CS63 Fall 2022
Applying Minimax and MCTS to Connect-4 Board Game

Kelvin Darfour, Nelson Dufitimana

1 Introduction

We are currently experiencing what computer scientist Melanie Mitchell calls the "Artificial Intelligence Spring," a period marked by numerous discoveries and breakthroughs in AI. For example, in the field of adversarial games, we have witnessed IBM's Deep Blue defeating chess champion Garry Kasparov in 1997 and DeepMind's AlphaGo victory over Go legend Lee Sedol in 2014. These successes in adversarial games, which are largely due to improved adversarial search algorithms aided by hardware improvement among a few other factors, highlight the advancements made in the field of AI. The work presented in this paper leverages this progress in adversarial search algorithms to play a board game Connect-4. Connect-4, also known as Captain's Mistress, is a two-player connection game played on a 6 x 7 board. The game was first published in 1974 by Milton Bradley. The ultimate goal of a player during the game is to be the first player to gather four chips in a row, either diagonally, vertically, or horizontally. To play, each player takes turns dropping one of their colored chips into the grid, with the chips falling to the bottom of the grid or on top of previously played chips. The game ends when one player has four chips in a row, or when the grid is filled and no player has four chips in a row.

Our work aims to compare the performance of two adversarial search algorithms, Monte Carlo Tree Search (MCTS) and Minimax. To evaluate their performance, we will pit MCTS against Minimax at different search depths and track the results. Additionally, we will provide insight into the performance of MCTS when playing against a human opponent. We predict that MCTS will outperform Minimax if given a sufficient number of rollouts, as MCTS is a Machine Learning algorithm that improves with more data about the game. Furthermore, we expect that the number of rollouts required for MCTS to outperform Minimax will increase as the search depth increases. Therefore, we anticipate that MCTS will require more rollouts to outperform Minimax at a depth of 6 than at a depth of 2, with Minimax at a depth of 4 falling somewhere in between. We believe that our findings will be useful for researchers and practitioners interested in the development and application of these algorithms to adversarial games.

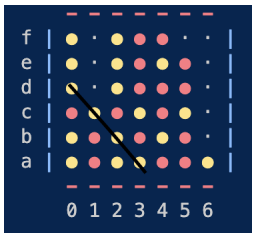

Figure 1: An Example Connect-4 Board

2 Methods

Search algorithms are successful on board games because the game environment possesses three conducive properties: discreteness, staticity, and determinism. In a discrete environment, the number of possible states that a game can be in is finite. A static environment remains unchanged throughout the game, and a deterministic environment guarantees that the next state of the game is solely determined by the moves made by the player in the current state. These properties make search algorithms well-suited for board games because they provide a certain level of predictability to the game environment. The only remaining challenge is the uncertainty introduced by the presence of an opponent whose moves are unknown to the other player. Adversarial search algorithms address this uncertainty by assuming that the opponent is playing optimally and proceeding to perform a state-space search on the game tree to find the most optimal move to make.

2.1 The Game Tree

Adversarial search algorithms construct a search space by starting from the current state of a game and considering all of the possible moves that a player can make from that state. In a game like Connect-4, each node in the search space represents a specific state of the game, and the branches emanating from a node represent the player's possible moves from that state. In Connect-4, a player can make up to 7 possible moves. The leaves of the tree represent terminal states, which are either states where there is a winner or states where the game board is full and no further moves can be made. It's important to note that the perspective of the game switches at each level of the tree; one level shows the perspective of player 1, the next level shows the perspective of player 2, and so on.

2.2 Minimax

Our implementation of the Minimax algorithm relies on a specific representation of the search tree. The necessity of this representation will become clear as we delve into the details of the representation. In our representation, the player who goes first is the maximizer, and the player who goes second is the minimizer. If we reach a winning state, we will return 1 if the player is the maximizer and -1 if the player is the minimizer. A value of 0 is returned for draws. Minimax operates on the search tree in the following way: it starts at the terminal nodes and backs up the values to the root of the tree. If it is the maximizer's turn, Minimax backs up the maximum value from the successors; if it is the minimizer's turn, it backs up the minimum value. When it reaches

the root of the tree, the move that is chosen is the one associated with the maximum value, which is the optimal move.

