An Explanation of the AI Techniques used in RoPaSci360

To further our knowledge and understanding of Artificial Intelligence, the project we have taken upon is the creation of a game-playing agent for the two-player simultaneous-play board game; RoPaSci 360. RoPaSci 360 is a fully observable (perfect information) static game with discrete moves. Our goal is to implement an AI utilising primarily game searching algorithms and its optimisation to be able to achieve reasonable performance within a limited runtime and memory limit.

Figure 1: Output game board by game-playing AI

Minimax

Our initial implementation used the minimax algorithm as the basis. The game was broken down into states where each state represents the state of the board. It contains information of all the tokens on the board, their symbol and position and how many throws were left for each side. Our algorithm took this state and returned the best action it believed it should take for the player. As minimax is not designed for simultaneous play, it has been implemented with the AI player going first with ‘paranoid’ reduction to restrict good moves for opponents.

Two classes were created to aid in the delegation of the program. The Gameboard class represents the state of the game. It included the number of tokens in hand, the token positions on the board and if our player was an upper or lower player. This class also contained methods to update the gameboard at the end of each turn and methods to create and update a new gameboard state to predict future moves.

The Token class was used to provide information for each token. This includes its position, symbol, player side (upper or lower) and the token types it can defeat or be defeated by. The Token class had methods to obtain all viable actions and to format actions to be returned to the referee.

One of the first issues of minimax on RoPaSci 360 is the large strategy space for both players combined. Each token had at most six slides or even more actions when swings were considered and many throws were possible depending on how many throws were left. Computing every possible move to 360 turns would be optimal as the game is sequential but was too computationally heavy as the time would grow exponentially per turn we added especially since the number of potential moves is so high. To work around this Alpha-beta pruning and a cutoff with an evaluation function was implemented.

Alpha-beta pruning would significantly improve performance by removing actions while not affecting the outcome and therefore still remaining optimal. The cutoff and evaluation function however was a more important focus during development as its implementation will have a larger effect on computational runtime but will influence the final decision in return.

Minimax Evaluation Function

This evaluation function was initially built on the utility score of a state which was a basic scoring value from the game's design. This utility score is equal to the number of opponent tokens defeated by the player minus their opponent's. This subtraction was to achieve a zero-sum value where a positive value means the player has more defeats but negative means the opponent has more. Therefore the goal of the minimax algorithm was to maximise opponent defeating and minimise own token losses. Having more opponent tokens defeats corresponded with being closer to winning as having more tokens available indicates having more strategies possible. Through this idea, maximising this score in the short term would evaluate towards optimal strategies.

This simple evaluation score system worked reasonably well, however it did not consider many other elements of the current state and did not consistently predict the future state a player would be in. It also did not break ties where two actions would have the same evaluation score. So as part of the evaluation, multiple sub-functions acting as predictors were added to the evaluation score.

Minimax Evaluation Predictors/Features:

- tokens_on_board() & tokens_in_hand()
    - This predictor split the score by the difference in the number of tokens on the board and the difference in the number of tokens in hand (not thrown).
    - The advantage of splitting it allows us to weigh it differently. We considered having more throws available was more important than having more tokens on the board for the long run.
- defeats_token_distance()
    - This uses the difference between the closest defeatable opponent token and the token that can defeat it.
    - This distance value is subtracted from 8 and then divided by the number of enemy tokens.
    - The reason it is divided is to weaken this weight in the event where the opponent has a large number of tokens making it riskier to reduce the distance without considering other factors.
- token_board_progression()
    - This calculates how far the player's tokens have progressed within the board ahead of their maximum throwing row.
    - Priority can be given to states where tokens are progressed less to reduce the opponent's ability to get a token defeat by throwing into the player's token.
- num_viable_actions()
    - This evaluation predictor uses multiple factors in determining its outcome.
    - Its goal is to maximise the number of moves possible in a state and the number of enemy tokens that are defeatable with its tokens on the board and to minimise the amount of its own overlapping tokens.
    - Maximising the number of moves also will prioritize states of closer tokens which promotes swing moves which greatly increase the strategy space.
- min_attacking _distance()
    - Affects the final evaluation by calculating the closest opponent defeatable token to the nearest token that can defeat it.
    - This value is subtracted from 8 and then returned unlike in defeats_token_distance() where the number of opponent tokens matter.

