Summary

Approach

We dedicated 9 hours per week to work on the coursework (in addition to the 3 hours spent together in OOP workshops). Work was done together either by task delegation or, where possible, programming together as a single mind – the latter approach was more commonly used during debugging. In implementing individual-delegated methods, we kept each other up to speed so we could understand and use them effectively. Git was an integral part of this process, where we used a feature branching workflow.

Closed Task

We made a static inner class that collected validation methods for *Detectives* in the *GameState* constructor, as they all required the list of *Detectives* to be looped over. This common behaviour gave rise to a strategy-like pattern where a predicate was passed into one of two generic looping methods in this class.

We used an anonymous visitor in *advance()* to distinguish between single and double moves, which returned the new *GameState*. The use of the visitor pattern in this situation is advantageous because it allows us to deal with both Double and Single Moves for advance() without changing their structure as they are both polymorphic to the class Move.

To deal with single moves, we checked if the move's destination was a successor of its source in order to validate the move, then the code gets the ticket used in the move. We made mutable copies of what we needed to edit for the new *GameState*; updating the player's tickets, log (if required) and *remaining*. To update *remaining*, we wrote a *nextRemaining()* method that, when handed the current *remaining* and the *piece* taking the move, returns the next state of remaining in accordance with the next round.

When advancing Mr X using a double move it is important to check that Mr X has enough tickets to fully complete a move. As a result, we found that simply using the ImmutableSet of tickets using player.tickets() would not suffice. We therefore decided to create a mutable copy of these tickets that removed the corresponding ticket when half of a double move had been built to assert that there were enough tickets for the double move.

One oddity of the structure of this model was that *Piece* was given where *Player* was needed. In response to this we wrote *getPlayerOnPiece()* which would allow us to retrieve a *Player* given its *Piece*.

With *makeSingleMoves()* we used a stream to test which nodes were occupied. For an unoccupied node, we test for all modes of transport; that the required ticket isn't a *Double*, and that the player both has the given type (in the case of checking a *Detective* on transport *Ferry*) and at least one of said type. Beyond this loop we check if the player has secret tickets, and at least one of them. We then add a secret version of this move to available moves.

Open Task

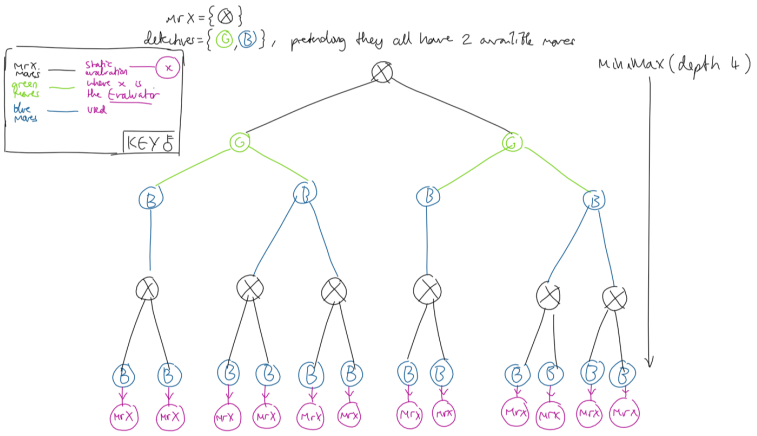
Our efforts towards the open task resulted in AI evaluators for both Mr X and the detectives with score functions that give static evaluations of the board. We used heuristics based on distances between Mr X and the detectives and the number of moves available. Our AI is based on a recursive minimax algorithm and we used Dijkstra's algorithm to measure the distances between Mr X and the detectives.

Gametrees and Minimax

The overloading minimax handler method *minimax()* takes a *depth* (corresponding to the depth of the tree) and a *GameState*. Within this handler, an order of *Turn*s are generated. These *Turn* objects contain the *Piece* that plays the turn and the state of *remaining* of the board after their turn. The first attribute is used during tree generation, and the second helps generate successive random turns such that they follow the game's structure/rounds.

Having generated an order up to the desired *depth*, the private overloaded recursive method *minimax()* is called with initial values for *depth, alpha* and *beta*, *GameState,* and the list of *Turns*: *order.* This recursive call stack is the heart of the AI's decision. We decided to use recursion with no underlying data structure because it offers reduced memory usage.

As an overview, and explained using a tree as an analogy, the root begins as the first *Turn* of *order* (whom we're choosing the move for). All the moves a player can take are filtered from the list of available moves on the board. These available moves correspond to new branches, which are passed the new board (having advanced according to its respective move), the new depth to index *order* with*, alpha* and *beta, branchID* (simply the index of the for loop which the branch is created on), and the list of the latest *Moves* available to the player we're computing for (whose go it is on *pickMove()*) called *myMoves*.



BranchID is required because once we reach ample depth we need to complete a static evaluation. Objects extending *Evaluator* (evaluators) outline evaluation strategies; these abstract classes have a *score* method which takes a set of *Moves*, among other things. *BranchID* is used to select from *myMoves* to decide which *Move* to retrieve the *Piece*'s destination from during static evaluation. Use of the strategy pattern allowed us to outline the core functionality (the *score* method) of each evaluator while allowing concrete implementations to differ in their approach of static evaluation.
Within Mr X's evaluator, we use the visitor pattern to return the destination of any moves, single or double. Our *DestinationVisitor* inner class visits the move and returns *destination* if it's a single move, or *destination2* if it's a double.

