Artificial Intelligence Strategies for Connect Four: A Comparative Study of Greedy, Iterative Deepening, Minimax with Alpha-Beta Pruning, and MCTS

Burak Kutlu

IEEE Conference Publishing

Tobb etu

Ankara Türkiye

***Abstract****-Artificial Intelligence (AI) has been widely applied to strategy games to simulate intelligent behavior and develop competitive agents. This study focuses on the implementation and evaluation of four different AI algorithms in the context of the Connect Four game: Greedy Search, Iterative Deepening Depth-First Search, Minimax with Alpha-Beta Pruning, and Monte Carlo Tree Search (MCTS). Each algorithm was integrated into a Python-based Connect Four environment to analyze its performance in terms of move quality, computational efficiency, and overall win rates. The results demonstrate varying trade-offs among these strategies, highlighting how different search paradigms influence decision-making effectiveness in a deterministic, turn-based game environment. This paper aims to provide a comparative understanding of these approaches and their suitability for similar game-playing scenarios.*

1. Introduction

Games have long served as ideal testbeds for developing and evaluating artificial intelligence techniques. Board games, in particular, offer controlled environments with well-defined rules, enabling researchers to explore decision-making, adversarial planning, and heuristic search. Connect Four, a two-player game requiring strategic depth and foresight, provides a suitable platform for experimenting with AI-based game-playing agents.

The purpose of this study is to implement and compare four distinct AI algorithms in the context of Connect Four: Greedy Search, Iterative Deepening Depth-First Search (IDDFS), Minimax enhanced with Alpha-Beta Pruning, and Monte Carlo Tree Search (MCTS). These methods represent a range of search strategies, from shallow heuristic evaluation to deep tree traversal and probabilistic simulations.

This project was implemented in Python, with modular code separating the game mechanics and algorithm logic. Each AI strategy was tested in head-to-head matches under consistent game rules and system settings. The objective is to investigate the strengths and weaknesses of each algorithm in terms of playing strength and efficiency.

1. Game Rules and Environment
2. Game Description

Connect Four is a deterministic, turn-based, two-player game played on a vertical grid with 6 rows and 7 columns. Players alternate turns, dropping a disc into one of the columns. The disc occupies the lowest available position in that column. The objective is to be the first player to form a sequence of four discs either horizontally, vertically, or diagonally. If the board fills up without either player achieving this condition, the game ends in a draw.

Due to its manageable state space and simple rules, Connect Four is well-suited for implementing and evaluating AI strategies. The game’s structure allows for both shallow and deep search approaches, making it ideal for comparing different algorithmic techniques.

1. Implementation Environment

The Connect Four environment was developed using Python 3. Each component of the game was modularized to promote clarity and maintainability. The project consists of the following main modules:

* board.py: This module defines the Connect Four board and encapsulates the core game mechanics, such as move validation, win condition checking, board updates, and heuristic evaluation. It provides a clean interface for interaction with the AI algorithms.
* connect\_four.py: Serves as the main entry point of the application. It manages the game loop, player turns, and AI selection. The script also handles user input and visualizes the board state in the terminal.
* AI Modules:
  + greedy.py: Implements a one-step greedy search that evaluates all possible immediate moves and selects the one with the highest heuristic score.
  + iterative\_deepening.py: Employs depth-limited depth-first search with iterative deepening to explore moves with increasing depth until a preset limit.
  + minimax.py: Implements the Minimax algorithm with Alpha-Beta Pruning to eliminate unnecessary branches during evaluation.
  + mcts.py: Executes Monte Carlo Tree Search using simulation-based rollouts to determine the most promising move.

The modular structure of the project facilitates easy switching between AI strategies and ensures consistent interaction with the game logic. This design also supports the implementation of benchmarking and comparative analysis, as each AI agent adheres to a common interface.

1. AI Algorithms

This study investigates four AI strategies representing a range of search and decision-making paradigms. Each algorithm interacts with the game board through a shared interface, allowing for consistent evaluation and comparison. The following subsections detail the logic, advantages, and limitations of each method.

A. Greedy Search

Greedy Search is a straightforward strategy that evaluates all legal moves based solely on the immediate board state resulting from each action. It employs a static evaluation function to score potential next moves and selects the one with the highest value.

In this project, the greedy agent uses a heuristic that assigns weights to configurations with two or three consecutive discs and penalizes moves that benefit the opponent. While computationally inexpensive and responsive, this method lacks foresight and often fails to block opponent strategies or plan long-term wins.

B. Iterative Deepening Depth-First Search (IDDFS)

IDDFS combines the depth-first traversal approach with iterative deepening, enabling the agent to incrementally search deeper layers of the game tree. At each iteration, the agent performs a depth-limited DFS, increasing the depth limit with each pass until a maximum threshold is reached or time constraints are met.

This technique is particularly useful in situations where a depth-limited search is desired but the appropriate depth is unknown in advance. IDDFS benefits from the low memory footprint of DFS while allowing for progressively deeper strategic planning. However, it may revisit nodes frequently and can be slower than more optimized methods in large trees.

