William Huang, Yun Lu

CS 5100

Spring 2025

# Connect Four with AI Opponents: Implementation and Analysis

## Introduction

Connect Four is a two-player board game played on a 7×6 grid where players take turns dropping colored discs from the top. The objective is to connect four of one's own discs consecutively (horizontally, vertically, or diagonally). Despite its simple rules, Connect Four offers significant strategic depth that makes it an ideal candidate for exploring artificial intelligence algorithms.

In this project, we developed a Connect Four implementation featuring multiple AI opponents of varying difficulty levels, leveraging different algorithms to create a tiered challenge system. We learned about these algorithms from our literature review in the field of game-playing AI, with a particular focus on algorithmic approaches to Connect Four.

Our literature review focused on two key papers that examine AI techniques for Connect Four. Qiu et al. (2022) conducted a comparative analysis of four AI algorithms commonly used in strategic board games: Greedy Algorithm, Alpha-Beta pruning, Principal Variation Search (PVS), and Monte Carlo Tree Search (MCTS). Through 36 rounds of gameplay with each algorithm playing 12 games against the others, they measured performance using win rate, move efficiency, and computational complexity. Their results confirmed that MCTS was the best-performing algorithm for Connect Four, as its randomized rollouts allowed it to anticipate long-term consequences and adapt better than deterministic search-based methods. PVS ranked second, followed by Alpha-Beta Pruning, with the Greedy Algorithm performing poorest due to its focus on immediate rewards over strategic planning.

Kang et al. (2019) specifically explored how different heuristics affect the minimax algorithm's performance in Connect Four. They evaluated two distinct heuristic approaches: the first heuristic that assessed different board features (four-in-a-row, three-in-a-row, two-in-a-row, and single pieces) and the second heuristic that evaluated each square based on its strategic importance (with central positions valued more highly). Their experiments demonstrated that performance improved with increased search depth, that alpha-beta pruning significantly enhanced runtime efficiency, and that using a combination of all features in the first heuristic yielded the best results.

Based on these findings, we developed three difficulty levels for our AI agent in Connect Four by using three different algorithms. We aim to demonstrate fundamental AI concepts and algorithms by creating an engaging experience for both beginners and experienced players.

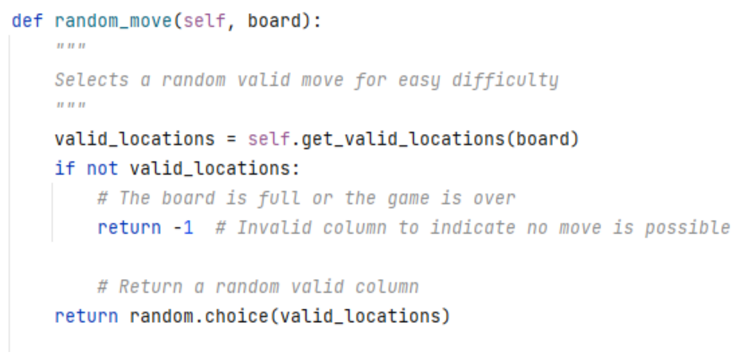
## Methods

We developed the Connect Four game with a graphical user interface on Python using the pygame library. The implementation follows an object-oriented design with three main components: the UI components, the game logic, and the AI decision-making logic.

The UI components were implemented inside of the Connect4Game class and the Button class, which handled user interactions and visual feedback during the game. The game logic was implemented inside the Connect4Game class, which managed game state, turn progression, win detection, and board rendering. Finally, the AI decision-making logic was implemented inside the AIAlgorithm class, which provided different levels of AI decision making through the three different algorithms.

The graphical user interface allows the player to select one of two game modes: single player mode (human vs. AI) or two player mode (human vs. human). If the player selects single player mode, then the player can further select from three difficulty levels (easy, medium, hard) that each employ a different algorithm. If the player selects two player mode, then a two player game starts.

