**Connect 4 with AI (Minimax Algorithm)**

**1. INTRODUCTION**

Connect 4 is a classic two-player strategy game where the objective is to be the first to form a horizontal, vertical, or diagonal line of four of one's own discs. While simple to learn, the game possesses a significant level of strategic depth. This project implements a Connect 4 game where a human player competes against an artificial intelligence (AI) opponent.

The core of this AI is the **Minimax algorithm**, a foundational decision-making algorithm in game theory and artificial intelligence. To enhance its performance and allow it to "think" several moves ahead in a reasonable amount of time, the algorithm is optimized with a technique called **Alpha-Beta Pruning**. This document will deconstruct the AI's logic, from the theoretical underpinnings to the practical code implementation.

**2. PROBLEM STATEMENT**

The primary challenge is to create a computer opponent that is both engaging and challenging to play against. A randomly-playing AI would provide no real competition. Therefore, the AI must be capable of:

1. Evaluating the current state of the game board to determine which player has an advantage.
2. "Looking ahead" to predict potential future moves and their outcomes.
3. Making strategic decisions that maximize its own chances of winning while simultaneously minimizing the human player's chances.
4. Operating within a reasonable time frame, so the human player is not left waiting for the AI's turn.

**3. GOAL**

The goal of this project is to implement a strategic AI for the game of Connect 4. This AI will use the Minimax algorithm with Alpha-Beta Pruning to select the optimal move in any given board state. The difficulty of the AI will be adjustable by changing the "depth" of its search, which corresponds to how many moves it looks into the future.

**4. THEORETICAL BACKGROUND:**

**Artificial Intelligence and Game Playing**

AI in the context of board games like Connect 4 is a form of "narrow AI," designed to solve a specific problem. These games are perfect environments for AI because they are:

* **Deterministic**: The outcome of every move is fixed.
* **Perfect Information**: Both players can see the entire game state at all times.
* **Zero-Sum**: One player's gain is the other player's loss.

The AI's task is to search through a "game tree" of all possible future moves to find the path that leads to the best possible outcome for itself.

**5.ALGORITHM EXPLANATION WITH EXAMPLE**

**Minimax Algorithm**: Imagine the game as a tree, where the current board state is the root. Each branch represents a possible move, leading to a new board state. The Minimax algorithm explores this tree to determine the best move. It operates on two principles:

1. **Maximizing Player (The AI)**: Tries to choose the move that leads to the highest possible score.
2. **Minimizing Player (The Human)**: Is assumed to play optimally, choosing the move that leads to the lowest possible score for the AI.

The algorithm recursively travels down the tree. At the "leaves" of its search (either a game-ending state or the maximum search depth), it uses a **heuristic function** to assign a score to the board state. A high score is good for the AI, a low score is good for the human. As it returns up the tree, it propagates these scores:

* At a **Maximizing** level, it chooses the child with the **maximum** score.
* At a **Minimizing** level, it chooses the child with the **minimum** score.

**Alpha-Beta Pruning Optimization**

Exploring the entire game tree is computationally expensive. Alpha-Beta Pruning is a clever optimization that reduces the number of nodes the algorithm needs to evaluate. It works by keeping track of two values:

* **Alpha**: The best score found so far for the **Maximizing** player.
* **Beta**: The best score found so far for the **Minimizing** player.

The core idea is: if the algorithm is evaluating a move and finds that it will lead to a worse outcome than a move it has already found, it can stop exploring that entire branch of the game tree. This is called "pruning."

**Example:**

* The AI (Maximizer) is considering Move A. It looks ahead and sees that the Human (Minimizer) could respond with a move that results in a score of **10**. So, the AI knows it can achieve at least a score of 10. (Alpha is set to 10).
* The AI then considers Move B. It looks ahead and sees the Human can make a counter-move that results in a score of **5**.
* The AI can immediately **stop** exploring any other counter-moves after Move B. Why? Because it knows the Human will pick the move with the lowest score (5), which is already worse than the 10 it can guarantee from Move A. The rest of the Move B branch is **pruned**.

