ARTIFICIAL INTELLIGENCE LABORATORY

MINI PROJECT REPORT

REGISTER NUMBER	2117240070317
NAME	SUPRIYA.A
PROJECT TITLE	COUNTER FOUR AI WITH MINIMAX USING ALPHA BETA PRUNING
DATE OF SUBMISSION	
FACULTY IN-CHARGE	Mrs. S. Divya

Signature of Faculty In-charge

Counter Four (Connect Four) AI using Minimax Algorithm with Alpha-Beta Pruning

INTRODUCTION

The Counter Four (Connect Four) AI project demonstrates the application of Artificial Intelligence in a two-player competitive strategy game. The AI uses the Minimax algorithm with Alpha-Beta pruning to analyze possible moves and predict the opponent's strategy. This project illustrates how decision-tree search algorithms enable computers to make optimal choices in adversarial environments.

PROBLEM STATEMENT

To design and implement an AI-based Counter Four (Connect Four) game where the computer uses the Minimax algorithm with Alpha-Beta pruning to evaluate future moves and select the best possible action against the human player.

GOAL

The goal of this project is to build an intelligent Connect Four AI that can make optimal and efficient moves using the Minimax algorithm and Alpha-Beta pruning to reduce computation time while maintaining high accuracy.

THEORETICAL BACKGROUND

The Counter Four AI system is based on the Minimax algorithm, a recursive decision-making technique used in two-player games. It simulates all possible moves, assuming both players play optimally. Alpha-Beta pruning enhances Minimax by removing branches that do not affect the final decision, significantly improving computational efficiency. A heuristic evaluation function is used to score board states, allowing the AI to make intelligent decisions within time constraints.

ALGORITHM EXPLANATION

Step 1: Initialize the board and define the AI and human players.
Step 2: For each move, simulate all possible future states using Minimax.
Step 3: Apply Alpha-Beta pruning to eliminate unnecessary branches.
Step 4: Evaluate board states using a heuristic function.
Step 5: Choose the move that maximizes the AI's advantage.
Step 6: Repeat until the game reaches a win, loss, or draw state.

IMPLEMENTATION AND CODE

```
import numpy as np

import pygame

import sys

import math

import random


# Initialize pygame

pygame.init()


# Game Constants

ROW_COUNT = 6

COLUMN_COUNT = 7

SQUARESIZE = 100

RADIUS = int(SQUARESIZE / 2 - 5)


# Colors

BLUE = (0, 0, 255)

BLACK = (0, 0, 0)

RED = (255, 0, 0)

YELLOW = (255, 255, 0)


# Screen size

width = COLUMN_COUNT * SQUARESIZE

height = (ROW_COUNT + 1) * SQUARESIZE

size = (width, height)
```

```
screen = pygame.display.set_mode(size)

# Fonts
myfont = pygame.font.SysFont("monospace", 75)

# Create the board
def create_board():
    board = np.zeros((ROW_COUNT, COLUMN_COUNT))
    return board

def drop_piece(board, row, col, piece):
    board[row][col] = piece

def is_valid_location(board, col):
    return board[ROW_COUNT - 1][col] == 0

def get_next_open_row(board, col):
    for r in range(ROW_COUNT):
        if board[r][col] == 0:
            return r

def print_board(board):
    print(np.flip(board, 0))

def winning_move(board, piece):
    # Check horizontal
```

```
    for c in range(COLUMN_COUNT - 3):
        for r in range(ROW_COUNT):
            if (board[r][c] == piece and board[r][c + 1] == piece and
                board[r][c + 2] == piece and board[r][c + 3] == piece):
                return True
    # Check vertical
    for c in range(COLUMN_COUNT):
        for r in range(ROW_COUNT - 3):
            if (board[r][c] == piece and board[r + 1][c] == piece and
                board[r + 2][c] == piece and board[r + 3][c] == piece):
                return True
    # Check positively sloped diagonals
    for c in range(COLUMN_COUNT - 3):
        for r in range(ROW_COUNT - 3):
            if (board[r][c] == piece and board[r + 1][c + 1] == piece and
                board[r + 2][c + 2] == piece and board[r + 3][c + 3] == piece):
                return True
    # Check negatively sloped diagonals
    for c in range(COLUMN_COUNT - 3):
        for r in range(3, ROW_COUNT):
            if (board[r][c] == piece and board[r - 1][c + 1] == piece and
                board[r - 2][c + 2] == piece and board[r - 3][c + 3] == piece):
                return True


def evaluate_window(window, piece):
    score = 0
```

```
    opp_piece = 1 if piece == 2 else 2

    if window.count(piece) == 4:
        score += 100
    elif window.count(piece) == 3 and window.count(0) == 1:
        score += 5
    elif window.count(piece) == 2 and window.count(0) == 2:
        score += 2
    if window.count(opp_piece) == 3 and window.count(0) == 1:
        score -= 4
    return score

def score_position(board, piece):
    score = 0

    # Center column preference
    center_array = [int(i) for i in list(board[:, COLUMN_COUNT // 2])]
    center_count = center_array.count(piece)
    score += center_count * 3

    # Horizontal score
    for r in range(ROW_COUNT):
        row_array = [int(i) for i in list(board[r, :])]
        for c in range(COLUMN_COUNT - 3):
            window = row_array[c:c + 4]
            score += evaluate_window(window, piece)
```

```
    # Vertical score
    for c in range(COLUMN_COUNT):
        col_array = [int(i) for i in list(board[:, c])]
        for r in range(ROW_COUNT - 3):
            window = col_array[r:r + 4]
            score += evaluate_window(window, piece)

    # Diagonal score (positive)
    for r in range(ROW_COUNT - 3):
        for c in range(COLUMN_COUNT - 3):
            window = [board[r + i][c + i] for i in range(4)]
            score += evaluate_window(window, piece)

    # Diagonal score (negative)
    for r in range(ROW_COUNT - 3):
        for c in range(COLUMN_COUNT - 3):
            window = [board[r + 3 - i][c + i] for i in range(4)]
            score += evaluate_window(window, piece)

    return score

def get_valid_locations(board):
    valid_locations = []
    for col in range(COLUMN_COUNT):
        if is_valid_location(board, col):
```

```
            valid_locations.append(col)
    return valid_locations


def is_terminal_node(board):
    return winning_move(board, 1) or winning_move(board, 2) or len(get_valid_locations(board))
== 0


def minimax(board, depth, alpha, beta, maximizingPlayer):
    valid_locations = get_valid_locations(board)
    terminal = is_terminal_node(board)
    if depth == 0 or terminal:
        if terminal:
            if winning_move(board, 2):
                return (None, 100000000000000)
            elif winning_move(board, 1):
                return (None, -10000000000000)
            else:
                return (None, 0)
        else:
            return (None, score_position(board, 2))
    if maximizingPlayer:
        value = -math.inf
        column = 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, 2)
```

```
            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:
        value = math.inf
        column = 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, 1)
            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 draw_board(board):
    for c in range(COLUMN_COUNT):
```

```
        for r in range(ROW_COUNT):
            pygame.draw.rect(screen, BLUE, (c * SQUARESIZE, r * SQUARESIZE + SQUARESIZE, SQUARESIZE, SQUARESIZE))
            pygame.draw.circle(screen, BLACK, (int(c * SQUARESIZE + SQUARESIZE / 2), int(r * SQUARESIZE + SQUARESIZE + SQUARESIZE / 2 - SQUARESIZE / 2)), RADIUS)

    for c in range(COLUMN_COUNT):
        for r in range(ROW_COUNT):
            if board[r][c] == 1:
                pygame.draw.circle(screen, RED, (int(c * SQUARESIZE + SQUARESIZE / 2), height - int(r * SQUARESIZE + SQUARESIZE / 2)), RADIUS)
            elif board[r][c] == 2:
                pygame.draw.circle(screen, YELLOW, (int(c * SQUARESIZE + SQUARESIZE / 2), height - int(r * SQUARESIZE + SQUARESIZE / 2)), RADIUS)
    pygame.display.update()

board = create_board()
game_over = False
turn = 0

draw_board(board)
pygame.display.update()

# Game Loop
while not game_over:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

