# Connect 4 Simulator with Minimax

**Allen Solomon** [*1] **Arthur Lee** [*1] **Taeyoon Kim** [*1] **Hengyi Liu** [*1]

## Abstract

Abstraction: This paper presents an implementation of the Minimax algorithm for the Connect-4 game and examines its effectiveness for decision-making in competitive gameplay. We investigate how evaluation functions influence performance and analyze the computational limitations of a naive Minimax approach. To address these limitations, we apply Alpha-Beta pruning to reduce the search space and improve runtime efficiency. To evaluate the system, we conducted a user study involving 50 human participants competing against the Minimax-based AI. The AI achieved a 100% win rate, demonstrating the competitive strength of optimized adversarial search in a solved game. These results highlight dominance of algorithmic game strategies over human intuition in deterministic games such as Connect-4.

## 1. Introduction to Minimax Algorithm

In the modern world, people engage with numerous competitive strategy games. Among them, a key category is zero-sum games, in which one player's gain is exactly balanced by the opponent's loss, resulting in a total net value of zero. These games typically end in one of three possible outcomes: win, lose, or draw.

One of the most influential adversarial search algorithms used to solve zero-sum games is the minimax algorithm. The minimax theorem was first introduced by John von Neumann in 1928 in his work On the Theory of Games of Strategy. It established that in any two-player, zero-sum game, there exists a game value and an optimal strategy for each player. The algorithm later became important in computer science during the late 1950s, when it began to be used for developing game-playing artificial intelligence.

[*]Equal contribution [1]Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, United States. Correspondence to: Taeyoon Kim <tkim1@unc.edu>, Arthur Lee <alee7@unc.edu>.

*Proceedings of the $42^{nd}$ International Conference on Machine Learning*, Vancouver, Canada. PMLR 267, 2025. 

In this paper, we demonstrate the application of the minimax algorithm to Connect-4, a widely known zero-sum strategy game.

### 1.1. What is Connect-4?

Connect-4 is a classic two-player strategy board game played on a vertical grid consisting of **7 columns and 6 rows**. Players alternate turns dropping colored discs into any column, and each disc occupies the lowest available position due to gravity.

The objective is to be the first player to connect four consecutive pieces of one's own color in any direction—

- `horizontal,`
- `vertical,`
- `or diagonal.`

When a player successfully forms a line of four, they win the game. If the board becomes completely filled without either player achieving four in a row, the game ends in a draw.

Once a player achieves four consecutive pieces in a row, that player wins. If the board becomes completely full without either player forming four in a row, the game ends in a draw. Connect-4 is chosen to demonstrate the minimax algorithm because it is a *solved* game—meaning that optimal strategies are known mathematically—and it provides an excellent environment for studying human versus AI performance.

### 1.2. Naive Minimax in Connect-4

The pseudocode illustrates the recursive nature of the Minimax algorithm by evaluating all possible future game states up to a specified search depth. Each recursive call alternates between the maximizing player (AI) and the minimizing player (human), simulating a full game tree exploration. Terminal conditions—such as detecting a win, loss, draw, or reaching the maximum depth—stop further expansion and return an evaluation score. By backtracking through the recursion and propagating the best achievable values upward, the algorithm determines the optimal move based on the assumption that both players will always choose the best

**Algorithm 1** Minimax Algorithm for Connect-4

```
Input: board, depth, maximizingPlayer, maxDepth
score ← EvaluateBoard(board)
if score is a terminal value (win/loss) or board is full or depth = maxDepth then
    return score
end if
if maximizingPlayer = true then
    bestValue ← −∞
    for each column in COLUMN_ORDER do
        if move in column is valid then
            row ← MakeMove(board, column, AI)
            value ← Minimax(board, depth + 1, false, maxDepth)
            UndoMove(board, row, column)
            bestValue ← max(bestValue, value)
        end if
    end for
    return bestValue
else
    bestValue ← +∞
    for each column in COLUMN_ORDER do
        if move in column is valid then
            row ← MakeMove(board, column, HUMAN)
            value ← Minimax(board, depth + 1, true, maxDepth)
            UndoMove(board, row, column)
            bestValue ← min(bestValue, value)
        end if
    end for
    return bestValue
end if
```

**Algorithm 2** Find Best Move

```
Input: board, maxDepth
bestValue ← −∞
bestColumn ← null
for each column in all columns do
    if move in column is valid then
        row ← MakeMove(board, column, AI)
        value ← Minimax(board, 0, false, maxDepth)
        UndoMove(board, row, column)
        if value > bestValue then
            bestValue ← value
            bestColumn ← column
        end if
    end if
end for
return bestColumn
```

possible action. This structure allows Minimax to identify the strongest strategic position among all available moves.

