Project Part B Report - Team ‘m’

1.0 Overview and Task Breakup

The core problem was split into two distinct sections: evaluation of the board state and operations on the problem tree. Evaluation was handled by Murray Peh and discussed in section 2.0 while creation, pruning and traversal of the problem tree was handled by Mihai Blaga and discussed in section 3.0.

2.0 Evaluation Function

The goal for the evaluation function was to input a board state, and return a score for the board, ranging between -100 and 100, where -100 would represent a win for lower and 100 a win for upper.

2.1 Tensorflow

The original plan was to use Tensorflow to train a model. After generating a base model, we would run the player ‘m’ using the model as the evaluation function simply using a greedy method of choosing the next move. This would be playing against a simple random move choosing player. Then, by taking the win percentage of 500 games, the AI would be rewarded accordingly. A more accurate model for scoring would lead to a higher win percentage.

This would make a simple greedy player. However, as this evaluation function has been trained against a random player, it most likely would not play defensively and could put itself into dangerous situations.

After saving this model, a new evaluation model could be created. By training this new model against a mix of games against both a random move player and the previous model, we hoped to create a more advanced evaluation function that could win against other greedy players. This should then be directly able to be put into the Problem Tree as the evaluation function, where the model would favour attacking the opponent, and avoid losing pieces.

The Specification document didn’t say that Tensorflow is installed in dimefox, so the model would need to be dissected and recreated to provide the scoring. Fortunately, work has been done into a method that prints all the weights and biases of a Keras model (Heaton, 2020).
This should be able to be slightly modified to print the actual code used to calculate the values which can be simply copied and pasted into the evaluation function.

Depending on the amount of layers and the size of each layer, the evaluation function might run for excessively long periods of time, so finding the smallest sized function that still provides an adequate win percentage would be important.

At this stage it's not completely clear whether the plan is possible to achieve. The main flaw with this method is the training process, as it is not clear to us whether this is feasible within Tensorflow and Keras. The plan was to give the reward to the model based on its performance over many games, but it seems that training has to happen on a smaller level, for each evaluation of a board.
Training the model seems to require inputs that this method is unable to supply. Several attempts to actually train the model were attempted, but we were not able to get the training program to run. After spending the majority of the time trying to get this to work, I moved on to ensure a working evaluation function was submitted.

2.2 Alternative Strategy Implemented

Instead of training a model by playing games, I planned to generate board states and their scores to train a model using the fit method in Keras. I then realised that if I was able to provide scores to random board states I might as well just use that as the evaluation function.

The final Evaluation function is fairly simple. It takes the board state and the player, then after counting certain aspects such as remaining pieces and distances from the enemy, it sums them together with certain weights to provide a final score. Testing for different weights is important to prevent unintended behaviours. For example to prevent our pieces from being targeted, early versions would take its own pieces to prevent negative penalties. Later on, the greedy function wouldn't take enemy pieces, as the new target distance would be greater by taking the piece.

Eventually a well working evaluation function for greedy play was found. It is able to beat a random move player in very few moves using very few pieces, and can beat the provided greedy player Greedo in the battleground. I then set up a way of playing two greedy functions against each other for n games, outputting the number of games that didn't end in a draw, and the win percentage of one of the players from these games. For example, out of 500 games, there may be 200 draw games. Of the remaining 300, the Upper player wins 200, with a "win" percentage of 0.66. This was used to modify the evaluation function, changing weights and adding variables to hopefully result in a more accurate function.

2.3 Evaluation Function References

1. Heaton, J., 2020. *Applications of Deep Neural Networks*. [online] GitHub. Available at: <https://github.com/jeffheaton/t81_558_deep_learning/blob/master/t81_558_class_03_5_weights.ipynb> [Accessed 1 May 2021].

3.0 Problem Tree

3.1 Overview of Solution

Sequential Game Explanation

The method implemented is a transformation of the problem into a sequential game where our player plays the first move and assumes the opponent will respond with the best response as determined by the above evaluation function (this was inspired by the lectures, Farrugia-Roberts 2021). With aggressive pruning, branch ordering and subgame evaluation, our player is able to analyse two moves for itself with one move for the opposition within the time restriction. While other solutions were considered, this was empirically tested against our other players and performed the best within the game and with respect to the time restriction. Surprisingly, due to the difficulty of accommodating the uncertainty that is a part of the game, our greedy solution consistently performed well against all our other players and was also a strong candidate.