C. Minimax with Alpha-Beta Pruning

The Minimax algorithm is a classical approach used in two-player zero-sum games. It simulates all possible move sequences up to a certain depth, assuming optimal play from both players. The maximizing player attempts to maximize their score, while the minimizing player tries to minimize it.

Alpha-Beta Pruning enhances Minimax by eliminating branches of the game tree that cannot affect the final decision. This significantly reduces the number of nodes evaluated, improving efficiency without compromising optimality.

In the Connect Four implementation, Minimax with Alpha-Beta Pruning uses a heuristic evaluation at terminal and intermediate nodes. The depth is fixed to balance between performance and strategic depth. This approach performs well against both shallow and deep agents but becomes computationally demanding at higher depths.

D. Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search is a probabilistic algorithm that builds a search tree by simulating many random playouts from the current board state. Each node in the tree represents a game state, and edges correspond to legal moves. The algorithm proceeds in four stages: selection, expansion, simulation (rollout), and backpropagation.

MCTS balances exploration and exploitation using the Upper Confidence Bound (UCB1) formula. In this project, the MCTS agent performs a fixed number of simulations per move and chooses the move with the highest average win rate from simulations.

While MCTS does not rely on heuristic evaluations and can handle high branching factors, it is computationally intensive and slower than other methods for time-constrained gameplay. Nevertheless, its performance improves with more simulations, making it a strong contender in longer decision windows.

1. Experimental Setup

**A. System Configuration**

All experiments were conducted on a standard laptop equipped with an Intel Core i5 processor, 8 GB of RAM, and Python 3.x running on a 64-bit operating system. No parallel processing or external libraries (e.g., NumPy, TensorFlow) were used to optimize the performance of AI algorithms, ensuring a fair comparison of computational efficiency based solely on algorithmic design.

**B. Evaluation Metrics**

To evaluate the performance of each AI algorithm, the following metrics were used:

* **Win Rate**: The percentage of games won by each AI when playing against others.
* **Total Win Rate:**
* **Average Move Time**: Time (in seconds) taken per move, measured using Python's time module.

These metrics were chosen to assess both the strategic effectiveness (win rate) and computational cost (move time) of each approach.

**C. Matchups and Testing Protocol**

Each algorithm was tested in round-robin style against every other algorithm. For each pairing, 20 games were played: 10 with the first algorithm going first and 10 with the second. This approach helps minimize first-move bias and allows for a more robust comparison.

**D. Heuristic Evaluation**

A shared heuristic function was employed by the Greedy, Minimax, and Iterative Deepening agents. The function assigns scores based on the number of connected discs in a potential winning line. Specifically:

* 2 in a row (open-ended): +10
* 3 in a row: +100
* 4 in a row (win): +1000
* Opponent threats are similarly penalized with negative values.

This consistent scoring function ensures comparability among heuristic-based methods while allowing MCTS to rely solely on rollouts and win statistics.

1. Results and Discussion

**A. Win Rate Comparison**

The performance of each AI algorithm was first evaluated based on win rates in pairwise matchups. Table I shows the aggregated win percentages across all match combinations.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Agent 1 (First Player) | Agent 2 (Second Player) | Agent 1 Win % | Draw % | Agent 2 Win % |
| Minimax | MCTS | 90 | 0 | 10 |
| Minimax | Greedy | 100 | 0 | 0 |
| Minimax | Iterative Deepening | 100 | 0 | 0 |
| MCTS | Minimax | 0 | 0 | 100 |
| MCTS | Greedy | 20 | 0 | 80 |
| MCTS | Iterative Deepening | 40 | 0 | 60 |
| Greedy | Minimax | 0 | 0 | 100 |
| Greedy | MCTS | 50 | 0 | 50 |
| Greedy | Iterative Deepening | 0 | 100 | 0 |
| Iterative Deepening | Minimax | 0 | 0 | 100 |
| Iterative Deepening | MCTS | 70 | 0 | 30 |
| Iterative Deepening | Greedy | 100 | 0 | 0 |

**Table I: Win Rates (%) Between AI Agents (First Player Advantage Balanced)**

**Table I** highlights the performance of each AI agent in direct matchups, factoring in player order. **Minimax with Alpha-Beta Pruning** dominated all opponents across both roles, maintaining a 100% win rate when playing second and achieving 90–100% when playing first. This indicates that Minimax's strength holds regardless of the first-move advantage.

**MCTS**, on the other hand, was highly sensitive to player order. It failed to secure any wins against Minimax as the first player and only achieved a 10% win rate when playing second. However, it performed significantly better against Greedy (50–80%) and had the upper hand over Iterative Deepening when playing second (60%), though it struggled when playing first (only 40%). This asymmetry shows MCTS’s adaptability in reactive positions but vulnerability when initiating play.

**Greedy Search** performed poorly across all matchups. It was completely outclassed by Minimax and Iterative Deepening, regardless of player order. Its only balanced result was a 50–50 split against MCTS, suggesting it is viable only against similarly shallow strategies, and even then, inconsistently.