A lot of these factors are subtracted by the result of the same predictor from the opponent's perspective to obtain a zero-sum value. With so many different predictors added to our evaluation function to score a state, each of them had to be weighted in such a way that more important factors have a stronger weight

and weaker or useless predictors would be removed. Thus we applied our external knowledge of machine learning and linear regression to quantitatively weigh these values in the final evaluation function.

Linear Regression Machine Learning

The basic idea of our process was to build an evaluation function that took all the factors above from a state and then outputted a prediction of the utility score as if we ran minimax on that state with a large cut-off. For these continuous inputs and output, we applied linear regression machine learning to formulate the linear relationship between these predictors and the utility value.

The first step of building this model was to build the data to train upon. This was done by running a modified version of our AI with a much higher cut-off distance. This version of our AI ran extremely slowly but outputted much more accurate utility values of a state. The modified AI returned all the predictor values and the final future score value into a CSV file which would be our training data. The length of the final CSV file was over 1000 instances long over ten hours of runtime with a high cut-off.

The process of building a model could be completed in two methods. The first idea was using the Sklearn library within python and then implementing the fitted Sklearn model within our final program. The second is to use R language and take the outputted values manually into our AI. We decided on using R as we were more familiar with its linear regression toolset and inputting the values manually allowed us to make manual modifications if we desired to change the AI's priorities despite its effect on the prediction.

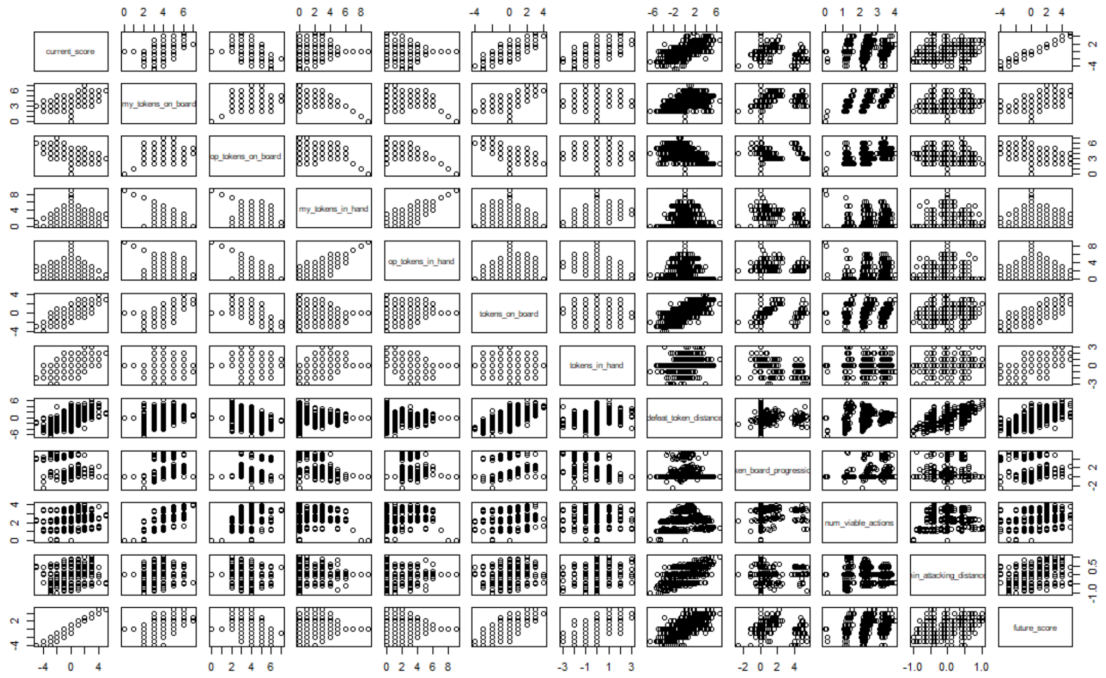

Figure 2: Each feature/predictor plotted against each other using R.

```
Residuals:
     Min       1Q   Median       3Q      Max
-1.04074 -0.36054 -0.02662  0.37840  0.99920

Coefficients:
                          Estimate Std. Error t value Pr(>|t|)
(Intercept)               0.026622   0.029939   0.889    0.374
tokens_on_board           1.023140   0.015643  65.405   <2e-16 ***
tokens_in_hand            1.067160   0.018907  56.442   <2e-16 ***
defeat_token_distance    -0.006318   0.012817  -0.493    0.622
token_board_progression   0.053871   0.012575   4.284    2e-05 ***
num_viable_actions        0.160682   0.012598  12.755   <2e-16 ***
min_attacking_distance    0.450556   0.051657   8.722   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.4283 on 1068 degrees of freedom
Multiple R-squared:  0.9389,    Adjusted R-squared:  0.9385
F-statistic:  2734 on 6 and 1068 DF,  p-value: < 2.2e-16
```

Figure 3: Linear Regression R output.
To see the weight of each predictor; look at its corresponding “Estimate” value.