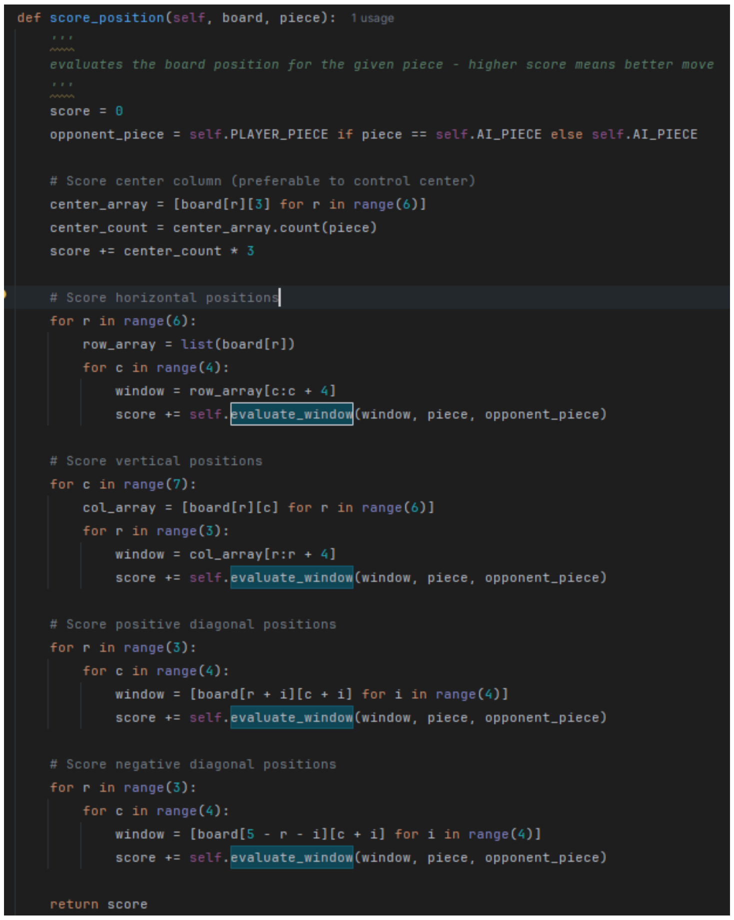
For the easy difficulty, we implemented a random move selector that randomly chooses a move from the remaining available columns. The random selection is shown below:



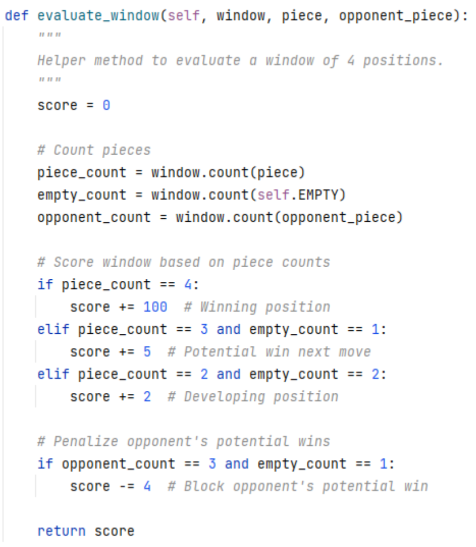
```python
def random_move(self, board):
    """
    Selects a random valid move for easy difficulty
    """
    valid_locations = self.get_valid_locations(board)
    if not valid_locations:
        # The board is full or the game is over
        return -1  # Invalid column to indicate no move is possible

        # Return a random valid column
    return random.choice(valid_locations)
```

For the medium difficulty, we implemented the Minimax algorithm with Alpha-Beta Pruning. Minimax is a recursive decision-making algorithm that simulates all possible game states to find the best move for the AI agent, assuming both players play optimally. Alpha-beta pruning increases efficiency by eliminating suboptimal branches on the Minimax tree.

