Monte Carlo Tree Search and Enhancements Checkers AI Model

Seth Singson-Robbins
Fordham Univesrity
ssingsonrobbins@fordham.edu

Abstract—This paper investigates the implementation of the Monte Carlo Tree Search (MCTS) and its adjustments on the game Checkers. This requires the creation of the Checkers AI environment and four models. The control will be MiniMax with the other three AI models being MCTS, RAVE-MC, and Heuristic RAVE-MC. After 546 simulated games, the results were that the standard MCTS had the best performance with a win rate of 58%. These results and further studies could be utilized to understand the pros and cons of each model for different games and game theory related industries.

I. Introduction

In the paper "Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go," Sylvian and Silver pioneer the use of the algorithms RAVE-MC and Heuristic RAVE-MC in attempting to optimize the Monte Carlo Tree Search Algorithm on the game Go, a complex strategy game that has long been considered a benchmark for testing AI capabilities.[1] The development of Rave-MC and Heuristic RAVE-MC played an important role in advancing the field of AI and machine learning and continues to inspire new research and innovations today.

The paper is an analysis of these improvements from the original Monte Carlo Tree Search (MCTS) on a new game. MCTS is a reinforcement learning artificial intelligence model used in decision-making processes, particularly in adversarial search scenarios like gameplay.[2] It is a simulation-based technique that involves building a search tree, which represents all possible moves and their outcomes in a game. The algorithm then traverses this tree using a combination of random to select the most promising paths based on the selected goal, in the case of games would be a win or a more optimal environment. MCTS is optimal in handling complex and large-scale decision-making problems with uncertain outcomes.

Rapid Action Value Estimate Monte Carlo Search (RAVE-MC) is a variant of the MCTS algorithm with a different algorithm in how it evaluates moves. While MCTS uses a combination of random sampling to select the most promising paths, Rave-MC incorporates a technique called Rapid Action Value Estimation (RAVE), which assigns values to moves based on their performance in previous simulations. In the game Go, it has allowed for faster and more accurate evaluations as it uses the knowledge it used in the previous simulations to score the current simulation round which is better when the branching factor is higher.

Heuristic RAVE-MC is another optimization on top of RAVE-MC where, instead of picking the branch purely by the best score, the algorithm also works to simulate games that have the most promising moves based on domain specific knowledge making a more efficient exploration of the space to estimate the value of each action. This helps Heuristic RAVE-MC focus on the most promising branches of the search tree, resulting in faster and more accurate decision making.

The benefits of these algorithms have been tested on the game Go and other cases even outside pure gaming. However, this paper will test the capabilities of these algorithms through the game Checkers. Checkers is a classic board game that is played on an 8x8 board played between two players, each controlling a set of twelve pieces that are typically red and black in color. The pieces can move diagonally forward one space at a time except if they eat an opponent’s piece by jumping over it or if they reach the end of the board where they become a king piece and can move backwards as well. The goal of the game is to capture all the opponent's pieces or to block their moves in such a way that they cannot make any more moves.

Checkers has similarities with Go having a small number of possible moves and an unlimited number of gameplay possibilities making it hard to map out every possible move from both players.[3] However, differences in the game itself make it difficult to determine if checkers will have the same results with these AI models as Go. Checkers is a reduction game, where pieces are removed over time while Go is a accretion game, where pieces are added over the course of the game. Checkers is also a much deeper but narrower game meaning that there are less possible moves per round but there is a much longer number of moves before the game terminates compared to Go. Lastly, Checkers can end in a draw due to one player’s pieces being stuck or impossible to eat the other without forfeiting. Our version will terminate after 300 moves to avoid this, but it will also make the evaluate step a bit more complicated than in games where there is a clear winner or loser.

By the end of this paper, one will be able to determine which AI models are the best at playing the game checkers as well as understand how the gameplay of these AI models differentiates.

II. Methodology

This section looks at the additional work done to determine and run the model. This includes how the Checkers environment was created, the minimax AI model that was used as a control variable, and how the three MCTS models were created.