### 1.3. Evaluation Function

The minimax algorithm the AI uses is heavily dependent on how we created the evaluation function. The evaluation function is how the AI assigns a value for each board state.

$$\text{EV} : \text{State} \to \mathbb{R}$$

The terminal states are the most trivial states to assign a value to. If either player reaches the goal of getting four in a row, that game is considered to be decided. Therefore, a terminal state in CONNECT-4 can be defined as any state in which either the human gets 4 in a row, or the AI gets 4four in a row.

Hence, the evaluation function should weight these terminal states accordingly. Winning will always be the most optimal state for the AI to choose, and losing is always the least optimal state for the AI to be in. Therefore, it is fairly easy to simply weight these two possible states as a very large integer and a very small integer respectively, far enough apart such that the AI cannot mistake an immediate win as an average move or an immediate loss as a good move. In our case, the evaluation function sets any winning state (AI win) as +1,000,000, and any losing state (human win) as –1,000,000.

Hence, so far the evaluation function is defined as such:

$$\text{EV} : \text{State} \to \mathbb{R}$$

$$\text{EV}(s) = \begin{cases} +1{,}000{,}000, & \text{if winner}(s) = \text{AI}, \\ -1{,}000{,}000, & \text{if winner}(s) = \text{Human}, \\ H(s), & \text{if } s \text{ is non-terminal.} \end{cases}$$

Now onto non-terminal states. A Minimax agent for CONNECT-4 must have an evaluation function that handles non-terminal cases (ie, a board that has no clear winner yet). In this case, the function must use knowledge of what constitutes an advantage and a disadvantage in CONNECT-4 and by using a heuristic that estimates based on general knowledge of the game.

$$\text{H} : \text{State} \to \mathbb{R}$$

In CONNECT-4, the goal is to get 4 in a row before the other player. Before getting 4 in a row, one must chain together 3 in a row successfully, before that, chain 2, and even before that, chain 1. Successfully chaining more 3 and 2-length chains is a considerable advantage as it allows for more opportunities for the owner of those chains to perhaps

create a winning move for a 4 in a row. Therefore, a simple heuristic could be one that can count how many chains the AI has and the Human, then subtract.

The flaw with this heuristic is that some chains are definitively 'dead'. Specifically, chains that will never constitute a win. For example, take into consideration the following configuration:

OO
XO
XO

This configuration would mean that the AI has a chain of two that the naive heuristic would score positively; however, the value of this chain is essentially zero. As it is blocked by the human. This highlights a situation where this heuristic is much too optimistic as it weighs dead chains.

Instead, our implementation counts only the unblocked chains that both the AI and the Human have. By counting only the unblocked chains, we negate being overly optimistic by counting blocked chains. In addition to that, it allows us to keep in mind the different branches a chain could create. For example, an isolated length-3 chain with no blockers around, is also a set of 3 length-1 chains.

One can also see that a length-3 chain is inherently more valuable than a length-2 chain. If a goal is four in a row, three in a row is the second closest to the goal, two in a row is the third, and so on. Therefore, weighting 3-chains more than 2's or 1's is valuable as it would mean the heuristic would push the agent's sub goals to make 3 and 2 in a row. In this implementation, we weighed it such that heuristic would be:

$$H(s) = 0.01 \cdot N_1(s) + 0.04 \cdot N_2(s) + 0.10 \cdot N_3(s) + C(s),$$

Here $N_1$, $N_2$, and $N_3$ are the differences in the number of chains from AI's coins compared to the Human's.

The board itself acts as an obstacle to 3's and 2's in a row, especially when considering the sides or corners where at least an entire side is permanently blocked to any one chain. This means that playing the sides is on average worse than playing the center columns, as playing the sides of the board actively impedes the amount of chains that can be created from a dropped coin. Therefore, we'll create C(s) which will be a function to direct the AI to play more to the center by counting how many coins it owns in the center.

### 1.4. Performance Analysis of Evaluation Function

Since the board never changes in size, as the standard Connect-4 game is a $6 \times 7$ grid, and we evaluate the entire board regardless of the state, the runtime for $\mathrm{EV}(s)$ is $O(1)$. More specifically, $\mathrm{EV}(s)$ consists of two important components:

- terminal state detection, and
- the non-terminal heuristic $H(s)$, which examines all possible 4-cell windows on the board.

In a standard $6 \times 7$ Connect-4 grid, there are exactly 69 such windows:

- 24 horizontal,
- 21 vertical,
- 12 diagonal (down-right),
- 12 diagonal (up-right).

