**AI Project Documentation**

**Aim :** To Create Checker’s Game

**Algorithm Used :** Minimax Algorithm with Alpha-Beta Pruning

The Minimax algorithm helps the AI (computer) make the most strategic moves by evaluating future board states. It works by assuming that the human player will try to minimize the AI's score, while the AI attempts to maximize its own score.

How It Works:

Maximizing Player (AI - Red): The AI aims to choose moves that give it the highest score.

Minimizing Player (Human - Green): The human player is assumed to choose moves that minimize the AI’s score.

The game simulates several future moves for both players to evaluate possible outcomes.

Alpha-Beta Pruning:

To speed up the algorithm, alpha-beta pruning is used. This optimization reduces the number of possible moves the AI needs to evaluate by skipping irrelevant branches of the game tree, making the AI faster while maintaining accuracy.

Use in the Game:

The algorithm looks 3 moves ahead to determine the best possible move.

The evaluation function assigns scores based on the number of remaining pieces, kings, and other factors.

The AI selects the optimal move after evaluating the possible future outcomes.

**Program :**

import pygame

import copy

pygame.init()

WIDTH, HEIGHT = 600, 600

ROWS, COLS = 8, 8

SQUARE\_SIZE = WIDTH // COLS

WHITE = (255, 255, 255)

BLACK = (0, 0, 0)

RED = (255, 0, 0)

GREEN = (0, 255, 0)

GOLD = (255, 215, 0)

FONT = pygame.font.SysFont('Arial', 40)

WIN = pygame.display.set\_mode((WIDTH, HEIGHT))

pygame.display.set\_caption('Checkers')

class Piece:

PADDING = 10

OUTLINE = 2

def \_\_init\_\_(self, row, col, color):

self.row = row

self.col = col

self.color = color

self.king = False

self.x = 0

self.y = 0

self.calc\_pos()

def calc\_pos(self):

self.x = SQUARE\_SIZE \* self.col + SQUARE\_SIZE // 2

self.y = SQUARE\_SIZE \* self.row + SQUARE\_SIZE // 2

def make\_king(self):

self.king = True

def draw(self, win):

radius = SQUARE\_SIZE // 2 - self.PADDING

pygame.draw.circle(win, self.color, (self.x, self.y), radius)

pygame.draw.circle(win, GOLD, (self.x, self.y), radius + self.OUTLINE) if self.king else None

def move(self, row, col):

self.row = row

self.col = col

self.calc\_pos()

class Board:

def \_\_init\_\_(self):

self.board = []

self.green\_left = self.red\_left = 12

self.green\_kings = self.red\_kings = 0

self.create\_board()

def draw\_squares(self, win):

win.fill(WHITE)

for row in range(ROWS):

for col in range(row % 2, COLS, 2):

pygame.draw.rect(win, BLACK, (col \* SQUARE\_SIZE, row \* SQUARE\_SIZE, SQUARE\_SIZE, SQUARE\_SIZE))

def move(self, piece, row, col):

self.board[piece.row][piece.col], self.board[row][col] = self.board[row][col], self.board[piece.row][piece.col]

piece.move(row, col)

if row == ROWS - 1 or row == 0:

piece.make\_king()

if piece.color == GREEN:

self.green\_kings += 1

else:

self.red\_kings += 1

def get\_piece(self, row, col):

return self.board[row][col]

def create\_board(self):

for row in range(ROWS):

self.board.append([])

for col in range(COLS):

if (row + col) % 2 == 1:

if row < 3:

self.board[row].append(Piece(row, col, RED))

elif row > 4:

self.board[row].append(Piece(row, col, GREEN))

else:

self.board[row].append(0)

else:

self.board[row].append(0)

def draw(self, win):

self.draw\_squares(win)

for row in range(ROWS):

for col in range(COLS):

piece = self.board[row][col]

if piece != 0:

piece.draw(win)

def remove(self, pieces):

for piece in pieces:

self.board[piece.row][piece.col] = 0

if piece.color == RED:

self.red\_left -= 1

else:

self.green\_left -= 1

def winner(self):

if self.red\_left <= 0:

return 'Green'

elif self.green\_left <= 0:

return 'Red'

return None

def get\_all\_pieces(self, color):

pieces = []

for row in self.board:

for piece in row:

if piece != 0 and piece.color == color:

pieces.append(piece)

return pieces

class Game:

def \_\_init\_\_(self, win):

self.\_init()

self.win = win

def \_init(self):

self.selected = None

self.board = Board()

self.turn = GREEN

self.valid\_moves = {}

def update(self):

self.board.draw(self.win)