A. Checkers Environment

To test the AI models, a checkers environment needs to be created first. The environment utilized was created by the GitHub user Tim Ruscica and utilized PyGame to allowed for a visualized version of the game.[5] The environment is designed with three python objects.

The first object is the Piece object which holds several object attributes including a piece's row and column on the board, which player's piece they are (by color so 'White' or 'Red' and if the piece has been kinged. Some of the piece's actions include making the object a king, listing the legal moves that the piece makes, and moving the piece as a gameplay move.

The second object is the board itself. The board's attributes are all the pieces for each player and how many pieces and kings each player has. The actions include getting all the pieces of each player, what the board will look like after a specific move, removing pieces after they have been eaten by the opponent, and getting a list of valid moves based on a specific piece.

The last object is the Game itself. It has several attributes including identifying the environment used for PyGame, identifying the board object that the game is currently at, the player whose turn the game is currently at, and listing all the valid moves that the board can play next for each player (same as the board).

A. Minimax AI Model

To ensure that the models created are tested by an outside model, the minimax model created by Tim Ruscica.[5] The model looks through the next few moves (the exact number being a hyperparameter called 'minimax moves' which will be test for performance purposes). It has a scoring system of the board by looking at how many more pieces the player has versus its opponent with extra points added or subtracted based on the pieces being kinged or not.

To determine the best move at every point, the model will maximize the evaluation score (the max in minimax) and assume that the other player will want to minimize the evaluation score (the min of minimax). This will backtrack back to the initial environment picking the best move for the next number of minimax moves. This model will be used as a control for each of the AI models.

B. MTCS Model

The MTCS Models require some background into how MTCS determines which is the best next move. MTCS requires four steps for each simulation that the algorithm runs. Figure 1 shows the steps in one diagram with each step explained below.

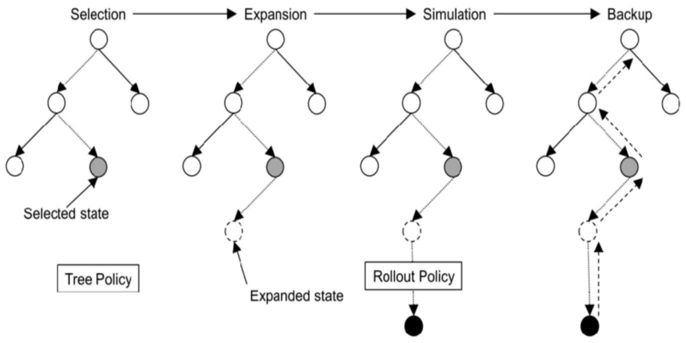

Figure 1. MCTS Step by Step Process

1) Selection

When the game starts, it creates a tree where the current environment is the root node. It then looks to see what each of the possible moves are for that board calling each of these possible moves the children of the root node.

For every step of the selection process, it then picks which of the children it should explore by a scoring system that has two metrics. The exploitation metric looks at the percentage of simulated games with that node led to a win. The exploration metric looks at the percent of games that the node was used for the simulated game and is multiplied by an 'exploration factor' which is a hyperparameter. The higher the exploration factor, the more likely that nodes that have not been utilized as much for simulations are going to be selected for simulations. These two metrics are then added together and the node with the highest score picked. This is repeated if the node has already been expanded until it reaches a child that has not been expanded with its children's nodes.

$$\text{Score} = \text{Exploitation} + \text{Exploration} * \text{Exploration Factor}$$

2) Expansion

Once the node has been selected, the children of the node will be added to the node and a random node will be utilized as the next step in a simulated game. This way more learning is added to the tree knowing what the possible next moves are in the game.

3) Simulation

Now that a node has been picked, the simulation will play a simulated game of checkers to see what the results will be for the game. For MTCS, each move between both the current player and the opponent will be moved at random until a winner has been declared. Due to technical limitations and to ensure that it can more efficiently predict the next move, it will only go 20 moves in the future and then pick a winner based on an evaluation metric similar to what was utilized for minimax (with the player with the higher evaluation score being the winner).

