Minimax and Monte Carlo Tree Search for Ultimate Tic-Tac-Toe

Anthony Lam
Boston University
adhl@bu.edu

Danny Chen
Boston University
dchen21@bu.edu

Jake Lee
Boston University
thejake@bu.edu

Abstract

In this paper, we analyze the performance of AI agents in Ultimate Tic-Tac-Toe. The methods used are tree searched based, namely Minimax with Alpha-beta pruning and Monte Carlo Tree Search. Series of games were simulated through pitting the AI's against each other with parameters tuned in between games to achieve results. Ultimately, we found that MTCS performs the best given high iterations. Due to the abstracted nature of the game, low iterations forbid MTCS from understanding the value of the mini tic-tac-toe games. This goes to show the importance of tuning parameters to the problem at hand for algorithms to perform as expected.

1 Introduction

In recent years, popular historically human dominated games such a Chess, Shogi, and Go have been defeated by sophisticated artificial intelligence agents. As new advancements in technology come, so does the rampant domination of AI in all aspects of human life. These AI perform tasks at a rate far surpassing humans, even leading professional go player Lee Se-dol to resign as a competitive player stating that "with the debut of AI in Go games, … even if I become the number one through frantic efforts, there is an entity that cannot be defeated".

Ultimate Tic-Tac-Toe (also known as Tic-Tac-Toe^2) is a two-player game on a 3x3 game board consisting of nine mini tic-tac-toe boards. Similar to normal tic-tac-toe, one player plays 'x' in squares she or he wishes to occupy while the other plays 'o'. The first player can play in any of the 81 squares on the board. After this each player is restricted to a board based off of what the opponent's previous move was. The restriction imposed is that the following move must be in the tic-tac-toe game corresponding to the grid of the tic-tac-toe the opponent last moved. However, if the mini tic-tac-toe game has already been won or completed (draw), the player is free to play anywhere they choose. For example, let's say player 1 played in the upper right square of the middle tic-tac-toe board. The following player would then need to play in the upper right board as illustrated below in Figure 1. Akin to normal tic-tac-toe, a player wins the game by winning three individual tic-tac-toe boards that connect in a line horizontally, vertically, or diagonally.

In this project, we explore two methods to tackle optimal tic-tac-toe, Minimax and Monte Carlo Tree Search, through the use of game trees. Game trees are used to track the states of our game board for there does not exist payoff matrices for tic-tac-toe due to their conditional winning moves.

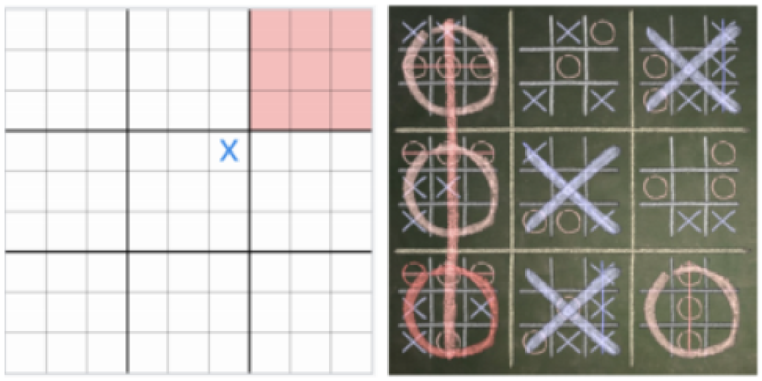
Figure 1: Example of move restriction and win state.

2 Code Structure

To implement ultimate tic-tac-toe, we created objects for tic-tac-toe and ultimate tic-tac-toe to enforce OOP principles. Our ultimate tic-tac-toe board consisted of 9 tic-tac-toe boards. Through this, we were able to achieve a framework to play and test our agents on. We tracked board status (ongoing, winner, etc..) and last moves to determine the states at which the board was in and to determine the possible states at that point in time.

For ease of computation, we denoted player markers as 1 and -1 as to easily find winning boards when a line's absolute value summed up to 3. As a result, we had a mapping function from 1 -> X and -1 -> O when drawing the board.

3 Minimax

Minimax is an extremely common decision rule frequently used in artificial intelligence and other game-theory related fields. There are two aspects to this algorithm. The name itself is derived from its two facets – minimizing the potential loss for the maximum loss, worst case scenario and maximizing the potential gain for the minimum case scenario.