Alpha-beta pruning is done as the tree is generated from left to right; *alpha* and *beta* are set via backpropagation of evaluations, and branch-generating loops stop if all remaining branches can be pruned.

Evaluators and Heuristics

When designing the Mr X Evaluator for the AI, a key discussion point was handling all of the detectives' distances when statically evaluating a move.

As part of the heuristics for both AI evaluators, we decided to include a distance score which is calculated using Dijkstra's algorithm, which calculates the distance between Mr X and the detectives.

When implementing the algorithm for our SY AI, we had two working versions, a standard min PQ implementation and a Java Collections Dictionary implementation. As timing was critical to the AI, we benchmarked both implementations and found that using a dictionary was significantly faster than the min PQ implementation and believed it was justified to opt for this approach.

The dictionary data structure still has an O(1) node access time and searching for the next node to evaluate (the current shortest distance) has a $\Theta(n)$ search time, whereas heapsort is typically O(nlog n). Our dijkstra implementation also only checks the adjacent node of the current node being evaluated, granted that node has not been fully visited. Again, this saves unnecessary computation.

Our initial choice was to simply take an average of the distances calculated. However, the disadvantage of taking an average of all distances is that further away detectives skew the results of the average. Thus taking an average did not prioritise extremely close detectives.In addition, we wanted the Mr X AI to still be able to dart away from more than one detective at a time. Our solution was to use basic statistics to improve the average distance evaluated.

Thus, we included the method *cumulativeDistance()* that calculated a standard deviation from all the distances and then created a new dataset that only contained the distances 2 standard deviations from the closest detective to Mr X. This allows the Mr X Evaluator to hyperfocus on its closest threats rather than detectives further away. Of course, the disadvantage of this approach is that there is a loss of information; however, this was a necessary compromise given the structure of our minimax algorithm.

One distinct merit in our AI was the notion of a Mr X boundary for the detective evaluator. The idea was that the boundary is initialised to the whole graph and narrowed/updated every time Mr X revealed himself. This allowed the detective evaluator to have an approximate idea about where Mr X is, even after he has been revealed. The one minor drawback of the approach was that the detective evaluator picks a random location from this boundary so the detectives have a broader common goal (where they believe Mr X to be) rather than a sole agreed location between reveal moves between reveal moves. Moreover, picking a move randomly from the pool made it more difficult to test the Detective Evaluator as moves weren't deterministic. It is arguable, though, that this discrepancy is diminishing as upon testing the AI the detectives are successfully able to attempt to corner Mr X and one could say that by not having a sole location, it prevents detectives from aiming to move to the same spot, thus optimising their moves.

Testing MiniMaxBox

Testing *MiniMaxBox* was challenging, and led to us implementing our own n-ary tree data structure to store the minimax tree.
For this we made the class *DoubleTree* and its accompanying *Node* class. A *Node* has the attributes *value* and *branches* (int and list of branch nodes respectively). Methods for adding nodes, removing, clearing/pruning the tree and pretty-printing the tree for holistic/visual debugging were also included. A new tree is inserted into a MiniMaxBox as a variadic argument, so it only ever populates the tree when it's given one (in the interest of performance). *DoubleTree* also contains an *equals()* method, which we used to test specific scenarios' minimax trees for alpha-beta pruning and min-maxing at given levels.

A limitation of this is the somewhat high coupling of *DoubleTree* and *MiniMaxBox*. Despite minimising this, 7 lines were inserted in *minimax()* which ask whether there is a *testTree*() to add to (and if so, add to it).

```
DoubleTree prunedTree = new DoubleTree(
        new Node( value: 1, Arrays.asList(
                new Node( value: 1, Arrays.asList(
                        new Node( value: 1),
                        new Node( value: 1)
                )),
                new Node( value: 1, Arrays.asList(
                        new Node( value: 1)
                ))
        )));
```

Defining a *DoubleTree* can easily be made to look exactly as its pretty-print for easy debugging.

```
[1.0]
  ├[1.0]
  │  ├[1.0]
  │  └[1.0]
  └[1.0]
     └[1.0]
```

Final Comments

Where possible, helper methods, which do not need to be known by other classes, are made private. This follows encapsulation, unless it needs to be public for testing, which promotes the idea of defensive programming. These situations were very rare and an example instance was getting the Mr X boundary from the detective evaluator to make sure behaviour was correct.

*Evaluator*s are passed weights when they're created which multiply the effect of each heuristic in the compound output of a static evaluation. A future improvement/something we hadn't the time to implement was an automated, headless test system which would quantify the performance produced by a given set of *Evaluator* weights on a given move. We would then research an algorithm which would analyse this data and intelligently tweak the weights to optimise a given move from the game state.

It's important to note that upon observing the AI, Mr X uses Taxi tickets most of the time. There are a few reasons for this. One, taxis are the most common mode of transport across the graph, which therefore means by choosing a taxi, naturally there will be a higher availability of moves. Secondly, when there are 5 detectives in the game, their joint effort is to get to Mr X as closely as possible -between reveal moves. In order to do this, detetives make use of underground and bus tickets to get as far across the board as possible, thus limiting the number of bus and underground stations Mr X can use.