Daniel Davies DD16785
Chetan Mistry CM16161

Scotland Yard Report

Model

The model was completed before the AI and all 114 (130 eventually after finding extra test features) tests pass. After several game playthroughs we are also confident that the model is fully functional and allows the user to play a game of Scotland yard.

Design Patterns

We used 2 design patterns in the development process of this model:

Visitor

This behavioural design pattern was used in the coursework as a way of separating out the implementation algorithms for moves from the moves themselves, thereby enabling different protocols to be taken depending on what kind of move the model was operating on.

Observer

This other behavioural design pattern was used to handle the spectators. Each time the game state changed, spectators of the game (the 'observers') were notified.

Encapsulation

The model is fully encapsulated by use of private on both attributes and method (more as a sign to other programmers that those attributes/ methods should not be accessed directly) as well as the final keyword, which prevented more than one instantiation of an object/ attribute, which was a way of checking through development that we weren't trying to change anything we shouldn't be.

Streams

Streams were used in some places as a convenient one liner instead of for loops where appropriate.

The valid moves helper class

Rather than creating the methods for generating valid moves within the Scotland yard model class, it was decided to make a nested helper class to group all needed functions for the valid moves into its own class. There were two main reasons for this:

- It is far easier to keep track of, develop, and maintain another nested class that stores all functionality for the valid moves. Changing anything/ finding bugs within this limited range was a lot easier than if it had been mixed in with the model methods.

- This valid moves function also needed to be used in the AI coursework model. Having grouped all of the functions together, it was easy to transfer one class than many methods into the AI coursework file.

Daniel Davies DD16785
Chetan Mistry CM16161

MyAI

Mr X AI

The Mr X AI created works on a standard minimax basis: recursively alternating between generating game tree nodes for Mr x and then generating moves for the detectives on top of those. From this, getting a score from the leaves to the root node involved selecting a minimum score from the sub-trees when on a node belonging to Mr x and a maximum score when on a detective node. The actual algorithm generates the leftmost possible node (almost like a tree traversal), then moves gradually right, with scores propagating to the root gradually at run time.

Making classes to store data

Two new classes were made for generating and storing data in the game tree.

- The Tree class. This class represented a node in the game tree, and hence needed to store the node score, the state of the current game (simply held with a list of players), the subtrees to that node (representing other levels to the game tree) and finally, a Boolean value which stores whether the node belongs to a detective level or a Mr x level.
- Static AIPlayer class, which stores just the basic features of a player needed in the game tree: a map of tickets, a location and a player colour. This ensured that the tree was as light on memory as possible. However, making a new class also meant being able to customise how the required data is taken from an object to be used in the implementation.

Encapsulation in classes

Both classes were fully encapsulated by use of the private keywords to signal to other programmers not to change the data directly.
To access/change the data, getters and setters were provided so that if data did need to be changed, side effects could be controlled/the state in the object after changing the data was consistent. This was useful for ticket management in the AIPlayer class for example.

Comparator

As part of the pruning described above, a sorting mechanism was needed to rank the moves in some way, and then take the best from that ranking. To do this, an anonymous comparator class was set up (overriding the compare method), which combined with the collections.sort() method, was able to sort the moves into a descending order.

Pruning

3 kinds of pruning were completed on the game tree to get as far ahead as possible. The three methods are outlined below:

Alpha beta

Alpha beta pruning was used as the first mechanism for pruning the game tree. With this, there was some increase in speed, (in some cases- where it could prune early) but by no means enough to get significant rounds ahead.

‘Killer move heuristic’

Many branches of the tree never even had to be generated by following a strategy.
Here we saved double moves (and if possible, secret moves) for reveal rounds only, since the best time to use them is to be able to get as far away as possible after being revealed to

leave the detectives guessing. This meant that, for at least the majority of the tree, many nodes on the Mr X level didn't have to even be generated.

Immediate best

After playing Scotland yard a few times, it was decided that in most cases, if a bad move is made by the player then that ultimately affects the quality of the situation further into the game. This gave an opportunity into some more pruning of the game tree. The moves were scored by use of a comparator (described below) and the best square root of these moves were taken to go into the game tree, with the hope that if a good move is chosen early on, then it will eventually lead to a good game playout.

Choosing rounds ahead

Because of the variable number of players in the game, rather than fixing how many plies the tree goes into, it is dynamically calculated at the beginning based on how many players there are. This means that the game tree can go into as much depth as possible.
The way the values for the rounds ahead were chosen was through testing: trying out the algorithm on each number of players and seeing how far ahead it could go in the time given.