Our implementation includes an evaluation function that scores the board positions based on three heuristics: different board features (four-in-a-row, three-in-a-row, two-in-a-row, and single pieces), strategically valuable squares, and blocking opponent’s potential winning sequences. We added a terminal state detection mechanism to find immediate wins or losses and improve the efficiency of the decision-making process. We also designed the depth-limited search to be three levels to balance computational efficiency and play strength. Our evaluation function is shown below:



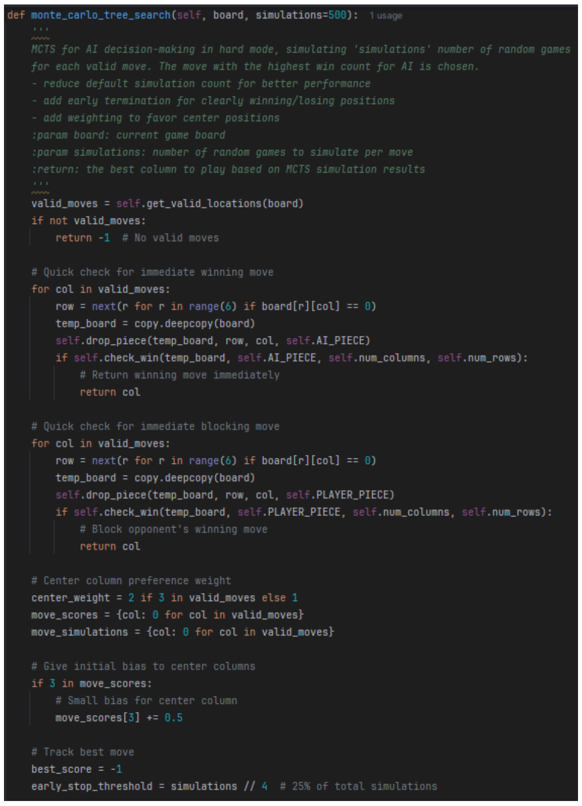
```python
def score_position(self, board, piece):  1 usage
    '''
    evaluates the board position for the given piece - higher score means better move
    '''
    score = 0
    opponent_piece = self.PLAYER_PIECE if piece == self.AI_PIECE else self.AI_PIECE

    # Score center column (preferable to control center)
    center_array = [board[r][3] for r in range(6)]
    center_count = center_array.count(piece)
    score += center_count * 3

    # Score horizontal positions
    for r in range(6):
        row_array = list(board[r])
        for c in range(4):
            window = row_array[c:c + 4]
            score += self.evaluate_window(window, piece, opponent_piece)

    # Score vertical positions
    for c in range(7):
        col_array = [board[r][c] for r in range(6)]
        for r in range(3):
            window = col_array[r:r + 4]
            score += self.evaluate_window(window, piece, opponent_piece)

    # Score positive diagonal positions
    for r in range(3):
        for c in range(4):
            window = [board[r + i][c + i] for i in range(4)]
            score += self.evaluate_window(window, piece, opponent_piece)

    # Score negative diagonal positions
    for r in range(3):
        for c in range(4):
            window = [board[5 - r - i][c + i] for i in range(4)]
            score += self.evaluate_window(window, piece, opponent_piece)

    return score
```



