Alice Ferreira, nj19721 Edward Faull, pb19639

Scotland Yard Project Report

Summary

The implementation of the cw-model part of the coursework was led by the tests provided. First, we made the constructor for MyGameState. For this to be fully implemented, we had to also implement createAvailableMoves, which calls createPlayerMoves, with either the Mr X piece or a detective piece passed as a parameter depending on whose turn it is, to calculate which moves are allowed given the piece and the tickets they possess. This allowed us to pass the first batch of tests.

Next, we worked on the advance method. This involved implementing UpdateLog, utilising the visitor pattern and dealing with the case of double moves. The advance method also updates the locations and tickets of pieces if the move made was legal. This is done by creating new versions of the detectives and/or Mr X and his log with the updated tickets and locations then passing them as parameters to the new MyGameState returned by the subroutine.

The GetWinner method was addressed next, which we split into 'createWinner' and 'getWinner' for figuring out which pieces won and returning a list of the winners, respectively. Finally, we added small bits of code to complete MyModelFactory and pass the Observer-related tests. With this, all 82 tests passed.

Our implementation of the cw-ai part of the coursework was based around the implementation of a MiniMax algorithm, which would calculate the best move to take by examining the next most likely moves of the piece in question and its opponent.

This required a depth-first search of the map to find the shortest paths between the detectives and Mr X. For this purpose, we implemented Djikstra's Shortest Path algorithm, which was used to pre-calculate the distances between nodes at the beginning of the game.

A tree data structure was necessary for the MiniMax algorithm, so we designed our own Tree interface with its MyTree implementation which stores a root node. Each node is an instance of our class MyNode, which stores the score and current board, as well as a set of edges represented as a map. The map links each move considered to a node containing the advanced board. For the functions in MyTree, in addition to the basic addNode we included findChild to search for the updated board at the beginning of a move, and a static subTree method. This method was intended for readability. In reality, it simply constructs a new tree. However, we felt it would be clearer of our intent to use a specifically named method when creating subtrees.

The buildTree function provides the backbone of the MiniMax algorithm. It calculates a 'score' for each possible move. Each node of the tree is the state of the board after one side moves and stores that move's score. At every level of the tree, either the maximum or minimum score is deemed the best/most likely course of action based on whether that level represents the turn of the piece who is deciding on a move or their opponent. This is based on the idea that on each turn the piece in question will be aiming to maximise its own score while its opponents will be aiming to minimise it as each side tries to win. The detectives are grouped together as Mr X's opponent for simplicity. The piece currently making a move will then choose the move that leads to the highest score for itself at the bottom of the tree. Branches that lead to a node where the piece wins or loses have very high or low scores and those nodes have no children.

The score for each move is based on the modes of transport that would be used and how much nearer or further Mr X would be from the detectives afterwards. Each ticket type has a set price, with taxis and buses being cheap (given the values '1' and '2', respectively) and underground and double moves being more expensive (valued at '3' and '5', respectively). Mr X is discouraged from using secret moves unless it provides a significant increase to the number of possible positions he could be in after the move. The detectives don't always know Mr X's position so before the first reveal round they aim for underground stations instead and on every round that isn't a reveal round they predict where Mr X could be given the last known location. To do this, they look at the neighbours of the last known location as well as their neighbours. The case of Mr X using a double move is dealt with separately. The amount of visibility, i.e. how many possible locations Mr X could be in, impacts the score of any given move as Mr X is likely to avoid locations with high visibility where there aren't many places to go, so the detectives avoid them too. Manoeuvrability, which is how many different types of transport Mr X can take from a location, is also factored into Mr X's score to encourage Mr X to get to locations with lots of modes of transport.

To work around the bug that prevented the terminate parameter from helping stop the pickMove method early, we implemented our own clock in the Clock class. It was quite simple, simply keeping track of when the method was first called (using System.nanoTime), and on each leaf we checked to see if it exceeded the time limit, given as 29 to allow an extra second for moving up the stack and selecting the correct move.

Finally, BoardToGameState uses the adapter pattern, converting an ImmutableBoard object into a GameState object. To do this, a new implementation of advance was written in the adapter, as well as short implementations of the Board methods which just returned what the ImmutableBoard object did.

Reflection

We were able to pass all the tests in cw-model, but that does not mean our solution was perfect.