This allows the AI to search deeper into the game tree in the same amount of time, making it a much stronger opponent.

**6. IMPLEMENTATION AND CODE (PYTHON)**

Below is the Python implementation of the AI logic, translated from the project's TypeScript code.

codePython

import math

import numpy as np

# --- Constants ---

ROWS = 6

COLS = 7

WINNING\_LENGTH = 4

PLAYER\_TOKEN = 1

AI\_TOKEN = 2

EMPTY = 0

# --- Helper Functions ---

def create\_board():

"""Creates an empty game board."""

return np.zeros((ROWS, COLS), dtype=int)

def drop\_piece(board, row, col, piece):

"""Places a piece on the board."""

board[row][col] = piece

def is\_valid\_location(board, col):

"""Checks if a column is not full."""

return board[0][col] == EMPTY

def get\_next\_open\_row(board, col):

"""Finds the next available row in a column."""

for r in range(ROWS - 1, -1, -1):

if board[r][col] == EMPTY:

return r

return None

def get\_valid\_locations(board):

"""Returns a list of all columns that are not full."""

return [c for c in range(COLS) if is\_valid\_location(board, c)]

def check\_win(board, piece):

"""Checks if a player has won the game."""

# Horizontal check

for c in range(COLS - (WINNING\_LENGTH - 1)):

for r in range(ROWS):

if all(board[r][c + i] == piece for i in range(WINNING\_LENGTH)):

return True

# Vertical check

for c in range(COLS):

for r in range(ROWS - (WINNING\_LENGTH - 1)):

if all(board[r + i][c] == piece for i in range(WINNING\_LENGTH)):

return True

# Positive diagonal check

for c in range(COLS - (WINNING\_LENGTH - 1)):

for r in range(ROWS - (WINNING\_LENGTH - 1)):

if all(board[r + i][c + i] == piece for i in range(WINNING\_LENGTH)):

return True

# Negative diagonal check

for c in range(COLS - (WINNING\_LENGTH - 1)):

for r in range(WINNING\_LENGTH - 1, ROWS):

if all(board[r - i][c + i] == piece for i in range(WINNING\_LENGTH)):

return True

return False

def is\_terminal\_node(board):

"""Determines if the game has ended."""

return check\_win(board, PLAYER\_TOKEN) or check\_win(board, AI\_TOKEN) or len(get\_valid\_locations(board)) == 0

# --- AI Logic ---

def evaluate\_window(window, piece):

"""Heuristically scores a 4-piece window."""

score = 0

opp\_piece = PLAYER\_TOKEN if piece == AI\_TOKEN else AI\_TOKEN

if window.count(piece) == 4:

score += 10000

elif window.count(piece) == 3 and window.count(EMPTY) == 1:

score += 10

elif window.count(piece) == 2 and window.count(EMPTY) == 2:

score += 5

if window.count(opp\_piece) == 3 and window.count(EMPTY) == 1:

score -= 80

return score

def score\_position(board, piece):

"""Scores the entire board for the given piece."""

score = 0

# Center column preference

center\_array = [int(i) for i in list(board[:, COLS // 2])]

center\_count = center\_array.count(piece)

score += center\_count \* 6

# Score horizontal, vertical, and diagonal windows

for r in range(ROWS):

for c in range(COLS - (WINNING\_LENGTH - 1)):

window = list(board[r, c:c+WINNING\_LENGTH])

score += evaluate\_window(window, piece)

for c in range(COLS):

for r in range(ROWS - (WINNING\_LENGTH - 1)):

window = list(board[r:r+WINNING\_LENGTH, c])

score += evaluate\_window(window, piece)

for r in range(ROWS - (WINNING\_LENGTH - 1)):

for c in range(COLS - (WINNING\_LENGTH - 1)):

window = [board[r+i][c+i] for i in range(WINNING\_LENGTH)]

score += evaluate\_window(window, piece)

for r in range(WINNING\_LENGTH - 1, ROWS):

for c in range(COLS - (WINNING\_LENGTH - 1)):

window = [board[r-i][c+i] for i in range(WINNING\_LENGTH)]

score += evaluate\_window(window, piece)

return score

def minimax(board, depth, alpha, beta, maximizing\_player):

