CS2203: Artificial Intelligence

Mini Project – Connect4

Date: 06/04/2025

Group Members:

- **Neetish Kumar** - 2301AI15
- **Ankesh Kumar** - 2301CS06
- **Eshan Bhaskar** - 2301CS16
- **Krishan Kumawat** - 2301CS25
- **Swapnil Sahoo** - 2301CS58
- **Raushan Raj** - 2301CS82

ai_agent.py

```python
import numpy as np
from board import Board
import random
import math

class Connect4AI:
    def __init__(self, max_depth=4, strategy='minimax'):
        self.max_depth = max_depth
        self.strategy = strategy
        self.temp = 1.0  # for SA

    def get_move(self, board):
        # first check if there are any valid moves
        valid_moves = board.get_valid_moves()
        if not valid_moves:
            return None

        if self.strategy == 'minimax':
            move = self.get_best_move_minimax(board)
        elif self.strategy == 'hill_climbing':
            move = self.hill_climbing_move(board)
        elif self.strategy == 'simulated_annealing':
            move = self.simulated_annealing_move(board)

        # if the strategy didn't return a valid move, returning first valid move
        if move is None or not board.is_valid_move(move):
            return valid_moves[0]

        return move
```

```python

    def get_best_move_minimax(self, board):
        best_score = float('-inf')
        best_move = None
        alpha = float('-inf')
        beta = float('inf')

        for col in board.get_valid_moves():
            board.make_move(col)
            score = self.minimax(board, self.max_depth - 1, False, alpha, beta)
            board.undo_move(col)

            if score > best_score:
                best_score = score
                best_move = col
            alpha = max(alpha, best_score)

        return best_move
```

```python
    def hill_climbing_move(self, board):
        # start with all valid moves and their scores
        moves = board.get_valid_moves()
        if not moves:
            return None

        # evaluate all initial moves
        move_scores = []
        for move in moves:
            score = self.evaluate_position(board, move)
            move_scores.append((move, score))

        # start with best move
        current_move, current_score = max(move_scores, key=lambda x: x[1])

        # trying to find better moves for several iterations
        for _ in range(10):
            # look at all neighboring moves
            better_move_found = False
            for move in moves:
                if move == current_move:
                    continue

                score = self.evaluate_position(board, move)
                if score > current_score:
                    current_move = move
                    current_score = score
                    better_move_found = True

            if not better_move_found:  # local maximum reached
                break

        return current_move
```

```python
    def simulated_annealing_move(self, board):
        current_move = random.choice(board.get_valid_moves())
        current_score = self.evaluate_position(board, current_move)
        best_move = current_move
        best_score = current_score
        temp = self.temp

        for _ in range(20):  # number of iterations
            if temp < 0.1:
                break

            next_move = random.choice(board.get_valid_moves())
            next_score = self.evaluate_position(board, next_move)

            # calculating probability of accepting worse move
            delta = next_score - current_score
            if delta > 0 or random.random() < math.exp(delta / temp):
                current_move = next_move
                current_score = next_score

                if current_score > best_score:
                    best_move = current_move
                    best_score = current_score

            temp *= 0.9  # cooling function

        return best_move
```

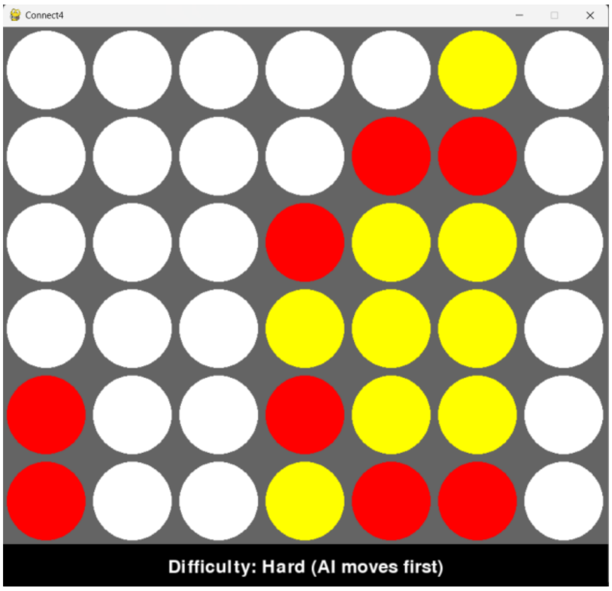
```python
    def evaluate_position(self, board, move):
        score = 0
        # make the move temporarily
        board.make_move(move)

        # heuristic
        score += self.evaluate_center_control(board) * 3
        score += self.evaluate_winning_potential(board) * 10
        score += self.evaluate_blocking_opponent(board) * 8
        score += self.evaluate_connectivity(board) * 5

        # undo the move
        board.undo_move(move)
        return score

    def evaluate_center_control(self, board):
        # evaluate control of center columns
        center_col = board.cols // 2
        center_count = 0
        for row in range(board.rows):
            if board.board[row][center_col] == 2:  # ai pieces
                center_count += 1
        return center_count

    def evaluate_winning_potential(self, board):
        # evaluate potential winning moves
        score = 0
        # check for potential wins in next move
        for col in board.get_valid_moves():
            board.make_move(col)
            if board.check_winner() == 2:  # ai wins
                score += 100
            board.undo_move(col)
        return score
```