4) Backpropagation

With a winner selected, the model will note that a simulation was played and if the player was the winner. It will then repeat this process for the parent of the node and repeating this process until it is back at the root node so that the AI model can learn at each node if that node was a participant in a successful or failed simulated game.

These steps are repeated many times, set by a hyperparameter that determines the number of iterations, to help the model make an educated guess at the best move at that point through the reiteration of many rounds of gameplay.

C. MCTS Model in Python

To create this model in Python, two objects had to be created. The first object is the model itself called MCTSAgentsAll. It holds the parameters necessary for MCTS and the other variants including the number of iterations and the exploration factor. Its main function is get_action which is the main AI Model and returns which board with the player's move would be optimal based on its calculations. Other actions include select_child, which is the formula used to determine which child node should be picked as it goes down the MCTS tree and add_child which creates a new child node and adds it to the existing node.

A second object is the Nodes themselves that are utilized in the MCTS tree. Each node has many attributes including what the board would look like at that point, the parent and children nodes, which child nodes have not been explored yet, what action was made to get to that board, and the number of visits and wins used in the AI calculations. The add_child action is what is used in the MCTS object to create a new child node but also removes the action from the list of actions yet to be explored.

D. RAVE-MC Alterations

As noted previously, the RAVE-MC algorithm is an updated version of the MCTS model. It utilizes the MCTS AI model with a few modifications at the scoring steps. On top of using the typical number of simulated games and wins by the node, it looks at the move that the node made as the anchor point of a visit or win. For each of the nodes that have already been explored, it adds that as a visit (meaning even if it has not been explored at that simulation it will count as a visit). It will then determine if the move was a winner and, if the moves done by that player was done at other nodes, it will also count those nodes as wins. This helps the model learn from other simulations that it was not a participant in and learns what moves led to wins. It will then calculate their own exploitation and exploration metrics (called rave_exploitation and rave_exploration, respectively) and add it to the original MCTS model as an additional consideration. A beta hyperparameter is added (and is tested) to determine how much influence the rave metrics will have on the final calculations and decisions with the higher the score the more important the rave metrics are than the typical MCTS metrics.

Formula: score =
(1 - beta) * exploitation + expl_factor * (1 - beta) * exploration +
beta * rave_exploitation + expl_factor * beta * rave_exploration

E. Heuristic RAVE-MC

Heuristic RAVE-MC model has an additional requirement beyond the RAVE-MC model previously mentioned. This AI model also updates the simulation portion of the model. Instead of picking a move for each simulated move at random, it picks the best child node on a heuristics scoring model. This model is based on several domain knowledge information of what strategies are good for checkers. In this Heuristic RAVE-MC model, the following rules each give or remove a point to the scoring model:

1. One point if it captures a piece.
2. One point lost if it loses a piece.
3. One point if a piece has been kinged.
4. One point lost if the first row (the homebase row) has a piece moved from it.

III. Results

A. Metrics

For model evaluation, the main metric will be the proportion of games that the model wins against each of the other models. This includes both the minimax model and the other AI models and the various hyperparameters each of the models entails.

Additionally, secondary metrics will look to better understand the gameplay strategy that comes from these models and along with their strengths and weaknesses. The number of moves shows how quickly it can win the game. The number of pieces left shows how defensive of a game the player played to win. The number of kings also shows the strategy of getting its pieces kinged. Each of these metrics will also look at how their opponent did to determine how it did on the defensive side.

Lastly, the time it takes to determine a move will be looked at via the time package to determine how efficient the model is at making a move. This is important as a very slow model would not be very useful against playing another opponent especially if there is a time limit on the game itself.

A total of 546 games were played by the models to determine the results.

B. Win Rate

Figure 2 shows the win, draw, and loss rate by model when only looking at games where two different models were playing against each other. By and large, the MCTS seems to have performed the best of the four models with a win rate of almost 60% and loss rate of only 19%. Another interesting point is that, except for MCTS model, the most common result from a game is a draw which happens after 300 moves or if one of the players cannot move anymore (therefore the pieces are stuck). This means that the games could go on for much longer and their strategies are not as long term focused on winning the game.

