Intelligent Game Playing: Implementing and Comparing AI Strategies for Connect 4

Melvin Roger
Western University
London, Canada
mroger58@uwo.ca

Tamanna Anandan Nair
Western University
London, Canada
tnair2@uwo.ca

Ilia Mehdizadeh-Hakak
Western University
London, Canada
imehdiza@uwo.ca

Ishan Jain
Western University
London, Canada
ijain5@uwo.ca

Abstract—**This paper presents the design, implementation, and evaluation of multiple artificial intelligence strategies for playing Connect 4, a classic two-player strategy game. We implement and compare four distinct AI approaches: random move selection as a baseline, greedy heuristic evaluation, minimax search without pruning, and minimax with alpha-beta pruning. The heuristic evaluation function considers multiple strategic factors including center column control, threat detection, and winning opportunity analysis. Our experimental results demonstrate that the minimax algorithm with alpha-beta pruning achieves optimal play while maintaining computational efficiency, capable of searching to depth 5 in real-time. The implementation provides a modular, well-documented codebase with both graphical and console interfaces, serving as an educational platform for understanding adversarial search algorithms in game AI.**

Index Terms—**Connect 4, Minimax Algorithm, Alpha-Beta Pruning, Heuristic Evaluation, Game AI, Adversarial Search**

I. Introduction

Game-playing has been a fundamental domain in artificial intelligence research since the field's inception in the 1950s. The development of game-playing programs has driven significant advances in search algorithms, evaluation functions, and computational techniques that have found applications far beyond games themselves. Two-player zero-sum games like chess, checkers, and Connect 4 provide well-defined environments for developing and testing adversarial search algorithms, with clear success metrics and deterministic outcomes.

Connect 4, invented by Howard Wexler and Ned Strongin in 1974 and published by Milton Bradley, presents an interesting balance between simplicity and complexity. While simpler than chess, it offers sufficient depth to demonstrate key AI concepts while remaining computationally tractable for educational purposes. The game is played on a 7-column by 6-row vertical grid where two players alternately drop colored discs into columns. Gravity pulls each disc to the lowest available position within the chosen column, and the objective is to be the first player to form a horizontal, vertical, or diagonal line of four consecutive discs.

Despite its simple rules, Connect 4 has approximately 4.5 trillion possible game positions, making exhaustive search impractical and necessitating intelligent search strategies [6]. This complexity makes it an ideal testbed for comparing different AI approaches, from simple baselines to sophisticated adversarial search techniques.

Our project implements and compares multiple AI strategies of increasing sophistication. We developed four distinct agents: a random baseline agent that selects moves uniformly at random, a greedy heuristic-based agent that evaluates immediate positions, a minimax search agent without optimization, and a minimax agent enhanced with alpha-beta pruning. Through this progression, we demonstrate how each technique improves upon its predecessor in terms of playing strength while analyzing the computational trade-offs involved.

The primary contributions of this work include a modular implementation separating game logic, AI strategies, and user interface components; a comprehensive heuristic evaluation function tailored for Connect 4; empirical comparison of four AI approaches; and an educational platform with both graphical and console interfaces for understanding adversarial search.

The remainder of this paper is organized as follows: Section II reviews related work and provides theoretical background on heuristics, minimax, and alpha-beta pruning. Section III details our methodology and implementation choices. Section IV presents experimental results and analysis comparing the four AI strategies. Section V concludes with a summary of findings and directions for future work.

II. Background and Related Work

A. Game Theory and Zero-Sum Games

Connect 4 belongs to the class of two-player, zero-sum, perfect information games. In such games, one player's gain equals the other player's loss, both players have complete knowledge of the game state, and there is no element of chance. These properties make such games amenable to mathematical analysis and algorithmic solution.

The game-theoretic value of Connect 4 was determined by Victor Allis in 1988, who proved that with perfect play, the first player can always force a win [6]. This was achieved through a combination of knowledge-based approaches and brute-force computation. However, achieving perfect play requires examining positions to significant depth, which motivates the development of efficient search algorithms and accurate heuristic evaluation functions.

B. Heuristic Evaluation Functions

A heuristic in AI is an approximate rule or evaluation function that estimates the quality of a particular state, enabling algorithms to make decisions without exhaustive exploration. As noted in the University of Wisconsin's AI course materials, heuristic evaluation "approximates the true utility of a state when the search must be cut off" [1]. In game-playing contexts, heuristics assign numerical values to board positions, guiding the AI toward favorable outcomes.