The above estimated values were used as weights of the evaluation function. The higher weights represented the stronger it was as a predictor of the future score. A train and test split was completed to evaluate the performance of this function. Outputs from using the test data as input on the model trained on the training set fell well into the prediction interval indicating a good model. However, this cannot be exactly interpreted from the absolute values due to the values of some of these functions being much higher on average than others and therefore had to be multiplied by a smaller value.

In our analysis of the RoPaSci 360, we believed that having more tokens in the player’s hand is more important than throwing them onto the board. This was further suggested by the fact the tokens_on_board was weighted further than tokens_in_hand. What also affected our evaluation was how low the weight of defeat_token_distance and its p-value when inside the model. It was then removed from the evaluation function as it was not a good enough predictor for choosing states with good utility.

Nash Equilibrium

The resulting algorithm we believe had good performance however was not the final product. We believed we could have improved it further by utilizing more game theory solutions. The result of our attempt to improve the AI is the addition of a Nash equilibrium solver to support decision making.

Minimax paranoid did evaluate actions for simultaneous play but it was very conservative in its decision making, missing potential good moves by assuming the opponent could see our move. In response to this flaw in our design was the implementation of the nash equilibrium strategy with a backwards-induction algorithm as the main strategy decider in riskier moves. This is effective as it considers both player’s actions simultaneously per turn while still assuming each player is a rational agent attempting to maximise their outcome. Equilibrium will also return multiple strategies and distribute them with probability adding randomness to our actions which would assist in conflicts of multiple defeatable tokens present. This is known as Mixed Strategy.

```
* asking player 2 (lower) for next action...
[[1.84  0.334 0.89  0.89  0.89  1.89 ]
 [1.84  0.89  0.334 0.89  0.89  1.89 ]
 [1.84  0.89  0.89  0.334 0.89  1.89 ]
 [1.84  0.89  0.89  0.89  0.334 1.89 ]
 [1.838 0.888 0.888 0.888 0.888 1.888]
 [1.838 0.888 0.888 0.888 0.888 1.334]
 [1.84  0.334 0.89  0.89  0.89  1.89 ]
 [1.84  0.89  0.334 0.89  0.89  1.89 ]
 [1.84  0.89  0.89  0.334 0.89  1.89 ]
 [1.84  0.89  0.89  0.89  0.334 1.89 ]
 [1.838 0.888 0.888 0.888 0.888 1.888]
 [1.838 0.888 0.888 0.888 0.888 1.334]]
[0.0, 0.0, 0.0, 0.0, 0.15, 0.35, 0.0, 0.0, 0.0, 0.0, 0.15, 0.35]
equilibrium
0.888
*   player 2 (lower) returned action: ('SLIDE', (3, -1), (4, -1))
*   time:  + 0.156s  (just elapsed)    3.672s  (game total)
```

We found that minimax had a shorter runtime and already made good decisions early on so in our implementation so remained in our AI for cases where there are tokens able to be defeated on either side. However for the case when there is, an equilibrium would be used to find all optimal strategies and randomly select between them. In summary, our AI uses two game theory decision rules depending on what state is initially inputted for that turn. This results in an AI that we found ran faster in earlier turns and performed better in decision making during conflicts.

Figure 4: Equilibrium Matrix by game playing AI and output array of probabilities associated with each action.

The equilibrium solution solver algorithm similarly implements a layer cutoff level and evaluation function and a form of pruning to reduce runtime. Unlike in minimax, both of these optimisations will affect the final output instead of just only cutoff. In equilibrium, our AI's pruning is more manual and aggressive, removing actions considered not worth computing such as moves that did not defeat an opponent token or did not save a player token. In our testing, the pruning had to be adjusted to not be too aggressive otherwise the AI would have no choice but to take its piece since all other possible turns were pruned.