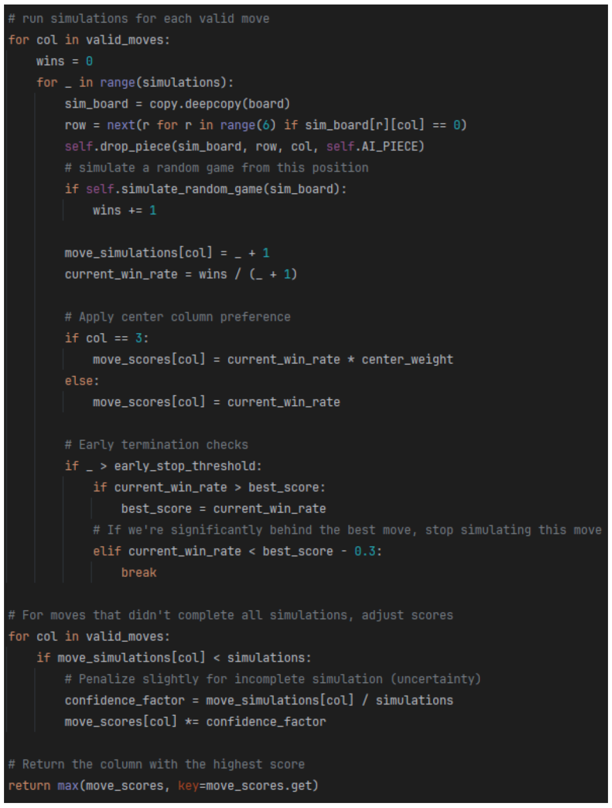
```python
def evaluate_window(self, window, piece, opponent_piece):
    """
    Helper method to evaluate a window of 4 positions.
    """
    score = 0

    # Count pieces
    piece_count = window.count(piece)
    empty_count = window.count(self.EMPTY)
    opponent_count = window.count(opponent_piece)

    # Score window based on piece counts
    if piece_count == 4:
        score += 100  # Winning position
    elif piece_count == 3 and empty_count == 1:
        score += 5  # Potential win next move
    elif piece_count == 2 and empty_count == 2:
        score += 2  # Developing position

    # Penalize opponent's potential wins
    if opponent_count == 3 and empty_count == 1:
        score -= 4  # Block opponent's potential win

    return score
```

The variable piece_count in evaluate_window represents the different board features heuristic. The variable opponent_count in evaluate_window represents the blocking opponent's wins heuristic. The variable center_count in score_position represents the strategically valuable squares heuristic.

Lastly, we implemented Monte Carlo Tree Search (MCTS) for the hard difficulty. MCTS utilizes both Monte Carlo strategies (random sampling and statistical evaluation) and search tree techniques to find the optimal moves for the AI agent. This allows the AI agent to balance between exploiting promising moves and exploring new game states. We implemented this in four steps: finding the selection of promising moves, simulating random games from each promising move, statistically evaluating the win rate for each move, and selecting the move with the highest win probability. To optimize our agent, we included several features in our algorithm. First, we prevented unnecessary simulation calculations with early detection of winning moves and blocking opponent's winning moves. Second, we added a center column bias to promote strategically advantageous play based on our literature review. Third, we improved computational efficiency with early termination of unproductive simulations. Lastly, we used confidence-based scoring for our simulation sample sizes. Our MCTS function simulates hundreds of random games for each possible move as shown below in the simulations input parameter.



```python
def monte_carlo_tree_search(self, board, simulations=500):  1 usage
    '''
    MCTS for AI decision-making in hard mode, simulating 'simulations' number of random games
    for each valid move. The move with the highest win count for AI is chosen.
    - reduce default simulation count for better performance
    - add early termination for clearly winning/losing positions
    - add weighting to favor center positions
    :param board: current game board
    :param simulations: number of random games to simulate per move
    :return: the best column to play based on MCTS simulation results
    '''
    valid_moves = self.get_valid_locations(board)
    if not valid_moves:
        return -1  # No valid moves

    # Quick check for immediate winning move
    for col in valid_moves:
        row = next(r for r in range(6) if board[r][col] == 0)
        temp_board = copy.deepcopy(board)
        self.drop_piece(temp_board, row, col, self.AI_PIECE)
        if self.check_win(temp_board, self.AI_PIECE, self.num_columns, self.num_rows):
            # Return winning move immediately
            return col

    # Quick check for immediate blocking move
    for col in valid_moves:
        row = next(r for r in range(6) if board[r][col] == 0)
        temp_board = copy.deepcopy(board)
        self.drop_piece(temp_board, row, col, self.PLAYER_PIECE)
        if self.check_win(temp_board, self.PLAYER_PIECE, self.num_columns, self.num_rows):
            # Block opponent's winning move
            return col

    # Center column preference weight
    center_weight = 2 if 3 in valid_moves else 1
    move_scores = {col: 0 for col in valid_moves}
    move_simulations = {col: 0 for col in valid_moves}

    # Give initial bias to center columns
    if 3 in move_scores:
        # Small bias for center column
        move_scores[3] += 0.5

    # Track best move
    best_score = -1
    early_stop_threshold = simulations // 4  # 25% of total simulations
```



```python
# run simulations for each valid move
for col in valid_moves:
    wins = 0
    for _ in range(simulations):
        sim_board = copy.deepcopy(board)
        row = next(r for r in range(6) if sim_board[r][col] == 0)
        self.drop_piece(sim_board, row, col, self.AI_PIECE)
        # simulate a random game from this position
        if self.simulate_random_game(sim_board):
            wins += 1

        move_simulations[col] = _ + 1
        current_win_rate = wins / (_ + 1)

        # Apply center column preference
        if col == 3:
            move_scores[col] = current_win_rate * center_weight
        else:
            move_scores[col] = current_win_rate

        # Early termination checks
        if _ > early_stop_threshold:
            if current_win_rate > best_score:
                best_score = current_win_rate
            # If we're significantly behind the best move, stop simulating this move
            elif current_win_rate < best_score - 0.3:
                break

# For moves that didn't complete all simulations, adjust scores
for col in valid_moves:
    if move_simulations[col] < simulations:
        # Penalize slightly for incomplete simulation (uncertainty)
        confidence_factor = move_simulations[col] / simulations
        move_scores[col] *= confidence_factor

# Return the column with the highest score
return max(move_scores, key=move_scores.get)
```