For each window, the function counts how many AI pieces are present, how many human pieces are present, and whether the window is blocked. All operations are constant time. Therefore, the overall computational complexity of evaluating a single board state is:

$$O(69) = O(1)$$

This constant-time behavior is crucial, because Minimax invokes $\mathrm{EV}(s)$ thousands of times per move as it expands the game tree.

## 2. Optimization: Alpha-Beta Pruning

When applied to the Minimax algorithm, *Alpha–Beta Pruning* allows the AI to reduce the computation time required to search to a given depth by avoiding exploration ("pruning") of branches of the decision tree that are guaranteed to have no effect on the final decision.

For example, suppose we are evaluating a minimizer node $v$ that is a child of the root maximizer node. Assume the root maximizer node already has another child whose value is known to be $\alpha$. If the first child of $v$ produces a value less than $\alpha$, then we do not need to explore the remaining children of $v$, since the minimizer node $v$ will ultimately have a value $\leq \alpha$, which will never be chosen by the maximizer root node. A symmetric case exists for a minimizer root node and a child maximizer node, using $\beta$ instead of $\alpha$.

In our model, this pruning is implemented within the Minimax function, which is extended to include two additional parameters, $\alpha$ and $\beta$:

```
minimax(board, depth, is_maximizing,
        max_depth, alpha, beta)
```

The algorithm follows the same structure as the basic Minimax procedure, except that pruning occurs when values exceed the known bounds. For a maximizer node, the logic is summarized as follows:

**Algorithm 3** Alpha–Beta Update (Maximizer Node)

```
for each possible move do
    row ← MakeMove(board, col, AI)
    val ← Minimax(board, depth+1, false, maxDepth, α, β)
    UndoMove(board, row, col)
    best ← max(val, best)
    if best ≥ β then
        return best {prune remaining children}
    end if
    α ← max(α, best)
end for
```

If the value of a child node becomes greater than or equal to $\beta$, the function immediately returns this value, since continuing the search would not influence the decision made by the parent minimizer node. Otherwise, the bound $\alpha$ is updated, which may allow additional pruning to occur later in the search.

## 3. Experiments

We performed two experiments to examine the running time and performance of the Alpha–Beta Pruning algorithm:

1. **Runtime Comparison** — We measured the running time of the Alpha–Beta Pruning algorithm and compared it to the runtime of the naive Minimax algorithm. The goal was to quantify the improvement in efficiency when pruning is used under identical conditions. We found that Alpha–Beta Pruning reduces running time significantly.

2. **Human vs. AI Gameplay** — We evaluated the strength of the AI in real gameplay by testing it against human participants. The objective was to observe whether humans could reliably challenge or defeat a depth-limited Minimax agent. We found that our AI consistently guarantees at least a tie against human players.

### 3.1. Method

#### 3.1.1. Runtime Comparison

We used Python's `time` library to record the running time for both algorithms, ensuring that the search depth and board positions were identical when testing the two approaches.

#### 3.1.2. Human vs. AI Gameplay

We invited students, including friends and classmates, to play Connect-4 games against our AI. A total of 50 trials were performed, and the outcomes of each game were recorded.

## 4. Results

### 4.1. Runtime Comparison

On average, the running time of the Alpha–Beta Pruning algorithm was approximately 1.5 seconds, while the naive Minimax algorithm took about 70 seconds. The speedup is calculated as:

$$\text{Speedup} = \frac{70}{1.5} \approx 46.7$$

Thus, the Alpha–Beta Pruning algorithm improves the running time by a factor of roughly 47.

### 4.2. Human vs. AI Gameplay

Across all 50 gameplay trials, none of the participants were able to defeat the AI. Most games resulted in AI wins, with only a few ending in draws.

## 5. Conclusion

In conclusion, the Alpha–Beta Pruning algorithm significantly accelerates the Minimax search, reducing computation time by approximately a factor of 47. Additionally, the Minimax-based AI produced moves that human players could not reliably overcome, and when the AI played first, it guaranteed at least a draw.

Future improvements may focus on enhancing the heuristic evaluation function used at the search depth limit, making it more admissible and potentially improving decision quality.

## Software and Data

Our implementation of the Minimax and Alpha–Beta Pruning algorithms for Connect-4 was developed in Python, using standard libraries and executed in a command-line environment. No external datasets were required. For reproducibility, all source code and instructions are provided in an anonymous Git repository, including the search algorithms, heuristic evaluation functions, and experimental framework for runtime testing and human vs. AI gameplay.

For the review process, the code and executable scripts are provided as supplementary material via an github link:

https://github.com/Taeyoon-J/COMP560-FinalProject/tree/main