The general form of a heuristic evaluation function combines weighted features:

$$evaluation = w_1 f_1 + w_2 f_2 + \cdots + w_n f_n \tag{1}$$

where each feature f_i captures a strategic aspect of the position and w_i represents its weight. The choice of features and their weights significantly impacts the quality of AI decisions. Stanford's AI course emphasizes that heuristics are essential because full game-tree search is often infeasible, requiring approximate evaluation functions to support real-time decisions [2].

For Connect 4, relevant features include the number of pieces in the center column (which participates in more winning lines), the number of open-ended threats (three pieces with an empty fourth position), potential winning configurations, and blocking of opponent threats. The challenge lies in appropriately weighting these features to produce strong play.

Heuristics dramatically improve upon random play by incorporating domain knowledge. While random agents select moves blindly without understanding position quality, heuristic-based agents evaluate and rank options based on meaningful features such as material advantage or positional strength, consistently choosing moves that improve their standing.

C. Minimax Algorithm

The minimax algorithm is a backtracking search algorithm for two-player zero-sum games, first formalized by John von Neumann in 1928 [4]. It models perfect play by assuming one player (MAX) attempts to maximize the evaluation score while the opponent (MIN) attempts to minimize it. The algorithm explores the game tree to a fixed depth, evaluating terminal states and propagating values upward: MAX nodes select the maximum child value, and MIN nodes select the minimum.

The minimax value of a node n is defined recursively as:

$$minimax(n) = \begin{cases} eval(n) & \text{if } n \text{ is terminal} \\ \max_{c \in children(n)} minimax(c) & \text{if MAX's turn} \\ \min_{c \in children(n)} minimax(c) & \text{if MIN's turn} \end{cases} \tag{2}$$

Minimax guarantees optimal play within its search depth by explicitly considering the opponent's best responses. MIT's AI course notes show that minimax performs full-width search, meaning it explores every possible branch, which causes exponential growth in computation: $O(b^d)$ where b is the branching factor and d is the search depth [3]. For Connect 4 with $b \approx 7$, even moderate depths become computationally expensive. At depth 6, the algorithm must examine up to $7^6 = 117,649$ positions.

Compared to heuristic-only approaches, minimax provides superior play by performing lookahead reasoning. A heuristic-only agent makes decisions based solely on the current position without considering opponent responses, potentially falling into traps or missing winning combinations. Minimax identifies sequences leading to guaranteed advantages or avoids forced losses, providing capabilities that pure heuristic evaluation cannot achieve [4].

D. Alpha-Beta Pruning

Alpha-beta pruning is an optimization of minimax that eliminates branches which cannot influence the final decision, developed independently by multiple researchers in the 1950s and 1960s [5]. During search, the algorithm maintains two bounds: α (alpha) representing the best value MAX can guarantee so far, and β (beta) representing the best value MIN can guarantee so far. When $\beta \leq \alpha$, the current branch is pruned because it cannot affect the outcome.

The pruning conditions are:

- At a MAX node: if any child value $\geq \beta$, prune remaining children
- At a MIN node: if any child value $\leq \alpha$, prune remaining children

UC Berkeley's CS188 course notes emphasize that alpha-beta pruning reduces the "effective branching factor," dramatically improving efficiency [5]. In the best case, with optimal move ordering, alpha-beta reduces complexity from $O(b^d)$ to $O(b^{d/2})$, effectively doubling the achievable search depth for the same computational cost. This improvement is significant: for Connect 4, instead of examining 117,649 positions at depth 6, optimal alpha-beta might examine only $\sqrt{117649} \approx 343$ positions.

Move ordering significantly impacts alpha-beta efficiency. By examining promising moves first (such as center columns in Connect 4), more cutoffs occur early, reducing the number of nodes evaluated. Our implementation sorts moves by proximity to the center column to exploit this property, as center moves tend to be stronger in Connect 4 due to their participation in more potential winning lines.

III. Methodology and Implementation

A. System Architecture

Our implementation follows a modular architecture separating concerns into four distinct components, as shown in Table I. We settled on this separation after some initial attempts at putting everything in one file became difficult to debug—isolating the AI logic from the game rules made it much easier to test each component independently.