pygame.display.update()

def reset(self):

self.\_init()

def select(self, row, col):

if self.selected:

result = self.\_move(row, col)

if not result:

self.selected = None

self.select(row, col)

piece = self.board.get\_piece(row, col)

if piece != 0 and piece.color == self.turn:

self.selected = piece

self.valid\_moves = self.\_get\_valid\_moves(piece)

return True

return False

def \_move(self, row, col):

piece = self.selected

if (row, col) in self.valid\_moves:

self.board.move(piece, row, col)

skipped = self.valid\_moves[(row, col)]

if skipped:

self.board.remove(skipped)

self.change\_turn()

else:

return False

return True

def \_get\_valid\_moves(self, piece):

moves = {}

left = piece.col - 1

right = piece.col + 1

row = piece.row

if piece.color == GREEN or piece.king:

moves.update(self.\_traverse\_left(row - 1, max(row - 3, -1), -1, piece.color, left))

moves.update(self.\_traverse\_right(row - 1, max(row - 3, -1), -1, piece.color, right))

if piece.color == RED or piece.king:

moves.update(self.\_traverse\_left(row + 1, min(row + 3, ROWS), 1, piece.color, left))

moves.update(self.\_traverse\_right(row + 1, min(row + 3, ROWS), 1, piece.color, right))

return moves

def \_traverse\_left(self, start, stop, step, color, left, skipped=[]):

moves = {}

last = []

for r in range(start, stop, step):

if left < 0:

break

current = self.board.get\_piece(r, left)

if current == 0:

if skipped and not last:

break

elif skipped:

moves[(r, left)] = last + skipped

else:

moves[(r, left)] = last

if last:

if step == -1:

row = max(r - 3, 0)

else:

row = min(r + 3, ROWS)

moves.update(self.\_traverse\_left(r + step, row, step, color, left - 1, skipped=last))

moves.update(self.\_traverse\_right(r + step, row, step, color, left + 1, skipped=last))

break

elif current.color == color:

break

else:

last = [current]

left -= 1

return moves

def \_traverse\_right(self, start, stop, step, color, right, skipped=[]):

moves = {}

last = []

for r in range(start, stop, step):

if right >= COLS:

break

current = self.board.get\_piece(r, right)

if current == 0:

if skipped and not last:

break

elif skipped:

moves[(r, right)] = last + skipped

else:

moves[(r, right)] = last

if last:

if step == -1:

row = max(r - 3, 0)

else:

row = min(r + 3, ROWS)

moves.update(self.\_traverse\_left(r + step, row, step, color, right - 1, skipped=last))

moves.update(self.\_traverse\_right(r + step, row, step, color, right + 1, skipped=last))

break

elif current.color == color:

break

else:

last = [current]

right += 1

return moves

def change\_turn(self):

self.valid\_moves = {}

if self.turn == GREEN:

self.turn = RED

else:

self.turn = GREEN

# Evaluation function for Minimax

def evaluate(board):

return board.green\_left - board.red\_left + (board.green\_kings \* 0.5 - board.red\_kings \* 0.5)

# Get all valid moves for a color

def get\_all\_moves(board, color):

moves = []

for piece in board.get\_all\_pieces(color):

valid\_moves = game.\_get\_valid\_moves(piece)

for move, skip in valid\_moves.items():

temp\_board = copy.deepcopy(board)

temp\_piece = temp\_board.get\_piece(piece.row, piece.col)

temp\_board.move(temp\_piece, move[0], move[1])

if skip:

temp\_board.remove(skip)

moves.append(temp\_board)

return moves

# Minimax algorithm with alpha-beta pruning

def minimax(position, depth, max\_player, alpha, beta, game):

if depth == 0 or position.winner() is not None:

return evaluate(position), position

if max\_player:

max\_eval = float('-inf')

best\_move = None

for move in get\_all\_moves(position, GREEN):

evaluation = minimax(move, depth - 1, False, alpha, beta, game)[0]

max\_eval = max(max\_eval, evaluation)

alpha = max(alpha, evaluation)

if beta <= alpha:

break

if max\_eval == evaluation:

best\_move = move

return max\_eval, best\_move

else:

min\_eval = float('inf')

best\_move = None

for move in get\_all\_moves(position, RED):

evaluation = minimax(move, depth - 1, True, alpha, beta, game)[0]

min\_eval = min(min\_eval, evaluation)

beta = min(beta, evaluation)

if beta <= alpha:

break

if min\_eval == evaluation:

best\_move = move

return min\_eval, best\_move

# Main game loop

game = Game(WIN)

def main():

run = True

clock = pygame.time.Clock()

while run:

clock.tick(60)

if game.turn == RED:

value, new\_board = minimax(game.board, 3, False, float('-inf'), float('inf'), game)

game.board = new\_board

game.change\_turn()

for event in pygame.event.get():

if event.type == pygame.QUIT:

run = False

if event.type == pygame.MOUSEBUTTONDOWN:

pos = pygame.mouse.get\_pos()

row, col = pos[1] // SQUARE\_SIZE, pos[0] // SQUARE\_SIZE

game.select(row, col)

game.update()

winner = game.board.winner()

if winner:

