move visitor for getting the final location of a move. **MiniMaxAlphaBeta** – runs the minimax algorithm with alpha beta pruning up to a specified depth to calculate the best move for mrX in conjunction with an evaluate class. **PlayerMoves** - deal with getting the best move for the detectives, including what type of evaluation to use. **PossibleLocationVectorCalculator –** a class with static methods to calculate the possible locations of mrX or detectives using adjacency matrices and vectors. **RunDSP** – a class with static methods for different ways we may want to call and utilise the shortest path class (abstracts Dijkstra away from evaluation classes for simplicity). **SMVP –** a static method used to calculate the product of a sparse matrix given as an adjacency list and a vector. **Stephen** - controls the detectives and make the call for getting best move. **UpdateableTicketBoard –** implementing ticketboard interface, a map holding ticket counts with getCount() and an additional update tickets function. Note also that there is an added dependency in the pom.xml for AI to allow access to the FinalDestinationVisitor class. **NodeType** – has a static functions to encourage AI to go to nodes with good access around.

CW-AI for MrX (Clive) – to determine the best move, the ai will create an instance of the minimax class. When get best move is then called, the algorithm will be run at a depth given by how many players there are, once the depth is reached the evaluate function in the evaluation class is called. Additionally, at each step evaluation for special moves made and tickets used is added. These heuristic evaluations are based on the distance from players, which tickets have been used and when good times to play double and secret moves are, we also check if he has won or lost. These provide a final double value that is passed back to the mini-max algorithm for evaluation of that branch. The distance of players to MrX is done using a Dijkstra's shortest path algorithm with MrX's location as the starting node.

CW-AI for Players (Stephen) – to determine the best move, first an instance of the possible locations class will be made. The goal of this class is to give the possible location of MrX at any given point via the move log. This is done by treating the Scotland yard graph for each transport mode separately and as an adjacency list (which can then be thought of as an adjacency matrix). Taking a vector where MrX's last location is 1 and all other nodes are 0, we can then run matrix-vector multiplication with said vector and the further transport he has taken (or the full graph if secret) to determine where he could be currently. (If he has not been revealed once yet, then players are instead instructed to disperse and go to nodes far from other players that have all three types of transport). Additional heuristic evaluation points are added/subtracted for the player getting stuck and using tickets they are low on. [Worth noting here we don't need to check if mrX gets stuck as this is a loss – already checked]. The players then moved based on a selected subset of the overall tree of where mrX could be (if they are close) or head towards the tree if far. Other evaluations of the state and move take place which are mentioned in achievements

Reflecting on Achievements

OVERALL – We feel that we have aptly understood the essence of OOP and used classes and objects appropriately. Made use of the double dispatch design pattern well, implementing standalone and anonymous inner classes where appropriate (all anonymous inner classes held within methods, unless needed elsewhere). Well commented and easily readable code.

CW-MODEL – put into practice techniques and design pattern discussed in lectures throughout tb2, albeit from the instructions given.

CW-AI for MrX (Clive) – Originally, we were making a game state to use the advance(move) function, replaced this for only keeping the necessary information. As for the tickets, we create a simple class implementing ticketboard that can also be updated. Implemented a working: mini max algorithm with working alpha-beta pruning to eliminate branches that are both too good for opponent and not good enough for MrX so that we may explore more depth; Dijkstra's shortest path algorithm – implemented/adapted in the appropriate way to allow it to work for the way in which the graph is given and how we would like out evaluation class/function to work; heuristic evaluation function which includes distance of players; tickets when he is low on them; when MrX should use a double move and secret moves.; staying on high degree nodes to increase chances of escape.

CW-AI for Players (Stephen) – determined, efficiently where MrX could be by treating each transport mode's adjacency list as an adjacency matrix. Since we know that this graph represented by a matrix will be sparse (obvious that nodes only connect to a very small number of other nodes)., implemented fast matrix vector multiplication for sparse matrices. Separated the logic into the two main parts as a human playing would: spread resources vs trying to corner. We stop the players from accidently allowing themselves to become stuck due to lack of tickets. We make it such that players keep tickets if possible, using the best ticket to get to the desired node too. Sorted out a way for players to stop choosing that best moves are to stay in the centre of the possible moves for mrX: by using a mathematical log to evalulate that even If the total distance from possible nodes is the same, the AI favours being close to some nodes and further from the rest as opposed to trying to stay central. As for preparing, we make it such that the AI is trying to get to a node with a bus and train station the move before mrX is revealed. This is done with the help of the possible moves class so we can see if we have the required type of node in the correct number of moves (only applicable at the start as the rest of the time we only have one move to go so we go straight there).

Reflecting on Limitations

OVERALL – The biggest and most important shortcoming is that overall, neither of the AIs play close to the level at which a human player could, even despite mrX having greater move tree analysis capabilities – this is very disappointing. Other less important limitations are: Lacking optimisation of code. Did not investigate parallelism for the Ais and we felt that monte carlo tree search could have been a good option for the Ais but did not seriously investigate implementation for these due to time. On top of this, our testing did not have a huge amount of structure and relied on print statements and playing games. We had look at the project structure and there was no tests folder so we assumed testing should be done like this, however in hind sight, perhaps we should have looked into concrete testing some of the classes/ evaluation functions.

CW-MODEL – did not spend much time looking for the most optimal ways to pass the tests and gain the needed functionality. Once tests passed we moved to the AI.

CW-AI for MrX (Clive) – lots of room for improvement given time. The following heuristics could have been implemented: not getting cornered (as it's own evaluation or by increasing depth via optimisations), making seemingly suboptimal moves to lead players off of your 'scent' (this is a very human thing to do that seems to work well during actual gameplay). Should add a timeout option to return best move calculated so far instead of being safe with low depth value (the only way we could think to do this would be to rework the minimax algorithm to work in a breadth search esque way – sorting evaluations for trees that should have their depth taken further, otherwise some moves are not considered at all if time runs out). As we added a lot of evaluating, which is not necessarily optimised, we were forced to reduce the depths. He seems to use his 2x moves prematurely sometimes, this was to fix an issue of him being caught due to lack of look ahead when just when 2x moves when it was his only option to escape.

CW-AI for Players (Stephen) – Didn't play enough Scotland yard to come up with a concrete strategy for the players, making it hard to know when they should disperse to increases their chance and when to try and corner to increase their chances (if these are even correct strategies in themselves). Again, heuristics that could have been implemented: trying to force MrX into low edge density part of the graph by forming a 'net' around him (pushing him to and edge or corner of the board). Considering the order in which to play moves to optimise for all pieces. Could have used mrX's past data to influence the values in the possible locations vector. Players can still lose by getting stuck due to lack of look ahead.