Timing

The AI has to determine how far it should go ahead based on how many players there are. Sometimes if it is running on a powerful computer, it would understate this guess and make the move in a much shorter time than it anticipated. If it overestimates the time it has though, there would be the problem that a move would not be selected in time and the opponent would win.
To prevent this, a timer was built into the AI. This stores the start time, and calculates an end time (exactly 14.5 seconds after the AI was called). While generating parts of the tree, a check is made on whether time has expired. If it has, then the generation stops, leaving values to propagate up through the tree to the root node, ready to be selected by the evaluation method and returned.
This ensures that a move is always selected by the AI.

Extension- Detective AI

NegaMax

The same code/ tree generation algorithm was used for the detective AI. This however led to one problem: each of the nodes were responsible for taking either the maximum (detective nodes) or the minimum (Mr X nodes) of their sub-trees, designed for Mr X. For the detective, the node order would be flipped- and hence the scores would be taken in the wrong order.
After research, it turned out that a different algorithm called NegaMax solves this problem. Rather than having to flip the precedence/ evaluation of sub-trees by their nodes, instead the scores are negated (-1 * score) as they are entered into the tree. This has the same effect as flipping all of the node evaluation functions, but doing it in one place and with one line of code, which is far simpler and far less dangerous.

Scoring Function

We implemented the scoring function using the Strategy Design pattern, which we did to make the code more dynamic as it determines what to score during runtime- either Mr x or a detective, and using overloading, which scoring function to use (Move or Position).
We developed the Dijkstra's as a Singleton class so that if there were different implementations of the algorithm for players, we could accommodate for that. Also, by making it Singleton, only one object can be created which was all that was needed since it was a helper class which nearly every function in the scoring used. Dijkstra's loops through a

set of nodes, incrementing the return value for each pass through the entire set, halting when the goal node is found.
Scoring MrX has its own class. The program decides whether we are scoring his current position, or a move. When scoring his current position, we consider the number of exit routes from his current position, how good those edges are, and his distance from all the detectives by using Dijkstra's. We also factor how many tickets he has remaining at that node according to a weighting.
When scoring a move for MrX, it is very similar to scoring his position, but instead of using his current position, it's the destination of the move. If the move lands on a detective, it is assigned a very low score and pruned out.
Scoring Detectives also has its own class, and the program determines whether or not to score a Move or the current position, depending on the arguments passed in. For their position, the score is based on the Detectives' distance from MrX's last known location, the number of tickets they currently have, and how good their current position is based on the number of Edges.
The routine for scoring a detective is almost the same as for Mr X, except instead of taking a DoubleMove into consideration, and checking if a move loses the game, we only check for a PassMove and have it pruned out by assigning a low score.

Reflection on Scoring

Given more time, we would want to improve it; this could be done by implementing strategies for the players. For example, MrX: if the next move is a reveal round, then we would want to make sure that he is as far away as possible from a node with an underground edge. Detectives: ensure that they are as close as possible to an underground node on the reveal rounds. Also, we would want to improve the detectives scoring function so that it guesses the best location for MrX and tries to get to that rather than his last known location. We could also implement a strategy such as forcing MrX into a corner rather than just chasing him.

Reflection on the AI

Although the AI dynamically calculates how many rounds ahead it should go, there is every possibility that it finishes early (good pruning, not many valid moves etc.) and hence time is wasted. To solve this, iterative deepening could have been used. This works by starting from 1-ply search, and repeatedly extending the search by one ply until we run out of time, propagating the current best move up.
Another issue with the AI was simply the lack of testing that it went through. When developing the Mr X AI, print statements and judgement were used to decide whether output was correct or not. While this was possibly sufficient for just the Mr X AI, the AI became suddenly very complex when making it compatible with detectives due to the increase in control logic. This meant that printing out was no longer sufficient, and in all likelihood, there may be bugs still present in the code that occur in edge cases etc. To combat this, we should have done two things:

- Firstly, a lot of the control logic and complexity could possibly have been eliminated by using design patterns, or even creating an entirely separate class for the detective AI.
- Auto testing should have been used to test as many edge cases and play-outs for both AIs, rather than relying on a few print statements and intuition.

Initially the AI could go 4 rounds ahead for the 6-player game, with all pruning strategies applied, and a basic scoring function. After applying the scoring function with Dijkstra's however, the performance significantly dropped to just 2 rounds ahead (in near instant time), but was not able to get 3 rounds ahead. This was because of the repeated application of Dijkstra's algorithm on the nodes of the tree. One way we could have increased the performance of this to get further ahead would have been to store a HashMap of Dijkstra values rather than recalculating every time.