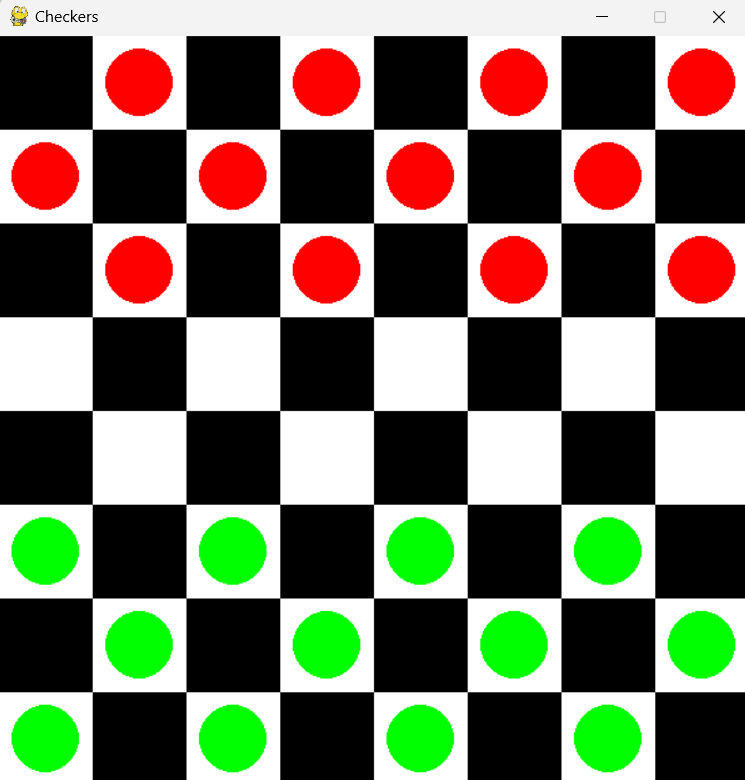
print(f'{winner} wins!')

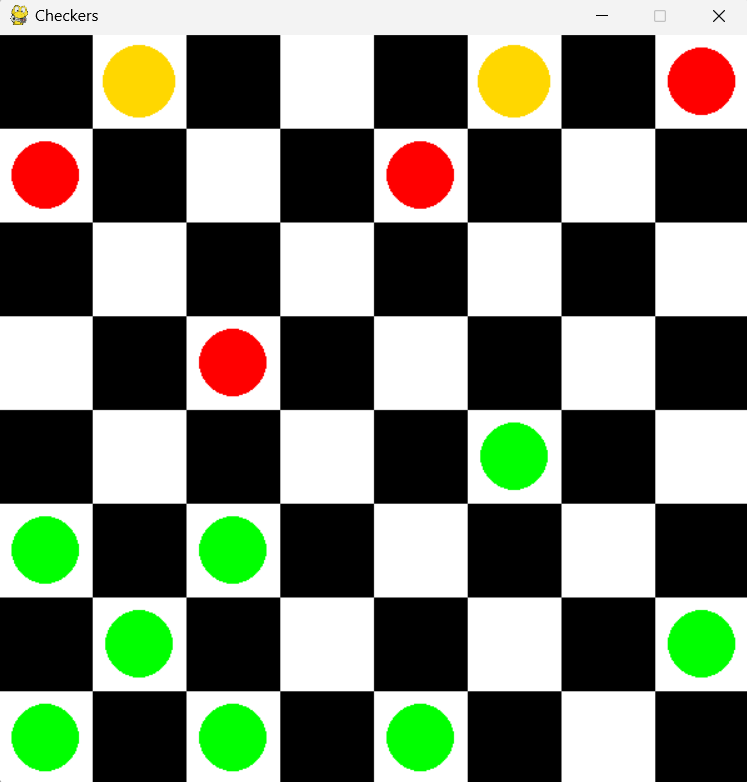
run = False

pygame.quit()

main()

**Output :**







**Conclusion :** The program to create a Checker’s game has been executed successfully