The game board is represented as a 6×7 two-dimensional array where each cell contains either a space character (empty), 'X' (player one piece), or 'O' (player two or AI piece). We chose this string-based representation because it made debugging output more intuitive—you could just print the board and immediately see the game state. The board uses row

TABLE I
SYSTEM MODULE ORGANIZATION

Module	**Responsibility**
connect4.py	Core game logic and rules
ai.py	AI strategy implementations
GameUI.py	Pygame graphical interface
main.py	Game controller and integration

0 as the top and row 5 as the bottom, with pieces dropping to the lowest available position in each column due to gravity simulation.

B. Game Logic Implementation

The core game logic in `connect4.py` provides essential functions for game state management. The `is_valid_location` function checks whether a column can accept another piece by examining the top row. The `get_next_open_row` function finds the lowest empty position in a column using a bottom-up search. The `winning_move` function checks all possible four-in-a-row configurations: 24 horizontal windows, 21 vertical windows, and 24 diagonal windows (12 in each direction).

Win detection iterates through all valid starting positions for each orientation and checks whether four consecutive cells contain the same piece. This approach runs in constant time $O(1)$ relative to the number of moves played, as the board size is fixed, making it efficient for repeated evaluation during AI search. We initially had a bug in the diagonal detection where we were checking out-of-bounds indices, which took some time to track down before we got the boundary conditions right.

C. Heuristic Evaluation Function

Our heuristic evaluation function (`score_position`) analyzes the board state by examining all possible four-cell windows in horizontal, vertical, and diagonal orientations. Each window is scored based on piece counts according to the weights shown in Table II.

TABLE II
HEURISTIC SCORING WEIGHTS

Window Configuration	**Score**
4 AI pieces (immediate win)	+100
3 AI pieces + 1 empty (strong threat)	+5
2 AI pieces + 2 empty (developing)	+2
3 opponent pieces + 1 empty (must block)	-4
Each piece in center column	+3

Center column control receives additional weight because the center column participates in more potential winning combinations than edge columns. Specifically, a piece in column 3 (center) can participate in 16 different winning lines, while a piece in column 0 or 6 (edges) can participate in only 7 winning lines. This strategic insight significantly improves opening play.

The asymmetric weighting between offensive threats (+5 for three-in-a-row) and defensive blocking (-4 for opponent three-in-a-row) creates an aggressive play style that prioritizes building winning threats while maintaining awareness of opponent attacks. We arrived at these specific values through trial and error—early versions with equal weights played too defensively, while versions with higher offensive weights sometimes ignored obvious threats. The -4 value for blocking seemed to hit the right balance in our testing.

D. AI Strategy Implementations

1) Random AI: The random AI serves as our baseline, selecting uniformly at random from all valid columns. This agent has no strategic understanding and wins only by chance. Its primary purpose is to establish a performance floor for comparison with more sophisticated approaches.

2) Greedy Heuristic AI: The greedy AI evaluates each possible move by simulating it, computing the resulting position's heuristic score, and selecting the move with the highest score. This represents one-ply (single move) lookahead combined with heuristic evaluation. While significantly stronger than random play, this approach cannot anticipate opponent responses or plan multi-move sequences.

3) Minimax AI: Our minimax implementation performs recursive game tree search to a configurable depth (default depth 4). Terminal conditions include: winning position for AI (score +10,000,000), winning position for opponent (score -10,000,000), reaching depth limit (return heuristic score), or no valid moves remaining (draw). We used these large terminal values to ensure that actual wins and losses always dominate heuristic scores—an earlier version used smaller values and occasionally the AI would choose a "better looking" position over an actual forced win.

The implementation creates temporary board copies for each simulated move using the `drop_temp` function, preserving the original board state for backtracking. This functional approach avoids the complexity of explicit undo operations and prevents subtle bugs from state modification side effects.

4) Alpha-Beta AI: The alpha-beta implementation extends minimax with bound tracking and pruning. Initial bounds are set to extreme values ($\alpha = -999999$, $\beta = +999999$). As the search progresses, these bounds tighten based on discovered values. Pruning occurs when $\alpha \geq \beta$, indicating that the current branch cannot improve the final result regardless of remaining moves.

Move ordering optimization sorts candidate columns by distance from the center column before evaluation, examining columns in order: 3, 2, 4, 1, 5, 0, 6. This heuristic significantly increases pruning efficiency because center moves tend to be stronger, causing earlier cutoffs. With this optimization, our implementation achieves depth-5 search with response times under 2 seconds on typical hardware.

E. User Interface Design

The graphical interface uses the Pygame library for rendering and event handling. The UI is completely separated from game logic, communicating through method calls for board display, event processing, and result presentation. This separation enables easy modification of the visual presentation without affecting game mechanics and facilitates potential porting to other graphics frameworks.