We used multiple assessment approaches to evaluate our implementation. First, we conducted user playtesting sessions to assess the appropriateness of the difficulty levels. Next, we assessed algorithm strength through direct comparisons, measuring win rates in AI vs. AI matches.

## Results

During our algorithm strength assessment, we ran 50 games for each AI vs. AI matchup and alternated first-move advantage to ensure fairness. Table 1 demonstrates the effectiveness of the implemented algorithms based on the algorithm win rates per match up.

Table 1: Win Rates by Algorithm Matchup

| Matchup | Algorithm 1 Win rate % | Algorithm 2 Win rate % | Draw % |
|---|---|---|---|
| Random vs Minimax | 0.0 | 100.0 | 0.0 |
| Random vs MCTS | 0.0 | 100.0 | 0.0 |
| Minimax vs MCTS | 36.0 | 64.00 | 0.00 |

Based on the table, the random move selector (easy mode) was dominated by MiniMax (medium mode) and MCTS (hard mode), as the random move selector did not win a single game against the other algorithms. This confirms that the random move selector is appropriate for entry-level players, providing unpredictable but weak opposition. Next, the Minimax algorithm dominated against the random move selector but was beaten by the more sophisticated MCTS algorithm, as indicated by the 36% win rate against MCTS. This confirms that the Minimax algorithm is appropriate for intermediate-level players, providing a significantly harder challenge for many but still can be beaten by higher level players. Lastly, the Monte Carlo Tree Search algorithm is the strongest algorithm among the three, winning 100% of games against random move selector and 64% of games against Minimax. Our findings match the findings of Qiu et. al. (2022) in that MCTS is the strongest algorithm for Connect Four.

Additionally, we tracked the computational performance of each algorithm by measuring the decision-making time across all test games as shown in Table 2.