We decided to use two subroutines for generating available moves, createAvailableMoves and createPlayerMoves. The separate subroutine createPlayerMoves for generating only single moves was implemented so that double moves could be dealt with as two successive single moves, which only Mr X was permitted to make. This was easier for us to handle than dealing with both double and single moves at once. This may have been possible to do without a separate subroutine, however.

In the advance method, we added several things to be passed into the constructor of the new MyGameState created and returned at the end of the subroutine. This was necessary for the way we decided to pass on the changes made in each turn to the next and so advance the game, which was to write them into the next game state. However, there should have been ways to update the game without needing to pass more parameters into the GameState constructor like this which may have been better than our method.

The placement of createWinner and createAvailableMoves in the construction of MyGameState was done to minimise the number of parameters passed into it by advance. The parameters we did pass in were all that were needed to instantiate all the variables in the class. An alternative implementation could have called createWinner and createAvailableMoves in advance instead and passed them as parameters with everything else.

The test-led nature of cw-model was useful for development as it guided us as to what to work on next and helped us be confident that what we had implemented worked robustly and as expected. However, we opted not to take this approach when developing cw-ai. Manual testing may have slowed down development time and led to crashes from parts of our program that we thought we had successfully implemented already. In future projects, it would benefit us to design automatic tests beforehand.

We decided it would be best to use a well-known gaming AI algorithm, MiniMax, rather than try to develop our own. It may have been possible to develop our own algorithm from scratch that was more heavily based around the Scotland Yard game rather than trying to fit an established algorithm into our program, however an established algorithm would be more thought-out, more thoroughly tested and so more guaranteed to work.

Our Djikstra algorithm used a list in place of an in-built priority queue because we found it easier to work with. We were able to successfully simulate a priority queue by constantly sorting it to keep the highest priority nodes at the front of the list. Some time may have been saved if we had used a priority queue instead, though.

While predicting the locations Mr X could be in when deciding on detectives' moves is the right approach, it is difficult to do accurately as the amount of possible locations grows exponentially with every turn after a reveal round. The guesses are most accurate right after a reveal round, but we had to limit the size of our list of possible locations beyond that, otherwise there would quickly become too many to handle. While this improves the efficiency of our detective AI it decreases its accuracy.

Our program works well and is fairly robust, however it is quite slow. Efficiency was not the focus when we were designing our code and it shows in double or triple nested 'for' loops which slow down the runtime. This is clear in the final project, which runs noticeably slowly.

The peak of this problem came during the development of cw-ai. While following the MiniMax algorithm was useful, we struggled to get our version to work in a reasonable amount of time. For the sake of efficiency, we added alpha-beta pruning to buildTree, which ignores branches of the tree the game is unlikely to go down and so saves itself from calculating the score of every node. We also added the subroutine cullExpensiveMoves, which removes moves that end up with the same result as moves that are cheaper, which also cuts down the amount of calculations required to build the tree. Even with these, though, the buildTree subroutine would recurse hundreds of thousands of times and the game would end via timeout before Mr X had decided on the first move. We were finally able to fix this by adding updateTree to update the tree built in the previous turn instead of building a new tree for every turn. However, it shows the importance of using efficient algorithms and for future projects we will more carefully examine the runtime of algorithms and/or how many times they will recurse before implementing them.

Another way we compensated for the speed of the MiniMax algorithm was to make sure the tree did not have many levels so that fewer calculations would have to be done. However, this decreased the accuracy of the AI as the moves it made were based on predictions from only a few turns ahead in the game. With a more efficient implementation of the algorithm we might be able to increase the tree depth and so have our AI make smarter moves.

We also wrote our Djikstra implementation to be more efficient to alleviate the speed issue. The shortest paths were all pre-calculated at the beginning of the game because calculating these while the game was running slowed it down. Additionally, we decided to do the sorting of our 'queue' list with a merge sort algorithm because it is one of the fastest sorting algorithms for general lists.

Other methods such as dynamic programming could have been used to speed up our program as well. However, some of these could have been tricky to implement and we wanted to focus on developing our program to a stage where it worked robustly instead of wasting too much time researching and trying to implement the most efficient algorithms.

Overall, while our program works, it isn't perfect. It could be more time-efficient and certain choices such as the use of a list over a priority queue in our Djikstra implementation may have detracted from our program rather than helped. This has given us lots to improve on when doing similar projects in the coming years.