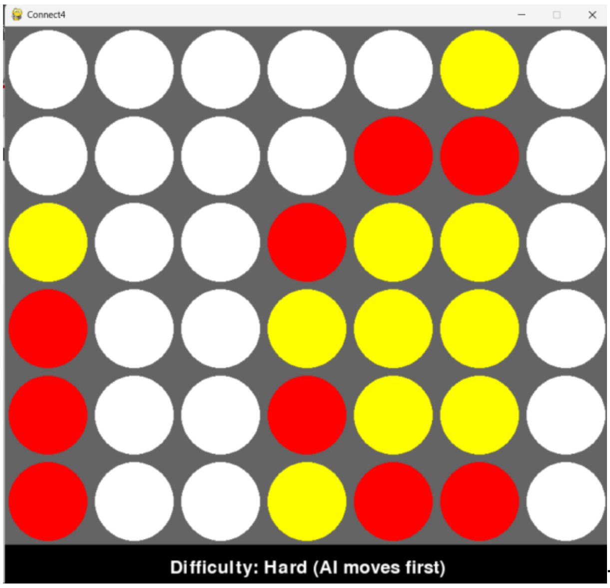
```python
    def evaluate_blocking_opponent(self, board):
        # evaluate blocking opponent's winning moves
        score = 0
        # checking if opponent would win in their next move
        for col in board.get_valid_moves():
            board.make_move(col)
            board.current_player = 1  # simulating opponent's turn
            for opp_col in board.get_valid_moves():
                board.make_move(opp_col)
                if board.check_winner() == 1:  # opponent would win
                    score -= 50
                board.undo_move(opp_col)
            board.current_player = 2  # reset to ai's turn
            board.undo_move(col)
        return score

    def evaluate_connectivity(self, board):
        # evaluate piece connectivity for potential future wins
        score = 0
        directions = [(0,1), (1,0), (1,1), (1,-1)]

        for row in range(board.rows):
            for col in range(board.cols):
                if board.board[row][col] == 2:  # ai piece
                    for dr, dc in directions:
                        connected = 0
                        r, c = row, col
                        # count connected pieces
                        while (0 <= r < board.rows and 0 <= c < board.cols and
                               board.board[r][c] == 2):
                            connected += 1
                            r += dr
                            c += dc
                        score += connected ** 2  # square for emphasis on longer connections
        return score
```

```python
    def minimax(self, board, depth, is_maximizing, alpha, beta):
        if depth == 0 or board.is_terminal():
            # if game is over, will evaluate without requiring a valid move
            if board.is_terminal():
                winner = board.check_winner()
                if winner == 2:  # ai wins
                    return float('inf')
                elif winner == 1:  # player wins
                    return float('-inf')
                else:  # draw
                    return 0
            # for non-terminal states, evaluate normally
            valid_moves = board.get_valid_moves()
            if not valid_moves:
                return 0
            return self.evaluate_position(board, valid_moves[0])

        if is_maximizing:
            max_eval = float('-inf')
            for col in board.get_valid_moves():
                board.make_move(col)
                eval = self.minimax(board, depth - 1, False, alpha, beta)
                board.undo_move(col)
                max_eval = max(max_eval, eval)
                alpha = max(alpha, eval)
                if beta <= alpha:
                    break
            return max_eval
        else:
            min_eval = float('inf')
            for col in board.get_valid_moves():
                board.make_move(col)
                eval = self.minimax(board, depth - 1, True, alpha, beta)
                board.undo_move(col)
                min_eval = min(min_eval, eval)
                beta = min(beta, eval)
                if beta <= alpha:
                    break
            return min_eval
```