The interface provides a complete game experience with multiple screens: a main menu with Start Game and Quit options; game mode selection for Player vs Player or Player vs AI; difficulty selection for AI mode with Easy (random), Normal (greedy), Hard (minimax depth 4), and Very Hard (alpha-beta depth 5); a game board with animated piece dropping preview; and visual feedback including an AI thinking indicator and win/draw announcements.

The board rendering uses a blue grid with circular holes, red pieces for player one, and yellow pieces for player two or AI. Hover effects show where pieces will drop before the player commits to a move, improving the user experience and reducing accidental misclicks. Getting the hover preview to align correctly with the actual drop position took some adjustment to the coordinate calculations.

IV. Experimental Results and Analysis

A. Qualitative Performance Comparison

We evaluated our AI implementations through systematic testing across difficulty levels, observing both playing strength and behavioral characteristics. Table III summarizes the performance characteristics of each implementation.

TABLE III
AI Performance Characteristics

AI Type	Search Depth	Avg. Response	Strength
Random	N/A	$<$1 ms	Baseline
Greedy	1 ply	$<$5 ms	Low
Minimax	4 ply	$\sim$500 ms	Medium
Alpha-Beta	5 ply	$\sim$300 ms	High

The random AI serves as a baseline with expected win rate of approximately 50% against itself in symmetric games. Against human players, it poses no challenge and loses consistently to anyone with basic Connect 4 knowledge.

The greedy heuristic agent significantly outperforms random play by exploiting immediate tactical opportunities. It successfully blocks obvious opponent threats and takes immediate wins when available. However, it remains vulnerable to multi-move combinations and traps that it cannot foresee, often losing to sequences that set up two simultaneous threats.

Minimax agents at depth 4 demonstrate genuine strategic planning, identifying winning sequences and blocking opponent threats several moves ahead. The agent successfully avoids simple traps and can execute basic forced winning combinations. Human players report that the depth-4 minimax presents a moderate challenge, though experienced players can still find ways to outmaneuver it by setting up threats that resolve beyond its search horizon.

The alpha-beta enhanced version achieves identical decision quality to equivalent-depth minimax while enabling depth-5 search within similar time constraints. The additional depth provides noticeably stronger play through improved tactical and strategic analysis, particularly in complex middle-game positions where deeper calculation reveals non-obvious winning paths.

B. Computational Analysis

Alpha-beta pruning demonstrates substantial efficiency gains over naive minimax. At depth 5 with branching factor 7, standard minimax would evaluate approximately $7^5 = 16,807$ positions in the worst case. With alpha-beta pruning and center-first move ordering, typical positions require evaluating only 1,000 to 3,000 nodes, representing a reduction of 80-95%. This improvement enables real-time response while maintaining optimal decision quality within the search depth.

Notably, the alpha-beta implementation at depth 5 often responds faster than the standard minimax at depth 4, despite searching one level deeper. This counterintuitive result occurs because effective pruning eliminates the majority of the search space, and the overhead of maintaining alpha-beta bounds is negligible compared to the savings from avoided node evaluations.

The heuristic evaluation function executes in constant time $O(1)$ relative to board size, examining a fixed number of 69 windows regardless of game state. This efficient design ensures the per-node evaluation cost remains minimal even as search depth increases, making the overall algorithm's performance dominated by the number of nodes visited rather than evaluation cost.

C. Observed Limitations

Our implementation has several limitations we identified during testing. The heuristic weights were manually tuned through experimentation rather than optimized through machine learning or systematic parameter search. Different weight combinations might yield stronger play, and optimal weights may vary depending on the opponent's playing style.

The fixed-depth approach may miss critical tactical situations that require deeper analysis. Positions with forcing sequences (checks in chess terms) benefit from extended search, but our implementation treats all positions equally regardless of tactical complexity. Quiescence search or iterative deepening with time management would address this limitation.

The implementation does not use transposition tables, meaning identical positions reached through different move orders are re-evaluated. In Connect 4, transpositions are common (for example, columns 3 then 4 versus columns 4 then 3 often lead to similar positions), and caching evaluations could significantly reduce computation. We considered adding this but decided it was beyond the scope of the current project.

V. Conclusions and Future Work

This project implemented and compared four AI strategies for Connect 4, demonstrating the progression from random play through heuristic evaluation to adversarial search with alpha-beta pruning. Each technique provided measurable improvements in playing strength, validating the theoretical foundations of game-playing AI.