One practical issue with using Minimax for Connect-4 is that the search tree can become very large. For example, in a standard 6x7 Connect-4 game, the branching factor is 7, and an average game takes about 41 moves[1]. This means that the search tree would have around 7^{41} nodes. In practice, it would not be feasible to search this entire tree using a depth-first search, which is what Minimax does, so the search depth for Minimax is typically limited to a certain number. For the remaining states that are not searched, their value is approximated using a static evaluator function. This results in a depth-bounded Minimax algorithm. During our experiments, the maximum search depth was 6.

Another important aspect of using the Minimax algorithm on search trees like the one we are describing is that there may be some branches of the tree that do not affect the outcome of the game, regardless of their value. Removing these branches can optimize the Minimax algorithm by allowing it to search deeper into the tree. This optimization is called alpha-beta pruning. Alpha and beta are used to represent the best value for the maximizer and the minimizer, respectively. In the beginning, alpha is set to negative infinity, and beta is set to positive infinity. As values are returned from the search, alpha and beta are updated by taking the maximum of the returned value and alpha for the maximizer and taking the minimum of the returned value and beta for the minimizer. If the search reaches a point where alpha is greater than or equal to beta, the search is cut off for the remaining children of the current node. Minimax with alpha-beta pruning always returns the same moves as depth-bounded Minimax, but it is more efficient and faster because it cuts off search branches that do not affect the outcome of the search.

2.3 The Static Evaluator

We applied a static evaluator that assigns a number to each position on the board and that number indicates how good that specific position is. This evaluator scores the board based on how useful a specific the position is to win a game. Higher numbers indicate a high value to winning the game to be while lower numbers indicate a low value in winning the game. The board is scored as follows:

$$\begin{bmatrix} 4 & 5 & 7 & 9 & 7 & 5 & 4 \\ 5 & 8 & 10 & 15 & 10 & 8 & 5 \\ 7 & 10 & 17 & 20 & 17 & 10 & 7 \\ 7 & 10 & 17 & 20 & 17 & 10 & 7 \\ 5 & 8 & 10 & 15 & 10 & 8 & 5 \\ 4 & 5 & 7 & 9 & 7 & 5 & 4 \end{bmatrix}$$

This static evaluator makes middle positions more valuable than corner positions because they allow for more potential connections in different directions. For example, if a player occupies the middle position (2,3), they can potentially make connections in all three different directions: vertical, horizontal, and diagonal. In contrast, if a player occupies a corner position (0,0) or (6,7), the most they can do is make a connection in only one direction. When applied to the Minimax algorithm, this evaluator calculates the value of each player's position and returns the difference between the two values. In the case of a win, it returns hardcoded values: 99999 for the maximizer and -99999 for the minimizer. These represent the highest possible value from both players' perspectives.

2.4 Monte Carlo Tree Search

Monte Carlo Tree Search(MCTS) is another algorithm that combines randomness and tree search to make gameplay decisions in adversarial games. It involves four main steps: selection, expansion, simulation and backpropagation.

1. Selection: Start at the current node and recursively or iteratively select a child node until a non-fully expanded node is reached.
2. Expansion: If the non-fully expanded node in the previous step is not a terminal node, then choose one of its children and call it C.
3. Simulation: Run a simulated rollout from C until the end of the game is reached.
4. Backpropagation: Update the tree with the simulation results up to the node where the next move is being determined.

It is important for MCTS to maintain a balance between exploiting moves with a high average winning rate and moves that have been visited quite a few times. This is achieved by attempting to maximize the Upper Confidence Bound, UCB, which is given by: $v_i + C * \sqrt{(\ln N / n_i)}$ where:

- v_i is the estimated value of the node based on the number of rollouts
- n_i is the number of times the node has been visited during the rollouts
- N is the number of times the parent of the node has been visited
- C is a bias parameter.