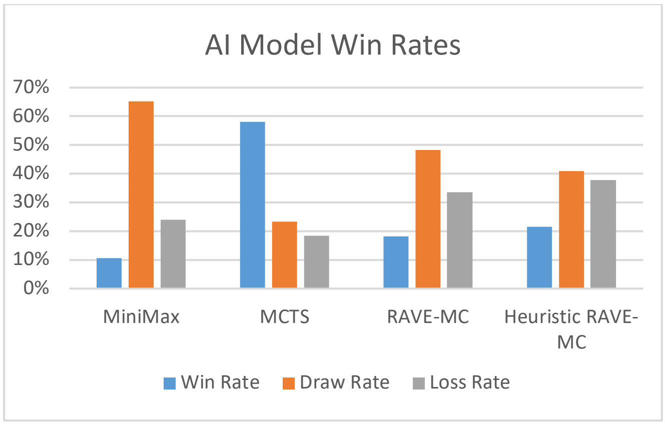

Figure 1. Win rate performance by AI model.

Figure 3 has the MCTS model win rates broken out by model. MCTS had a win rate of over 50% for every other AI model. However, a couple interesting notes. The MiniMax model had a much higher draw rate than the other models at almost 38%. The Heuristic-MC model also outperformed the other models in beating the MCTS 31% of the time.

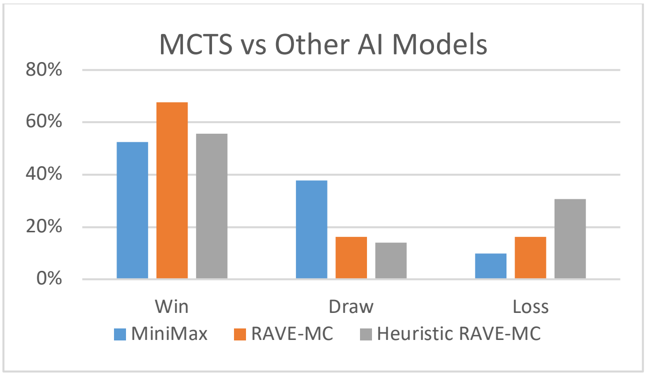

Figure 2. MCTS win rate broken down by opponent AI models.

C. *Secondary Metrics*

There are many secondary metrics that can help get a better understanding of the strengths and weaknesses of each AI model. Figure 4 shows the average number of moves that occurred in the game with that AI model. MCTS had the lowest number of moves needed to win at 194 moves while the other three models were all above 220. This average number of moves is logical as it is highly correlated with the proportion of games that ended in a draw.

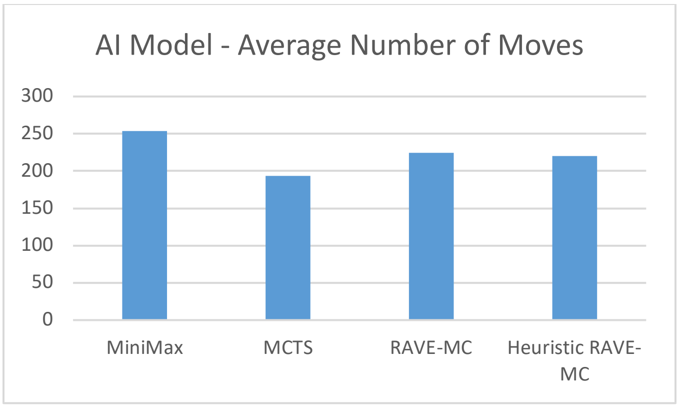

Figure 4. Average number of game moves by AI model.

Figure 5 and 6, which is split up by when they won or lost the game, shows the number of pieces left of the winner, the number of kings that model made, and the number of kings the opponent had. MCTS was great at keeping pieces throughout the game with an average of over 4 pieces left after defeating the opponent. Minimax was slightly better at kinging its pieces that MCTS but MCTS was really good at ensuring their opponent did not kings its pieces when it won. When models lost, the opposite happened where the MCTS model had the most kings by its opponent. Minimax was very good at kinging when it lost too which might have something to do with the high draw rate against MCTS, it most likely was able to defend itself by creating kings which MCTS was not as good at attacking.

