# COMS10017 – SCOTLAND YARD PROJECT

Patrick Lee, Himanshu Bhandari

**Note – for some reason, I can't build the project from command line using maven. I get compiler errors even installing the entire project file fresh. However, both projects compile and run fine in IntelliJ and cw-model passes all tests.*

## PART 1 – MODEL

### Evaluation

Our implementation was completed successfully, passing all 83 of the given tests and making use of a variety of techniques and OOP concepts learned in the past few months. This was extended further in the AI task.

## PART 2 – AI

### State and Action Classes

We decided to create our own implementations of Board and Move, 'State' and 'Action' such that these classes could also contain scores and relationships between moves and resulting positions. These classes both implement in the built in 'Comparable" interface allowing them to be sorted by score. Initially, we thought about having these classes inherit from 'Board' and 'Move' so as to basically be able to extend those classes with the extra data but decided against that as we wanted to create completely separate objects in their own right. The State and Action classes have attributes and methods to deal with things like: scoring, the 'Board' and 'Move' instances which they relate to (on a one to one basis), a uniquely generated key (for the State – explained further down) and an object reference to the next state (for the Action).

We also created a data structure called an 'Action Set' which is able to store a list of actions, sorted by the score of the state they lead to. This is sorted in descending order if maximizing MrX and ascending otherwise.

These classes aim to encapsulate all necessary attributes and methods required for a given AI agent to reason about the game in terms of 'Actions' and their resulting 'States', in a similar fashion to an API, as well as create cleaner 'MyAI' class where the focus can be on the AI logic itself as opposed to game data.

### Scoring Function

We implemented a basic scoring function which would provide a rating for each move explored in the game tree as well as giving a base case when either the tree has reached a maximum depth or a winning/losing position is found. In the case of a MrX win, the score for the board configuration would be 100, and -100 in the case of a loss. Otherwise, the score would be the sum of the average distance between MrX and the detectives, found using Dijkstra's algorithm, and the number of moves available to him (this would be subtracted if a detective move was being evaluated).

### Minimax Algorithm

Due to the turn-based and perfect information (from MrX's perspective) nature of Scotland Yard, we felt an algorithm based on MiniMax would be a good fit, seeking to maximise the score of a move when processing MrX's turn and minimising it otherwise. The algorithm essentially plays out all possible sequences of moves up to a certain number of turns (defined by the depth parameter). One thing which became apparent straight away was that the size of the resulting recursion tree meant the problem became intractable for greater tree depths due to having a complexity O(n^x) where n and x roughly equate the average number of possible moves and depth of the tree respectively. In an attempt to optimize the algorithm, we did the following:

## Optimizations

### Reducing the move list

It is often the case that more than one of the moves available to a player will lead to the same destination due to being able to use different means of transport. As these will lead to the same board configuration, these moves become redundant and can be removed from the list of moves to evaluate, speeding up the execution time of minimax. In our implementation, excess moves are removed from the list based on the scarceness of the required ticket(s) i.e. rarer tickets, such as double tickets and secret tickets are preserved where possible, unless they are the only way to get to a certain destination.

### Ordering the moves list by likelihood of a good score

The moves available to the agent are initially sorted by heuristic score, i.e. the scoring function mentioned earlier is applied to the board state reached by each move, and then each move is sorted using merge sort such that the moves with the highest heuristic appear first in the list. This sorting process is also done at each layer in the game tree, massively increasing the efficiency of alpha-beta pruning.

### Transposition table to contain scores for already seen moves

When a particular board state is being evaluated in the minimax tree, it is looked up in a table to see if a score for the position has already been found. If so, it will use this value instead of going deeper down the recursion tree. A basic hashing algorithm is used to access the table whereby the key is the sum of the detective locations each multiplied by an increasing power of the total number of nodes in the graph, subsequently being multiplied again by this total and then added to MrX's location. It is analogous to a binary shift. The formula for the algorithm is expressed as follows:

$$Index = X + \sum_{i=0}^{n-1} N^{n-i} D_i$$

Where *X* is MrX's location, *N* is the total number of nodes in the game graph, *D* is the sorted array of detective locations of size *n.*

### Timeout

The time taken to explore the Minimax tree varies with depth, branching factor i.e. how many moves and whether many moves have been logged in the transposition table. This time can be considerably longer than the time allowed for a move for a great depth with a sparsely populated table. As such, we developed a mechanism for terminating minimax when this time limit is reached. The minimax algorithm returns a pair consisting of a Boolean and the score value as an integer. The Boolean is returned as true if the time taken is within 5% of the time limit so the recursion tree can essentially 'unwind' and return the current maximum score found at the top without exploring the tree further.

**Alpha Beta Pruning**

Alpha Beta pruning increases the efficiency of minimax by eliminating less-promising subtrees meaning the game tree can be explored deeper down the routes which are more likely to end in a maximised score. The ordering of moves mentioned makes it easier for the alpha beta pruning to eliminate branches. Overall, this decreasing of the branching factor made a noticeable decrease in time taken to explore the game tree to a greater depth.

**Parallelisation**

Making use of threading offered a significant increase speed. We created a 'MinimaxThread' class which could be instantiated and run as a thread, containing all attributes and methods necessary for minimax to run recursively within it. When picking a move, a thread is created for each move available and the game trees for each move are then explored in parallel. Each thread will exit either when the time limit is reached (the minimax function will return) or after the tree is fully explored

## Evaluation

There was some level of success in AI; when playing, the agent would generally move away from detectives and avoid cornering himself but this was less effective when more detectives were playing. Five or six detectives would often result in the agent only being able to look a few moves ahead for himself as the combinations of moves from the remaining detectives would, in most cases, mean the game tree would have more layers than could be fully processed within the time limit for making a move, even with the optimisations.

However, despite often losing to a full team of detectives, the AI agent would play what seemed to be a fairly optimal strategy, keeping its options open, and often lasting well over half of the rounds without getting caught. The performance of MrX was increased most by the parallelisation of Minimax as the game tree could be explored much deeper than before this was implemented.

Overall we were happy with the design but if we started the project again, we would likely try and break down the AI model further into more classes and create more custom data structures to make code more reusable and easier to maintain, as well as explore different algorithms to minimax. If we had more time we would, likely break down the minimax procedure itself into smaller chunks as it as fairly large block of code currently. Also we would explore a more efficient way to use threading (a probem initially encountered was that data was being inadvertently manipulated between threads which we fixed by using synchronization and making more 'MinimaxThread' attributes final), perhaps also creating more threads further down the game tree so as to be able to look ahead further.