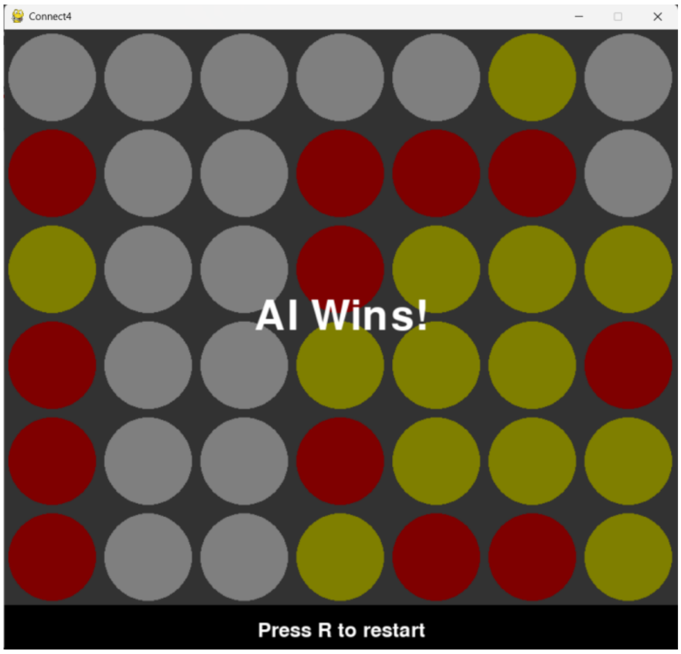
<u>Some Results:</u>

In-game screenshots:



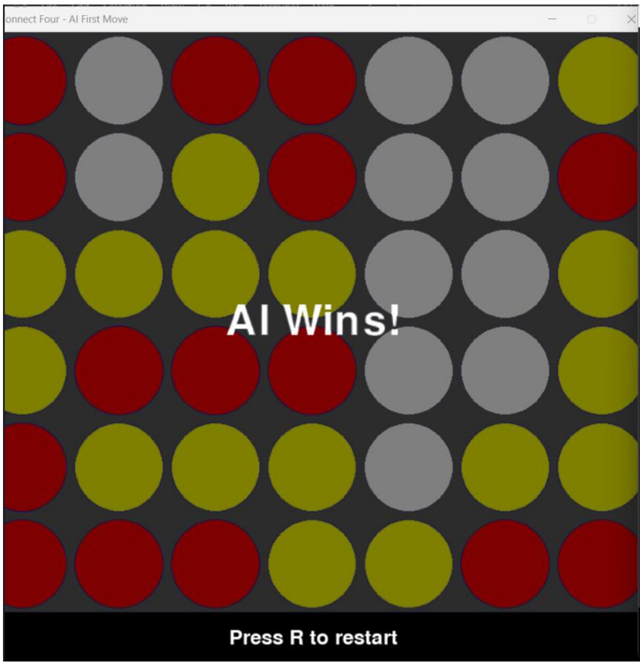


After some steps

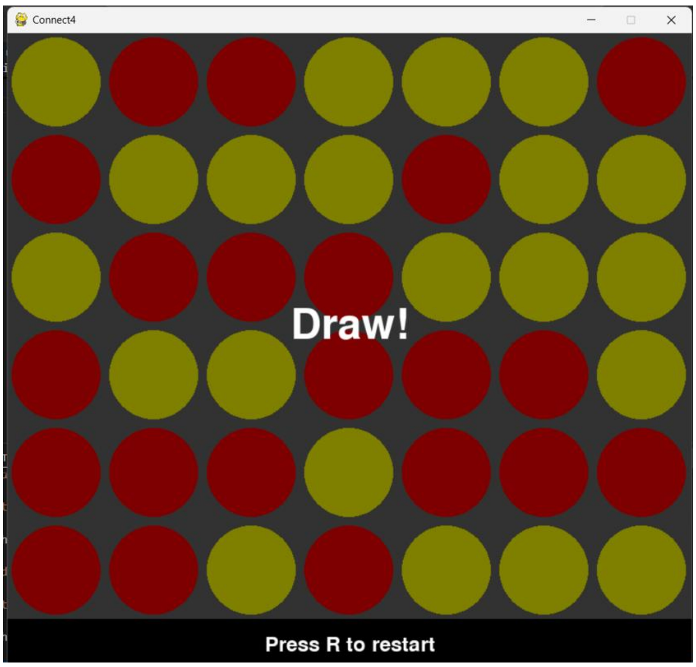


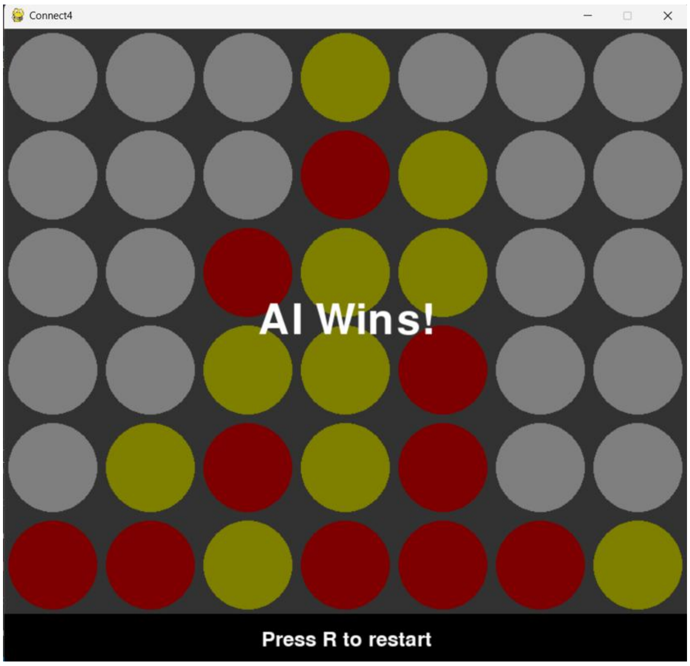
Some other final outcomes :

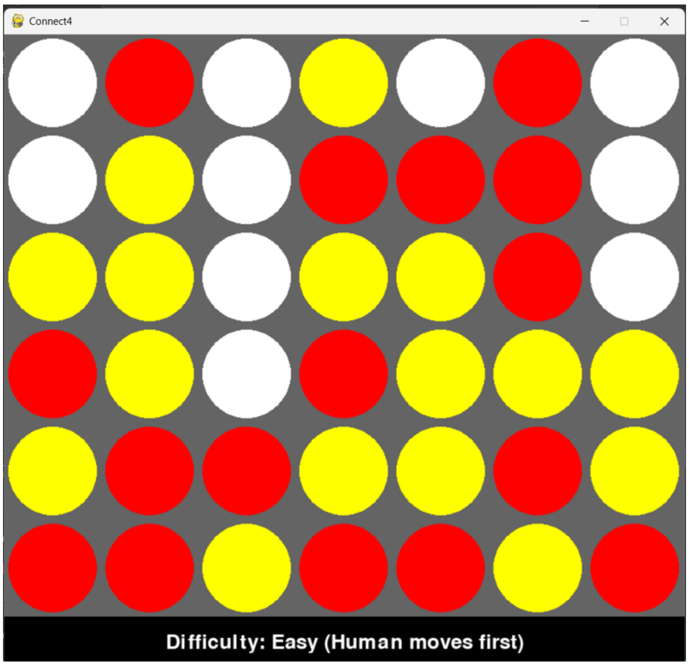
1. Playing two games simultaneously (AI move in first = Human move in second & vice-versa)