In comparison to minimax, this strategy required a single evaluation function for both players that had to be zero-sum to fit in the payoff matrix. The resulting function is similar to the one in minimax. However, we decided to add two extra functions which will heavily weigh the score to one player. The player can reach a state which concludes in victory (likely by defeating an opponent token).

Performance Evaluation

After each stage and change to our AI, we needed to rate its overall game playing performance and if our change has had a positive effect. The evaluation of our AI's components was done in mainly two forms; qualitatively and quantitatively. The focus of the qualitative analysis was a manual assessment and breakdown of the AI's decision making by running it and recording its actions in vital turns. The quantitative analysis was a numerical analysis of win rates against itself and the previous AI.

In the qualitative analysis of our AI's performance, domain knowledge of the game and strategy was required to judge its performance. This process of running the AI multiple times against itself and previous iterations and manually judging its strategy required us to know what was a good strategy in the first place. This was less generalized and a more specialized approach to performance evaluation as it took specific states and gave us examples where our AI lacked performance. This analysis was supported with the use of printing, code breakpoints, and the abilities to read values in memory at any point using our selected IDE.

For the quantitative analysis, a more direct modification of the referee game driver was required. The game was modified to repeatedly execute itself until a set limit and count the total wins, ties and losses up. Playing the AI against itself with random-ness enabled was expected to have a split fifty-fifth win-rate

or high tie-rate and against previous iterations. However, if our current AI was facing a previous version we expected it to have a slightly higher win rate over a large sample of games. This analysis is more general as it rates the performance over a larger sample and takes the average.

When using this as the measure of performance, our AI tested against our own greedy and random game agents perform significantly better with a ~100% win rate. When comparing previous iterations and small changes the win-rate difference varied from 50% to 60% over a large sample. It was not certain proof that an AI was better but provided a good measure and goal to work towards in every iteration. This is why a qualitative breakdown was used alongside.

When evaluating our initial equilibrium qualitatively we used a very aggressive pruning process, especially when attacking another piece. If the player token, let's say is Rock, could defeat an enemy Scissors token next turn, it would move to defeat the token without considering where the enemy token might move. This was because running away actions were pruned from both sides. This approach also meant that if the enemy scissors token was on a hex shared with the player's scissors token, the Rock token would end up defeating its scissors token, while the enemy scissors token could run away. This is one of many qualitative performance evaluations that led to the altering of the final AI by allowing the matrix to feature more actions where more tokens could move.

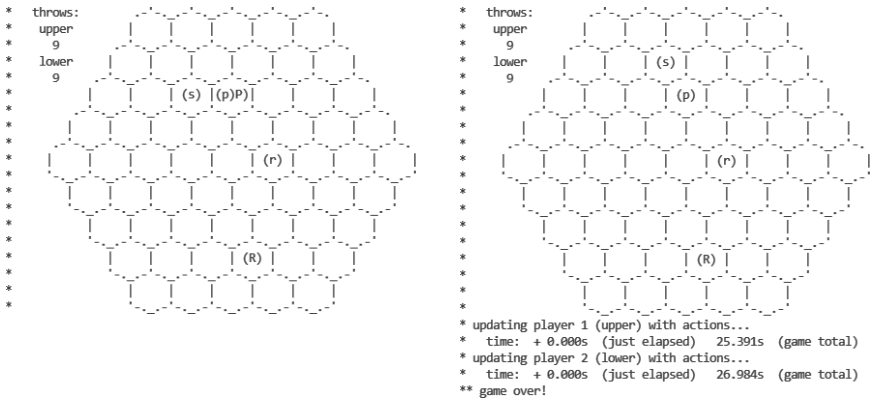

Figure 5: AI Print output showing improved decision making. Left is turn 141 and right is turn 142
In the previous version of this AI, lower (s) would go and attack it's own lower (p). Now the AI predicts and expects (P) to run away which ends the game.

Conclusion

In the end, using our understanding of game playing agents and machine learning regression; the final AI built for the game RoPiSci 360 utilizes a combination of minimax and equilibrium game theory. Minimax sets the player up for good moves and equilibrium strategy finds the optimal moves while adding an element of unpredictability with mixed strategy. Each of these algorithms utilizes a cut-off and evaluation function to predict the utility of a state using linear regression to optimize its weights.