Project report

Artificial Intelligence Course

First let's talk about the Implementation’s design pattern and then about the outline of each class.

The design pattern used in this code is the**Model-View-Controller (MVC)** pattern. Let's break down the code and see how it aligns with the MVC pattern.

1. Model (ConnectFourModel):
- The ConnectFourModel class represents the game's data logic.
- It maintains the state of the game board and the current player.
- It provides methods to interact with the board, such as making moves, checking for a winner, and getting valid moves.

2. View (ConnectFourView):
- The ConnectFourView class handles the visual game representation.
- It uses the Pygame to create a game board graphical user interface
- It draws the board & pieces and handles user input for making moves.

3. Controller (ConnectFourGame):
- This class acts as the controller that connects the model and view.
- It controls the flow of the game.
- It interacts with model & view, making moves & checking conditions.
- It uses the MinimaxAgent class to choose moves for the AI player.

The MVC pattern promotes separation of concerns, where the model represents the data and logic, the view represents the user interface, and the controller manages the interaction between the model and view.

Now let's provide a short description of each class:

▶ **ConnectFourModel:**
- Represents the game's data and logic.
- Maintains the game board, current player
- provides methods for interacting with the board.
- Allows making moves and getting valid moves
- Allows checking for a winner and checking if the game is over.

▶ **MinimaxAgent:**
- Implements the Minimax algorithm to make optimal moves for the AI
- Chooses the best move by evaluating state and assigning scores.
- Evaluates the board using heuristics to determine the best move.

▶ **ConnectFourView:**
- Handles the visual representation of the game with Pygame.
- Displays the game board, pieces, and the winner on the screen.
- Allows the user to make moves by clicking on the desired column.

▶ **ConnectFourGame:**
- Acts as the controller that connects the model and view.
- Controls the flow of the game, taking user input and updating model.
- Uses the MinimaxAgent to choose moves for the AI player.
- Checks for game conditions and displays the winner.

As you can see above we use Minimax Agent as our AI-agent in the first section of the project (the second section is Q-learning)

To determine the appropriate depth, k, for the minimax tree in the Connect Four game, we need to consider the trade-off between computational complexity and the agent's ability to make optimal moves. Increasing the depth of the tree allows the agent to explore more possible moves and make better decisions. However, a deeper tree also requires more computational resources and time to evaluate.

In choosing the value of k, we can consider the following logical simplifications to increase the efficiency of the tree:

1. Board Evaluation: Implementing efficient heuristics or evaluation functions to quickly assess the desirability of a given game state can reduce the need for deep tree searches. By accurately evaluating the current state of the board, we can prioritize promising moves and avoid exploring unpromising branches.

2. Alpha-Beta Pruning: Utilizing alpha-beta pruning technique allows us to reduce the number of nodes evaluated in the minimax tree. This technique eliminates branches that are guaranteed to be worse than previously evaluated branches, further reducing the computational complexity.

3. Transposition Tables: Implementing transposition tables can store previously computed results, allowing for faster lookups and avoiding redundant computations. This can significantly improve the efficiency of the tree search.

Considering these logical simplifications and the computational resources available, the choice of k depends on finding a balance between accuracy and efficiency. A higher value of k will lead to more accurate evaluations and potentially better moves but will also require more computational resources. Conversely, a lower value of k will yield faster calculations but may sacrifice optimality in decision-making.

The selection of k should be based on empirical testing and performance evaluation. It is recommended to start with a moderate depth and gradually increase it while measuring the agent's performance in terms of win rate or playing against other agents. This iterative process allows us to find the optimal value of k that balances computational efficiency and decision quality.

Pay attention we have trade of between the K and the execution time And also we have a relation between execution time and the Minimax enhancement. Due to this, if we can enhance our agent; then we can increase K. To increase the value of k, you can employ several strategies:

1. **Iterative Deepening:** Instead of searching the entire tree with a fixed depth, you can perform an iterative deepening search. Start with a small depth and gradually increase it in each iteration until a time limit is reached. This way, you can make the best possible move within the available time frame.