```
if event.type == pygame.MOUSEMOTION:
    pygame.draw.rect(screen, BLACK, (0, 0, width, SQUARESIZE))
    posx = event.pos[0]
    if turn == 0:
        pygame.draw.circle(screen, RED, (posx, int(SQUARESIZE / 2)), RADIUS)
    pygame.display.update()

if event.type == pygame.MOUSEBUTTONDOWN:
    pygame.draw.rect(screen, BLACK, (0, 0, width, SQUARESIZE))
    if turn == 0:
        posx = event.pos[0]
        col = int(math.floor(posx / SQUARESIZE))

        if is_valid_location(board, col):
            row = get_next_open_row(board, col)
            drop_piece(board, row, col, 1)

            if winning_move(board, 1):
                label = myfont.render("Player wins!", 1, RED)
                screen.blit(label, (40, 10))
                game_over = True

            turn += 1
            turn %= 2
            draw_board(board)
```

```
# AI Turn
if turn == 1 and not game_over:
    col, minimax_score = minimax(board, 4, -math.inf, math.inf, True)

    if is_valid_location(board, col):
        pygame.time.wait(500)
        row = get_next_open_row(board, col)
        drop_piece(board, row, col, 2)

        if winning_move(board, 2):
            label = myfont.render("AI wins!", 1, YELLOW)
            screen.blit(label, (40, 10))
            game_over = True

        draw_board(board)
        turn += 1
        turn %= 2

if game_over:
    pygame.time.wait(3000)
```

OUTPUT

The output displays the Connect Four grid after each move. The AI selects the most strategic column based on the Minimax evaluation. The user can play interactively against the AI, observing how it blocks potential wins and creates its own winning opportunities.

RESULTS AND FUTURE ENHANCEMENT

The project successfully demonstrates how the Minimax algorithm with Alpha-Beta pruning can be used to develop an intelligent and efficient game-playing AI. Future improvements could include a graphical interface using Pygame or JavaFX, deeper heuristic evaluation for advanced difficulty levels, and integration of reinforcement learning for adaptive gameplay.

LITERATURE SURVEY

- 1. Russell & Norvig (2010) – *Artificial Intelligence: A Modern Approach*
- This book explains the Minimax algorithm as a key AI decision-making method used in two-player games. It helps the AI choose the best possible move by predicting the opponent's responses.
-
- 2. GeeksforGeeks – *Minimax Algorithm in Game Theory*
- This source gives a practical explanation of how Minimax works in games. It shows examples and steps for implementing Minimax logic in programming.
-
- 3. TutorialsPoint – *Alpha-Beta Pruning in AI*
- It explains how Alpha-Beta pruning improves Minimax by removing unnecessary paths in the search tree, making the algorithm faster and more efficient.

REFERENCES

1. Russell, S., & Norvig, P. (2010). Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall.
2. GeeksforGeeks. Minimax Algorithm in Game Theory | Set 1 (Introduction). https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction
3. TutorialsPoint. Alpha-Beta Pruning in Artificial Intelligence. https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_alpha_beta_pruning.htm