Heuristic RAVE-MC had the lowest number of kings when it won but the highest number of kings when it lost. This might mean that MCTS's strategy of minimizing the number of kings from its opponent did not work in hurting Heuristic RAVE-MC's strategy which could be why MCTS had a higher rate of losses against this AI model.

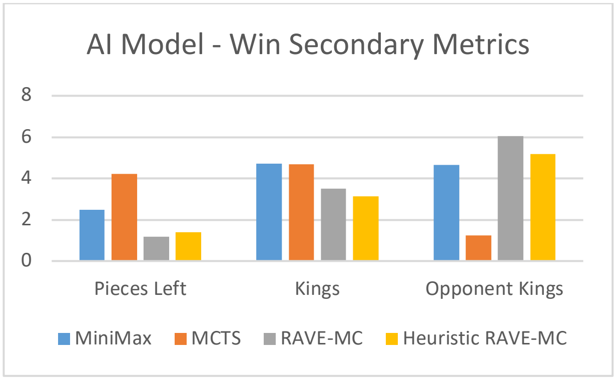

Figure 5. Secondary metrics of the AI models' wins

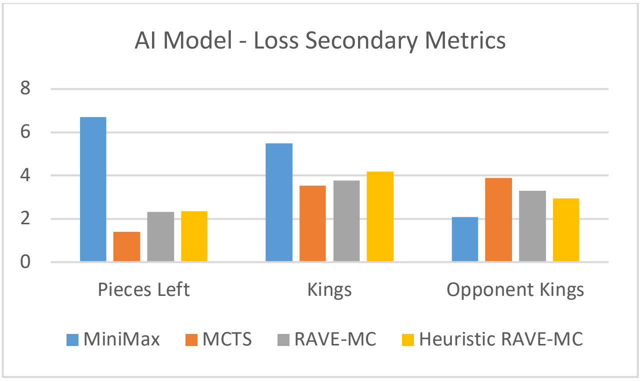

Figure 6. Secondary metrics of the AI models' losses

One last note is that there did not seem to be a correlation with win rates and the beta or iteration parameters. This might mean that the iterations do not matter as much but could also just not be as obvious without more testing.

IV. Conclusion

MCTS seems to have been the best AI model at playing checkers when comparing to the other models. There could be a number of reasons for these results, but one theory is that specific moves in checkers do not affect the outcomes without context of the placement of other pieces.

More optimizations would be useful into getting more insights from this data. One is that these models are very slow making it hard to increase the iterations (right now the max number of iterations was at 1,000 simulated games). It would be good to either get faster computers or run the model calculations on faster programming languages like C. With the exception of heuristic MC the models also do simulated games where both players play at random which is not how the models and real players actually play. Having a better method of simulated gameplay could help improve the results. Actual humans, especially those that are experts at the game, could have helped improved the model with feedback on how the AI Models' gameplay could be improved.

More testing on the use cases of these models could help open the possibilities of what is possible and when these models thrive or fail. It could be played on other adversarial games or even in industries that have competition decisions such as buying advertising spots or when to launch products into the market.

References

1. Silver, David, et al. "Monte-Carlo tree search and rapid action value estimation in computer Go." Artificial Intelligence, vol. 175, no. 11, 2011, pp. 1856-1875. https://www.sciencedirect.com/science/article/pii/S000437021100052X
2. Russell, Stuart, and Peter Norvig. Artificial Intelligence: A Modern Approach. 4th ed., Pearson, 2020.
3. Newell, Bob. "The Monte Carlo Method and Go." Bob Newell's blog, 9 Jan. 2008, https://www.bobnewell.net/nucleus/bnewell.php?itemid=220.
4. T. Ruscica, "Python-Checkers-AI," GitHub, Oct. 15, 2022. https://github.com/techwithtim/Python-Checkers-AI