2. **Move Ordering:** By implementing efficient move ordering techniques, you can prioritize exploring more promising moves first. This can be based on heuristics or evaluation functions that estimate the desirability of moves. Ordering the moves can help the agent find good moves earlier in the search, potentially reducing the need to explore deeper levels.

3. **Transposition Table and Caching:** Utilize a transposition table to cache previously computed results. This can prevent redundant evaluations and improve the efficiency of the search. By storing the evaluation of a particular board position, you can reuse that information when encountering the same position later in the search.

4. **Parallelization:** If your hardware allows, you can explore parallelization techniques to search different branches of the tree simultaneously. This can significantly speed up the search process and allow for deeper exploration within the given time constraint.

5. **Pruning Techniques:** Implement advanced pruning techniques, such as aspiration search or null move pruning, to reduce the number of nodes evaluated during the search. These techniques aim to prune unproductive branches early on and focus on more promising paths.

It's important to note that increasing the value of k will inherently require more computational resources and may eventually reach a point of diminishing returns. Therefore, it's crucial to strike a balance between the desired depth and the available time/resources to achieve the best possible performance within the given constraints.

As we know the k value and the optimality of the minimax aren't only factors in the efficiency of the Agent. Heuristics and evaluation play an important role in it too. To develop an advanced heuristic for Connect 4, you need to consider several factors that can influence the game's outcome. Here are some important factors to consider and potential heuristics for each:

1. Piece Count: The number of pieces on the board can be a good indicator of the player's advantage. A simple heuristic is to assign a higher score to positions with more of the player's pieces and a lower score to positions with more opponent's pieces.

2. Piece Placement: The placement of a piece can be crucial in determining the potential for creating winning positions. You can assign higher scores to positions that have a higher chance of leading to a win, such as positions that complete a row, column, or diagonal.

3. Threats and Blocking: Identifying threats from the opponent and blocking them is essential. A heuristic can assign scores to positions that either create threats for the player or block the opponent's potential winning moves.

4. Center Control: The center column is strategically important in Connect 4 as it offers more opportunities to create winning positions. You can assign higher scores to positions that occupy the center column or are adjacent to it.

5. Edge Control: The edges of the board can be advantageous for creating multiple threats. A heuristic can assign higher scores to positions that control the edges, as they provide more possibilities for winning moves.