**Iterative Deepening** showed improved results over Greedy and some competitiveness against MCTS. It notably achieved a draw (100%) as the second player against Greedy and managed a 70% win rate against MCTS as the first player. Still, it was entirely defeated by Minimax from both sides, indicating that while it’s deeper than Greedy, it still falls short of competing with more advanced algorithms.

**B. Total win**

|  |  |  |
| --- | --- | --- |
| Algorithm | Total win as Agent 1 % | Total win as Agent 2 % |
| Minimax | 96.7 | 100 |
| MCTS | 20 | 30 |
| Greedy | 16.7 | 26.7 |
| Iterative Deepening | 56.7 | 33.3 |

**Table II: Total Win Rates (%) of each algorithm**

Minimax clearly dominated, winning 96.7% of its matches as the first player and 100% as the second.

Iterative Deepening came in second, showing moderate success (56.7%) as the first player but was less effective as the second.

MCTS followed, with 20% win rate as Agent 1 and 30% as Agent 2, highlighting its flexibility but inconsistent performance.

Greedy lagged behind all others, with the lowest win percentages overall.

**C. Average Move Time**

**Table III** reports the average time (in seconds) each agent took to decide on a move. These results reflect the trade-offs between strategic depth and computational speed.

|  |  |
| --- | --- |
| Algorithm | Avg. Time (s) |
| Minimax | 0.79 |
| MCTS | 1.7 |
| Greedy | 0.002 |
| Iterative Deepening | 2.5 |

**Table III: Average Move Time Per Agent**

As expected, **Greedy Search was the fastest**, with an average move time of only 0.002 seconds, but it was also the least effective. **MCTS required the most time per move (1.7 seconds)** due to its reliance on numerous simulations. **Iterative Deepening was the slowest (2.5 seconds)** because of its repeated deepening approach. **Minimax struck a good balance**, taking 0.79 seconds per move while maintaining high win rates.

**D. Strategic Behavior**

Observational analysis during gameplay revealed distinct behavioral patterns among the agents:

* **Greedy** prioritized short-term gain, often selecting moves with immediate benefits. However, it frequently failed to anticipate or block the opponent’s strategies, leading to preventable losses.
* **Iterative Deepening** improved upon Greedy by evaluating moves more deeply, allowing it to detect potential threats and opportunities. Despite this, it occasionally suffered from redundant node expansions, impacting efficiency.
* **Minimax with Alpha-Beta Pruning** demonstrated superior strategic foresight. It consistently blocked threats, planned multi-move sequences, and executed well-coordinated strategies.
* **MCTS** was less consistent in its performance. Although it occasionally found creative moves in complex positions, it often struggled to prioritize immediate tactical threats. Its reliance on randomized simulations made it prone to overlooking straightforward winning or blocking moves, especially when simulation depth was limited.

**E. Draw Rates and Game Length**

Draws were rare across all matchups, occurring exclusively in games where **Greedy** played as the first agent and **Iterative Deepening** played second-resulting in a 100% draw rate in that specific pairing (10 out of 10 matches). No other matches ended in a draw.

In terms of game length, **Greedy**-involved games typically concluded quickly—either due to early defeats caused by weak defensive play or fast, unplanned wins against less reactive opponents.

Conversely, matches between more advanced agents such as **Minimax** and **MCTS** tended to be significantly longer. These games featured prolonged sequences of mutual blocking and calculated maneuvers, showcasing deeper strategic planning and more complex decision-making.

1. Conclusion and Future Work

This study implemented and evaluated four AI strategies for the Connect Four game: Greedy Search, Iterative Deepening DFS, Minimax with Alpha-Beta Pruning, and Monte Carlo Tree Search (MCTS). Each algorithm showcased unique strengths and limitations in terms of strategic depth, response time, and adaptability.

Among the methods, **Minimax with Alpha-Beta Pruning** achieved the best overall performance, combining solid decision-making with computational efficiency. **MCTS** also demonstrated competitive results, especially in long-term planning and adaptability, albeit with a higher computational cost. Simpler methods like **Greedy Search** offered fast responses but were less effective in competitive settings.

This work highlights the importance of balancing strategy depth and computational resources in game-playing AI. It also demonstrates that classical algorithms, when properly optimized, can outperform more modern, simulation-based approaches in specific environments like Connect Four.

**Future Work**

Several directions can extend this project:

* **Parallelization of MCTS**: Implementing multi-threaded simulations could significantly reduce decision time and improve MCTS competitiveness.
* **Dynamic Heuristics**: Integrating machine-learned evaluation functions or reinforcement learning techniques may enhance heuristic-based agents.
* **Adversarial Testing**: Introducing human players or unpredictable heuristics could further validate the robustness of each AI agent.

By combining classical AI techniques with modern enhancements, future work can continue pushing the boundaries of intelligent game-playing systems, not only in Connect Four but also in other domains requiring strategic decision-making.