From the above function, it is clear that high values of v_i increase the chances of selection of the associated node, which encourages exploitation while higher values of n_i decrease the chances of selection, which increases exploration as a result.

To calculate v_i that is used in the UCB function, the following formula is used: $v_i = 1 + ((wins - losses)/gamesPlayed)$. Because this value should always be positive, 1 is added to enforce that restriction. It is always important that the drawing is given a value higher than that of loss which is very crucial for efficient selection. Value is calculated from the perspective of player 1 and is adjusted for the player by subtracting it from two i.e. $2 - v_i$.

3 Results

For our experimentation, we tested Monte Carlo Tree Search against Minimax at different search depths while varying the number of rollouts for MCTS where each variation was tested for 10 games. We ran Minimax at search depths of 2, 4, and 6, and MCTS at rollouts ranging from 10 to 50,000. The graph below shows the performance of MCTS against Minimax at different depths over time as the number of rollouts increased.

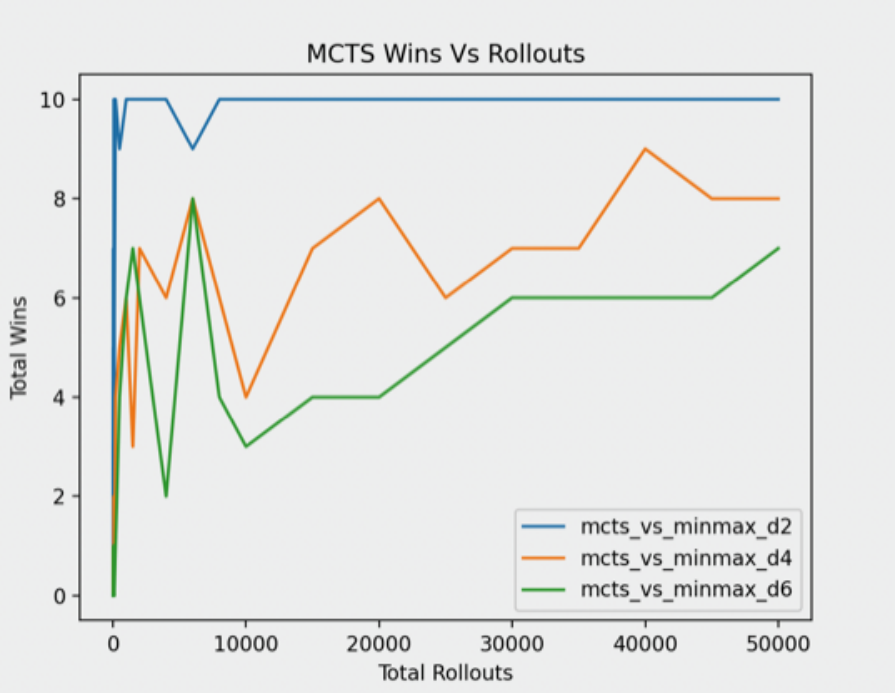

Figure 2: Results from 10 of MCTS against Minimax at different depths

The results showed that MCTS begins to consistently outperform Minimax at a depth of 2 when the number of rollouts is close to 8000. However, MCTS has more difficulty against Minimax at a depth of 4, even with up to 50,000 rollouts. When we increased the search depth of Minimax to 6, MCTS was challenged by Minimax a lot of times. Interestingly, there were instances where MCTS was able to beat Minimax at a search depth of 6 with only 10,000 rollouts. This pattern was observed across all search depths for Minimax, indicated by early spikes in the graphs. One possible explanation for this is that Monte Carlo Tree Search has to balance exploration and exploitation, so it may sometimes explore moves that turn out to be beneficial early on, but shift to exploitation as the number of rollouts increases. For example, since Minimax uses a static evaluator that highly rewards moves in the middle of the board, we speculate that it would tend to play in the middle positions. In such instances, if MCTS explores moves in the corner positions, it may be able to outperform Minimax even with a moderate number of rollouts. However, as we will show, MCTS later realizes that it is more valuable to play in the middle positions and tries to stick to moves that place chips there. More experimentation would be needed to confirm this speculation.