Table 2: Computational Performance bay Algorithm Matchup

| Matchup | Total time for 50 games | Time per game (avg.) | Iterations per second |
|---|---|---|---|
| Random vs Minimax | 4 seconds | 0.08 second | 11.92 it/s |
| Random vs MCTS | 6 minutes 39 seconds | 7.99 second | 0.13 it/s |
| Minimax vs MCTS | 8 minutes 43 seconds | 10.48 second | 0.10 it/s |

The computational performance metrics reveal significant differences between the algorithms. The matchup between Random and Minimax algorithms completed all 50 games in just 4 seconds, averaging 0.08 seconds per game and achieving 11.92 iterations per second. This indicates that the games utilizing the Random selection algorithm and the Minimax algorithm with alpha-beta pruning require the least computational overhead.

In contrast, the matchups involving MCTS required substantially more computation time. The Random vs MCTS competition took 6 minutes and 39 seconds to complete 50 games, averaging 7.99 seconds per game. When both strategic algorithms faced each other in the Minimax vs MCTS matchup, the computational demands increased further, requiring 8 minutes and 43 seconds to complete 50 games with an average of 10.48 seconds per game. The slower performance of MCTS vs. Minimax in comparison to MCTS vs. random suggests that more complex board positions arose when two strategic algorithms competed, which required more extensive simulations by the MCTS algorithm.

These performance differences reflect the inherent computational complexity of each algorithm. The Random selection algorithm has the least computational complexity due to the minimal calculations required to make its decisions. The Minimax with alpha-beta pruning algorithm is more computationally complex because it searches for the best move via a pruned search tree. The MCSC algorithm is the most computationally complex due to evaluating hundreds of potential game outcomes for each potential move.

## Conclusions

We successfully implemented a fully functional Connect Four game with three different difficulty levels to accommodate players with various skill levels. The three AI algorithms provided insight into the trade-offs between computational efficiency and gameplay quality. The random selection algorithm (easy mode) was computationally cheap and easy to implement. However, the algorithm had no strategy, causing the weak agent to frequently miss winning opportunities and defensive moves. The Minimax with alpha-beta pruning algorithm (medium mode) had improved gameplay by recognizing immediate tactical opportunities and threats. However, the algorithm's reliance on fixed depth search caused it to miss long-term strategic positions beyond its search depth. The Monte Carlo Tree Search algorithm (hard mode) had the best decision-making performance because of its sophisticated strategic understanding. The MCTS AI agent applies its advanced understanding by consistently controlling the center column, building simultaneous threats, and setting up winning combinations several moves ahead. However, the MCTS algorithm is computationally intensive, so it may cause a bad user experience on older computers with less computation power.

For future improvements, we could improve the graphical user interface to have more aesthetically pleasing menus, a how-to-play menu for beginners, and an in-game pause menu. Additionally, we could improve the run time of the MCTS algorithm by creating a separate cache to store the best move in the current state of the AI, increasing the speed of the hard difficulty games. Creating a cache will reduce the number of computations because the algorithm can quickly find the optimal move in a separate cache instead of recalculating the optimal move for game states it has seen before. The only drawback to this improvement is that the MCTS algorithm will have less exploration and more exploitation.

In conclusion, our current approach offers a well-rounded and functional Connect Four experience, but there still is significant room for improvement by using more advanced AI techniques.

Worked Cited

1. Yiran Qiu, Zihong Wang, Duo Xu(2022). Comparison of Four AI Algorithms in Connect Four. MEMAT 2022; 2nd International Conference on Mechanical Engineering, Intelligent Manufacturing and Automation Technology. Print ISBN:978-3-8007-5761-9 https://ieeexplore.ieee.org/document/9789036
2. Kang, X. Y., Wang, Y. Q., & Hu, Y. R. (2019). Research on Different Heuristics for Minimax Algorithm Insight from Connect-4 Game. Journal of Intelligent Learning Systems and Applications, 11(2), 15-31. https://doi.org/10.4236/jilsa.2019.112002
3. Pygame. (2023). Pygame Documentation. https://www.pygame.org/docs/
4. NumPy. (2023). NumPy Documentation. https://numpy.org/doc/