The minimax algorithm is a recursive algorithm that alternates its objective – whether minimizing the loss or maximizing the gain – with each iteration up until a certain pre-defined depth. This is accomplished by playing every possible step and then scoring the resulting future states. The logic within is akin to that of the methodology we learned mid-semester to find saddle points.

As previously mentioned, there are two recursive stages of minimax, with one base case. As a shorthand, the maximizing aspect can be thought of as the AI considering its own point of view: which play will give me the most value? While considering its own perspective, the AI will try every position and return the maximum score of the set. Conversely, the minimizing aspect occurs when the AI is considering its opponent's turn. What is the highest value that the opponent can obtain, were I to make that move? These two maximizing and minimizing aspects always alternate, in correlation to the rules of the game, as opponents alternate turns.

The pre-defined depth details how many look-aheads the minimax function, and by extension the AI, will perform, i.e., a depth value of 3 means the AI will look 3 moves into the future. Higher depth results in more look-aheads, resulting in smarter play, but also longer computational times.

3.1 Alpha-Beta Pruning

Alpha-beta pruning is often associated with minimax functions. This is a search algorithm that seeks to reduce the total number of future states that must be considered in minimax functions. As minimax involves recursion and choosing either the maximum or minimal value for future states, keeping track of the current respective values is greatly beneficial. Alpha refers to the minimum score that the maximizing player is guaranteed to obtain, while beta refers to the maximum score that the opponent is guaranteed to obtain.

To ensure that their starting values do not disrupt the algorithm, alpha should be initialized to negative infinity, while beta should be initialized to positive infinity. Thus, the first values obtained from minimax iterations are guaranteed to replace the alpha and beta variables. During minimax, whenever a play returns a loss greater than the current alpha value, or a future state returns a gain greater than the current beta value, that move and all its future states are discarded. In this fashion, detrimental plays are not considered, thus reducing the total number of computations required for minimax.

Consider a scenario in which the AI's move would allow it to gain a value of 5. That 5 becomes the current alpha move. In the next phase of the minimax algorithm, the opponent's potential moves are considered. Unfortunately, the first potential move results in a victory for the opponent. Immediately, for the alpha value has been exceeded, the future moves are not explored by the minimax algorithm. This is alpha-beta pruning. As values, gain, and losses are so far still abstract, vague concepts, now is pertinent to introduce our heuristics.

3.2 Heuristics

Minimax functions are made and broken by how accurately or poorly the coding engineers value game states. Heuristics refers to the scoring system for moves. As heuristics shape the moves that the AI will prefer, ensuring an appropriate scoring system is essential. The more optimal a play is, the higher the number our heuristics function is outputting will be. Heuristics are called whenever the depth reaches zero, i.e., when the minimax algorithm reaches the end of its look-aheads. At the end of every minimax algorithm is a potential future state of the board, which is passed into the heuristics function. Our heuristics calculates the potential score for both players.

3.2.1 Small Boards

Each board has a value of ten. The winning player of each small board has ten added to their score. Corner boards – 1, 3, 7, 9 – are worth four more points. The very center board, 5, is worth eleven additional points. In summary, whichever player wins the center board will have twenty-one added to their score.

3.2.2 Tiles

The center tile of every small board and the tiles in the center small board are worth three points. The center tile of the center board is worth six points.

3.2.3 Miscellaneous

Doubles are worth two points. A double occurs when a player owns two tiles out of three required in a row, column, or diagonal to win a small board. This scoring encourages the AI to place their tiles in such an order to increase the number of potential wins, except in those scenarios where an instant small board win is possible.

4 Monte Carlo Tree Search (MCTS)

Similar to Minimax, MCTS is a probabilistic based tree search algorithm that calculates future states to determine the most optimal move. The main difference, however, is that it 1.) is not restricted to any layers of computational depth and 2.) does not require any heuristic based evaluation function. Furthermore, in the basic implementation of Minimax (sans alpha-beta pruning), the exploration of the full game tree can take an unfeasibly large amount of time, especially in games with high branching factors such as ultimate tic-tac-toe. As a result, MCTS's use of random simulations of games to determine the value of a move produces more accurate results than a positional rule based evaluation. In order to find the most optimal move given the current state of the board, MCTS iterates over 4 phases: selection, expansion, simulation, and back-propogation, hundreds and up to thousands of times.