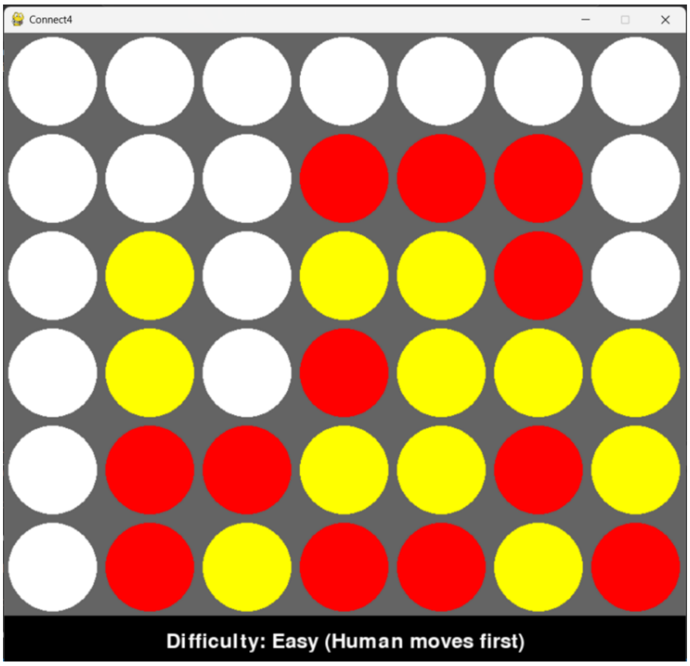


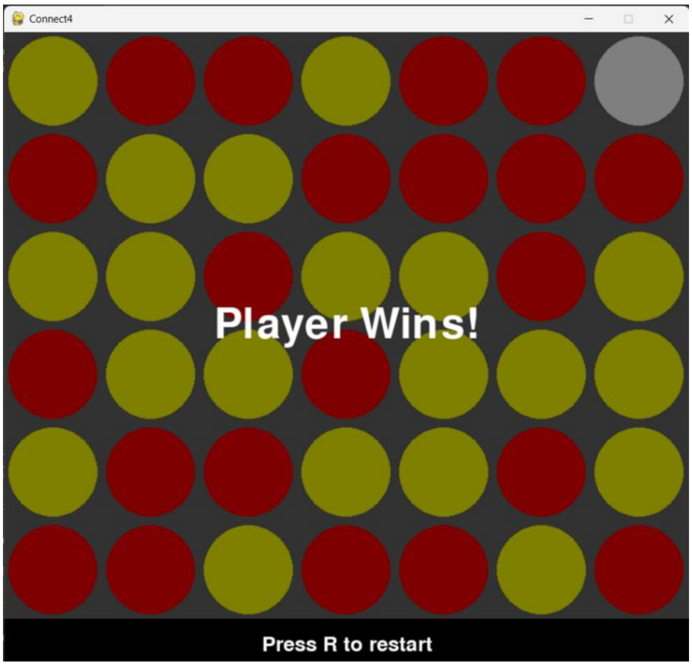
2. Human v/s AI (different kind of outcomes)

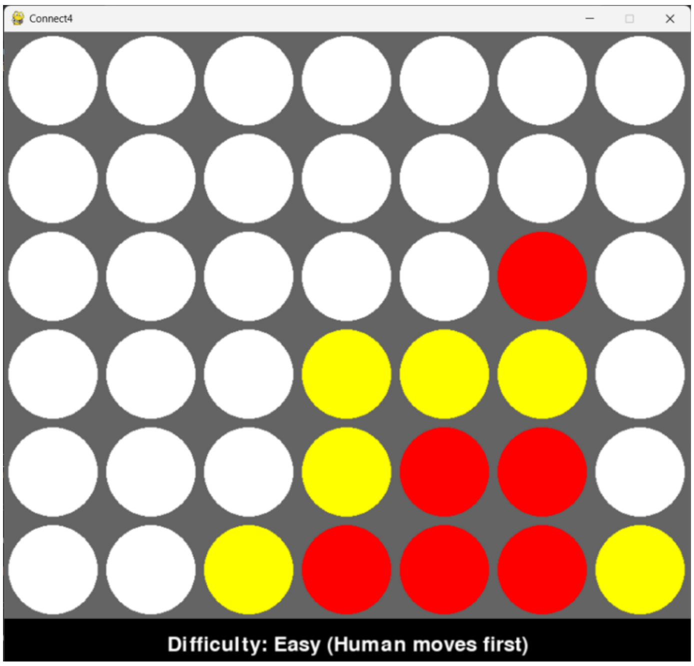


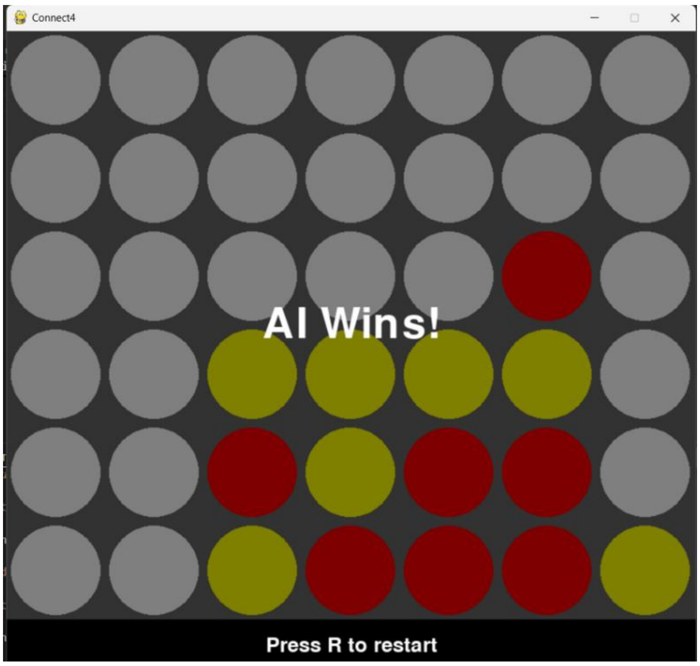

Connect4
AI Wins!
Press R to restart


Connect4
Difficulty: Easy (Human moves first)


Connect4
Difficulty: Easy (Human moves first)


Connect4
Player Wins!
Press R to restart


Connect4
Difficulty: Easy (Human moves first)


Connect4
AI Wins!
Press R to restart