Move Ordering

Moves are ordered first by throws and then by recency of when a piece was thrown (most recent piece thrown to least). The reasoning for this ordering is that the more recent pieces are expected to be closer to the opposition, and so would be more likely to be the pieces that have the best moves. It is important for the best moves to be considered first as this optimises the alpha-beta pruning performed. We could not determine how much of a difference this made to the evaluation time but believe that it did improve it.

Move Space Optimisation

The move space for each player was reduced using three different techniques. This sacrifices optimality but improves performance and allows for the consideration of an additional move. Firstly, this was accomplished by considering all throw moves as one. The piece type was decided based on the proportion of opponent and own pieces, favouring pieces which could attack many of the opponents but could not be easily attacked themselves. This served to at least half the number of moves considered at all stages of the game.

Secondly, only throws in the two most aggressive rows were taken into account, once again because they were considered to be the most likely moves the opposition would play. Again, this almost halved the number of moves being considered at each turn.

Finally, throw moves were only considered when the number of pieces on the board were fewer than a predetermined threshold (4). In this way, we both limited the number of our own pieces (reducing the number of moves that need to be considered) but also left opportunities to throw invulnerable pieces later on in the game. Once again, we believe this roughly halved the number of moves considered.

Result

With alpha-beta pruning, move ordering and reducing the number of moves considered to ⅛, the player is able to consider more moves and make a more justified decision.

3.2 Other alternatives considered and tested

Sources

While reading some literature (Bonsasky, 2013 and Saffidine, 2012) several other techniques were considered but could not be implemented well enough to outperform the paranoid sequential solution. Nevertheless, the methods are still in the code submitted and can be viewed.

Optimistic Sequential Game

Of these, the most successful was an optimistic sequential game which is similar to the paranoid sequential game but the opponent is considered to move first to which our player would then choose the move effective response.

Simultaneous Alpha-Beta Search

Literature explained a process in which the paranoid player was treated as a lower bound while the optimistic player is treated as an upper bound. This is expected to be the most effective extension of the player but, there was difficulty combining these bounds together into a single solution. The primary issue was that of performance. The extreme branching factor of RoPaSci 360 makes the techniques discussed in this research paper very difficult to implement. The matrices and linear problems being considered are too large to solve within the minute allocation. As such, they would only evaluate very shallow board states and were used mostly as testing partners against our paranoid player and in the creation of the evaluation function.

Game Theory and Nash Equilibrium

The other player which we created attempted to calculate the Nash Equilibrium (discussed in the game theory section of the lectures, Farrugia-Roberts 2021). In this, all possible combinations of moves were considered and evaluated, and with the implementation “gametheory.py” (Farrugia-Roberts 2021) a mixed strategy was calculated. This greatly underperformed when compared to the sequential techniques discussed above. The reason we believe this to be the case is because of the shallow depth of the moves which were considered. Not enough insight about the board state could be evaluated by considering a single combined move and any further analysis was too time consuming. Further, we had trouble modelling the game states as truly zero-sum and so the probability distributions calculated for each move was greatly affected. Once again, this solution is still present within the code.

3.3 Problem Tree Sources

1. Saffidine, Abdallah & Finnsson, Hilmar & Buro, Michael. (2012). Alpha-Beta Pruning for Games with Simultaneous Moves. Proceedings of the National Conference on Artificial Intelligence.
2. Bosansky, Branislav & Lisý, Viliam & Cermak, Jiri & Vítek, Roman & Pechoucek, Michal. (2013). Using Double-oracle Method and Serialized Alpha-Beta Search for Pruning in Simultaneous Move Games. IJCAI International Joint Conference on Artificial Intelligence.
3. Farrugia-Roberts (2021) Solving Simultaneous-play Games. Subject in Artificial Intelligence at the University of Melbourne.