In the first phase, selection, the algorithm selects a child node such that it has the maximum win rate. As to not face the exploration-exploitation trade-off in which our algorithm selects the same node repeatedly instead of exploring other nodes due to uncertainty, we utilize Upper Confidence Bounds

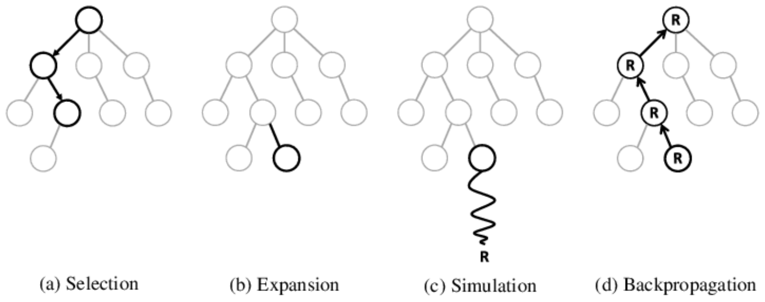

Figure 2: Visual Representation of the 4 Stages in MCTS

(UCB) to construct statistical confidence intervals for each node.

$$\frac{w_i}{n_i} + c\sqrt{\frac{ln(t)}{n_i}}$$

Where w_i denotes the number of wins after the i-th move, n_i denotes the number of simulations after the i-th move, c is the exploration parameter set to $\sqrt{2}$ or 1.41, and t is the total number of simulations for the parent node. The value calculated from this and used to select the most promising node ensures that over time, none of the possible nodes with be subject to starvation while selecting the most promising nodes.

Upon reaching a leaf node in our game tree of states, we can no longer apply our aforementioned UCB formula and must expand. We do this by visiting a random child, exploring all possible states, and appending them into our global game tree.

After expanding, we simulate the remainder of the playout randomly or vis-a-vis the implementation of simple weighting heuristics. While more computationally expensive, the agent playing against random moves is not realistic, particularly in a game with high branching factor. Thus, a heavy play out allows for more strategic and realistic play making.

Lastly, we must back propagate, hence the name of the phase, by traversing up towards the root node and incrementing the visit score of all visited nodes. Additionally, we update the win score of each node if it matches the player who won.

As mentioned previously, MCTS iterates over these four phases where the higher the number of iterations, the more reliable the estimate becomes. With our implementation of MCTS, we found that 600+ iterations is needed to make reasonably smart plays. Of course, 1000+ will produce even better moves though with diminished returns. This is an important parameter to hyper tune to essentially set the difficulty of the agent; however, with more iterations comes more computations and will slow down the agent.

5 Results

Running Minimax agents against one another with the same parameters, we found that the player going first won 56.66% of the time as summarized in table 1. While this may be confounded with the depth parameter, further exploration summarized in table 2 and 3 shows that regardless of the difficulty, going first provides an advantage in ultimate tic-tac-toe. This matches results from other studies performed with similar methods.

Table 1: Effect of turn order on win rate

First AI		Second AI		
Wins	Losses	Wins	Losses	Ties
64	49	49	64	13

Table 2: Turn order for max difficulty

Hardest AI (depth = 2)		
Going First Wins	Going Second Wins	Total Wins
32	27	59

Table 3: Turn order for medium difficulty

Medium AI (depth = 1)		
Going First Wins	Going Second Wins	Total Wins
16	10	26

Furthermore, we found that when pitting MCTS against Minimax at max difficulty, with an emphasis placed on winning value and 1200 iterations, MCTS won 6 out of 9 games with 2 ties. Unfortunately, due to our implementation, run times were astronomically high leading to the small sample size of games performed. As a result, we cannot conclude that MCTS beats Minimax with statistical cause; however, the results are not unprecedented as it matches our aforementioned expectations.

6 Conclusions

After running both implementations. We found that while the MCST proved to be better than the Minimax due to its thoroughness in going through each possible scenario. However, this also proved to be a detriment as the time it took for the algorithm to go through every scenario every turn ultimately lead the AI to take too longer to make a move. Additionally, we found that player 1 won a higher percentage of games regardless of agent and settings. This implies that player 1 has an advantage in ultimate tic-tac-toe, a conclusion that was extrapolated from normal tic-tac-toe and now verified. Furthermore, we found other studies that share our views as well as similar implementations that came to identical conclusions.

References

[1] Arnav P, (2019) Playing Ultimate Tic-Tac-Toe Using Reinforcement Learning

[2] Eytan L, & Tsurel D Ultimate Tic-Tac-Toe

[3] Phil C, AI AGENTS FOR ULTIMATE TIC-TAC-TOE

[4] Shakak B, (2017) Using Reinforcement Learning to play Ultimate Tic-Tac-Toe