"""Minimax algorithm with Alpha-Beta Pruning."""

valid\_locations = get\_valid\_locations(board)

is\_terminal = is\_terminal\_node(board)

if depth == 0 or is\_terminal:

if is\_terminal:

if check\_win(board, AI\_TOKEN):

return (None, 10000000)

elif check\_win(board, PLAYER\_TOKEN):

return (None, -10000000)

else: # Game is over, no more valid moves

return (None, 0)

else: # Depth is zero

return (None, score\_position(board, AI\_TOKEN))

if maximizing\_player:

value = -math.inf

column = np.random.choice(valid\_locations)

for col in valid\_locations:

row = get\_next\_open\_row(board, col)

b\_copy = board.copy()

drop\_piece(b\_copy, row, col, AI\_TOKEN)

new\_score = minimax(b\_copy, depth - 1, alpha, beta, False)[1]

if new\_score > value:

value = new\_score

column = col

alpha = max(alpha, value)

if alpha >= beta:

break

return column, value

else: # Minimizing player

value = math.inf

column = np.random.choice(valid\_locations)

for col in valid\_locations:

row = get\_next\_open\_row(board, col)

b\_copy = board.copy()

drop\_piece(b\_copy, row, col, PLAYER\_TOKEN)

new\_score = minimax(b\_copy, depth - 1, alpha, beta, True)[1]

if new\_score < value:

value = new\_score

column = col

beta = min(beta, value)

if alpha >= beta:

break

return column, value

def find\_best\_move(board, difficulty):

"""

Entry point for the AI to find its best move.

The difficulty corresponds to the search depth.

"""

return minimax(board, difficulty, -math.inf, math.inf, True)[0]

**6.OUTPUT:**

****

**7. OUTPUT EXPLANATION**

The primary output of the AI logic is from the find\_best\_move function. This function is called whenever it is the AI's turn to move. It takes the current board state and the selected difficulty (which is simply the search depth for the minimax algorithm) as input.

After the minimax function recursively explores the game tree and determines the most optimal move, find\_best\_move returns a single integer. This integer represents the **column index** (from 0 to 6) where the AI has decided to drop its piece.

This returned column is then used by the main game loop to update the board state, and the turn passes back to the human player. The result is a strategic, calculated move rather than a random one, providing a competitive gameplay experience

**8. RESULTS**

The developed *Connect 4 with AI* system successfully simulates a strategic board game where a human player competes against an intelligent computer opponent.

* The **AI player** makes optimal moves using the **Minimax Algorithm with Alpha-Beta Pruning**, effectively reducing unnecessary computations.
* The AI can analyze multiple future moves ahead, making the gameplay challenging and realistic.
* The **adjustable difficulty level** (depth of search) allows players to set different levels of challenge.
* The system provides **instant feedback**, **error-free move validation**, and **accurate win detection** (horizontal, vertical, and diagonal).
* The experiment shows that as the search depth increases, the AI’s performance improves, producing more strategic decisions.

**Overall Result:**  
The project demonstrates that implementing Minimax with Alpha-Beta Pruning in Connect 4 results in an efficient and competitive AI opponent capable of providing engaging gameplay.

|  |  |
| --- | --- |
| **Git Hub Link of the project and report** | https://github.com/aswathaanalina-19/connect-4-ai |

**9.Reference:**

* Russell, S. & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4th Edition). Pearson.
* GeeksforGeeks. “Minimax Algorithm in Game Theory | Set 1 (Introduction)” — https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/
* Stack Overflow Discussions – “Connect 4 AI using Minimax Algorithm”
* TutorialsPoint. “Alpha-Beta Pruning in Artificial Intelligence.”
* GitHub repositories on *Connect 4 AI Minimax implementations* for reference to logic optimization and performance tuning.