Our key findings are as follows. Heuristic evaluation provides significant improvement over random play at minimal computational cost. Even simple features like center column preference and threat counting produce noticeably stronger play than random selection. Minimax search enables genuine strategic planning but suffers from exponential complexity growth. The ability to anticipate opponent responses and plan multi-move sequences represents a qualitative improvement over heuristic-only approaches. Alpha-beta pruning maintains decision quality identical to minimax while dramatically reducing computation, enabling deeper search within practical time constraints. With good move ordering, the efficiency gains allow searching approximately twice as deep for the same computational cost.

Future work could explore several directions. Iterative deepening would provide time-bounded search with anytime properties, ensuring the AI always has a reasonable move available while using remaining time to search deeper. Transposition tables would cache and reuse position evaluations, avoiding redundant computation for positions reachable through multiple move orders.

Machine learning approaches could optimize heuristic weights through self-play or tune them against specific opponent types. More ambitiously, replacing the hand-crafted heuristic with a learned evaluation function (as in AlphaZero) could potentially achieve stronger play without manual feature engineering.

Monte Carlo Tree Search (MCTS) techniques that have proven effective in games like Go could also be implemented and compared against our minimax-based approaches. MCTS offers different trade-offs, potentially performing better in positions where accurate heuristic evaluation is difficult.

The modular implementation provides an educational platform for understanding adversarial search algorithms. The clean separation between game logic, AI strategies, and user interface facilitates experimentation with new techniques and straightforward extension to other two-player games such as Tic-Tac-Toe, Othello, or simplified chess variants.

Individual Contributions

Melvin Roger: Implemented the core minimax and alpha-beta algorithms, including debugging an issue where the AI would occasionally prefer heuristically "good" positions over actual wins. Conducted performance benchmarking and contributed to report writing.

Tamanna Anandan Nair: Developed the heuristic evaluation function and spent considerable time testing different weight configurations. The final values came from playing dozens of games against the AI to see how it responded to different situations. Contributed to the methodology section.

Ilia Mehdizadeh-Hakak: Designed and implemented the Pygame graphical user interface, including the menu system and hover preview feature. Handled the coordinate system mapping between screen positions and board indices, which required some debugging to get right. Contributed to testing.

Ishan Jain: Implemented the game logic module including win detection for all orientations. Developed the random and greedy AI baselines and contributed to documentation and code review.

All team members participated in design discussions, code review sessions, and integration testing.

References

[1] University of Wisconsin–Madison, "Game Playing," CS540 Introduction to Artificial Intelligence. [Online]. Available: https://pages.cs.wisc.edu/∼bgibson/cs540/handouts/game_playing.pdf

[2] Stanford University, "Games I: Minimax," CS221 Artificial Intelligence. [Online]. Available: https://web.stanford.edu/class/archive/cs/cs221/cs221.1186/lectures/games1.pdf

[3] MIT OpenCourseWare, "Lecture 6: Search, Games, Minimax, and Alpha-Beta," 6.034 Artificial Intelligence. [Online]. Available: https://web.mit.edu/6.034/wwwbob/handout3-fall11.pdf

[4] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Upper Saddle River, NJ: Prentice Hall, 2020, ch. 5.

[5] UC Berkeley, "Note 5: Games," CS188 Introduction to Artificial Intelligence. [Online]. Available: https://inst.eecs.berkeley.edu/∼cs188/sp24/assets/notes/cs188-sp24-note05.pdf

[6] V. Allis, "A Knowledge-based Approach of Connect-Four," M.S. thesis, Vrije Universiteit Amsterdam, 1988.

[7] Iowa State University, "Alpha-Beta Pruning," CS472 Principles of Artificial Intelligence. [Online]. Available: https://faculty.sites.iastate.edu/jia/files/inline-files/13.alpha-beta-pruning.pdf

[8] Portland State University, "Game Playing and Alpha-Beta Search." [Online]. Available: https://web.pdx.edu/∼arhodes/ai11.pdf

[9] GeeksforGeeks, "Heuristic Search Techniques in AI." [Online]. Available: https://www.geeksforgeeks.org/artificial-intelligence/heuristic-search-techniques-in-ai/

[10] DataCamp, "Minimax Algorithm for AI in Python." [Online]. Available: https://www.datacamp.com/tutorial/minimax-algorithm-for-ai-in-python