To gain insight into the strategies employed by MCTS, we played a few games between MCTS and a human opponent. MCTS used 5000 rollouts and the human played optimally. From the board snippets below, it is clear that MCTS focuses on controlling the middle positions of the board. This gives it an advantage because it allows MCTS to potentially create multiple winning combinations. Even when the human player tries to distract MCTS by playing in different, unclustered positions, MCTS remains focused on stacking its chips in the middle of the board until it has a sufficient number to create a winning combination or until the opponent is one move shy away from a win. This shows the strategic nature of MCTS and its ability to adapt to the opponent's moves.

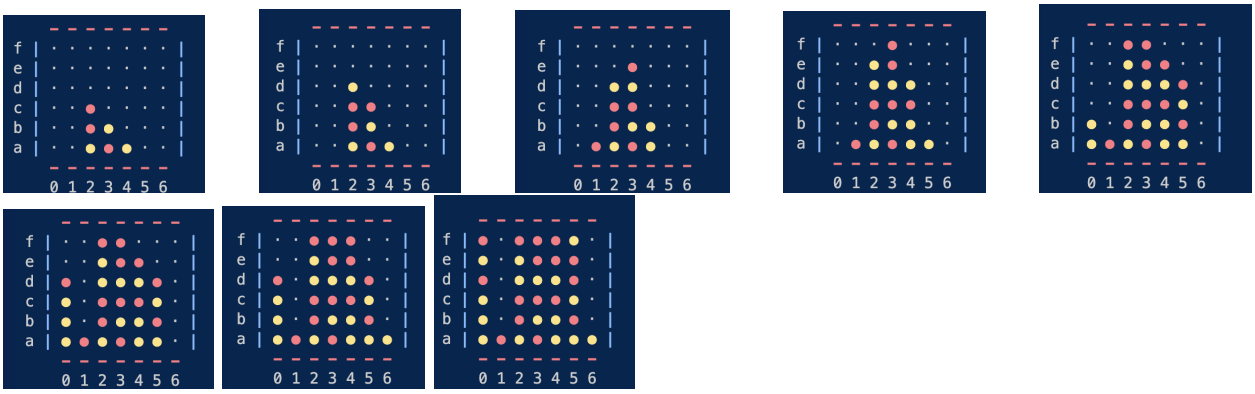

Figure 3: From left to right: snapshots from a game between MCTS(in red) and Human Player(in yellow)

4 Conclusions

In this paper, we compare the performance of Monte Carlo Tree Search (MCTS) and Minimax in the board game Connect-4. We ran experiments where MCTS played 10 games against Minimax at different depths for different numbers of rollouts. The results show that MCTS consistently outperforms Minimax at a search depth of 2 when the number of rollouts is close to 8000. However, MCTS has more difficulty against Minimax at depths of 4 and 6, even with up to 50,000 rollouts. Interestingly, there are instances where MCTS can beat Minimax at all search depths with only 10,000 rollouts. We speculate that this is due to MCTS' balance between exploration and exploitation, as well as the nature of the static evaluator that Minimax uses. We also observed the strategies used by MCTS in games against a human opponent with 5000 rollouts. Our observations indicate that MCTS learns that having many chips in the middle positions of the board is advantageous, and it tries to maintain control over these positions. This allows MCTS to create multiple winning combinations. Although our work shows that MCTS can outperform Minimax, we were unable to find a point where MCTS consistently beat Minimax at any depth. In the future, we can give MCTS more rollouts to capture this point.

References

[1] L.V. Allis. A Knowledge-based Approach of Connect-Four. *ICGA Journal*, 11:165–165, 1988.