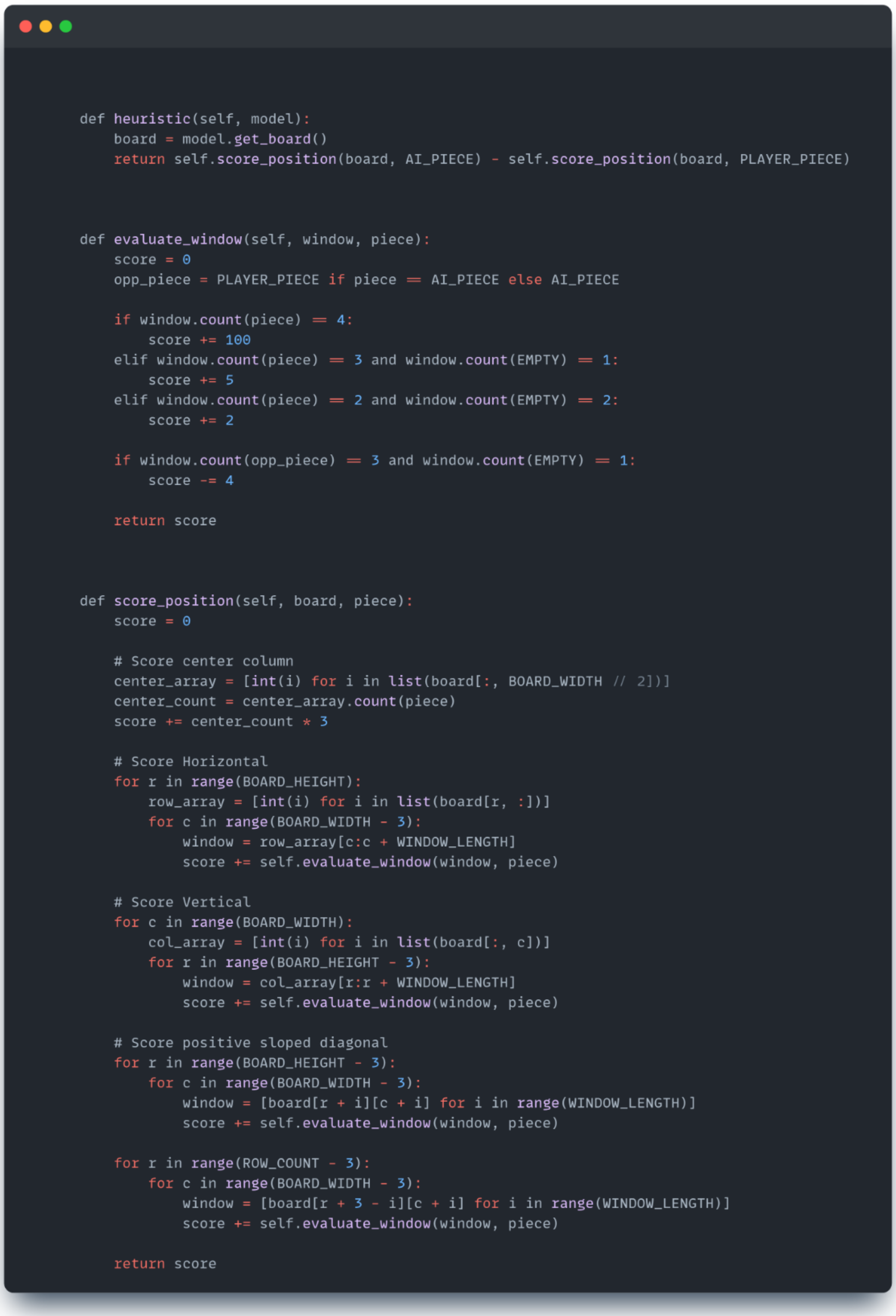
6. Mobility: Having more options for future moves can give a player an advantage. A heuristic can assign scores based on the number of legal moves available after making a particular move, favoring positions that offer more possibilities.

7. Chaining: Creating chains of connected pieces can increase the likelihood of winning. A heuristic can assign scores based on the length and potential for extending chains of connected pieces.

8. Defensive Measures: Sometimes, preventing the opponent from creating winning positions is more important than creating your own. A heuristic can assign scores to defensive moves that prioritize blocking the opponent's potential winning moves.

These are just a few factors and corresponding heuristics to consider when developing an advanced Connect 4 heuristic. You can combine and fine-tune these factors based on their relative importance to create a more sophisticated heuristic. Additionally, machine learning techniques like neural networks can be employed to automatically learn the optimal heuristic from a large dataset of expert gameplay.

In our code we use this heuristic :



```python
def heuristic(self, model):
    board = model.get_board()
    return self.score_position(board, AI_PIECE) - self.score_position(board, PLAYER_PIECE)


def evaluate_window(self, window, piece):
    score = 0
    opp_piece = PLAYER_PIECE if piece == AI_PIECE else AI_PIECE

    if window.count(piece) == 4:
        score += 100
    elif window.count(piece) == 3 and window.count(EMPTY) == 1:
        score += 5
    elif window.count(piece) == 2 and window.count(EMPTY) == 2:
        score += 2

    if window.count(opp_piece) == 3 and window.count(EMPTY) == 1:
        score -= 4

    return score


def score_position(self, board, piece):
    score = 0

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

    # Score Horizontal
    for r in range(BOARD_HEIGHT):
        row_array = [int(i) for i in list(board[r, :])]
        for c in range(BOARD_WIDTH - 3):
            window = row_array[c:c + WINDOW_LENGTH]
            score += self.evaluate_window(window, piece)

    # Score Vertical
    for c in range(BOARD_WIDTH):
        col_array = [int(i) for i in list(board[:, c])]
        for r in range(BOARD_HEIGHT - 3):
            window = col_array[r:r + WINDOW_LENGTH]
            score += self.evaluate_window(window, piece)

    # Score positive sloped diagonal
    for r in range(BOARD_HEIGHT - 3):
        for c in range(BOARD_WIDTH - 3):
            window = [board[r + i][c + i] for i in range(WINDOW_LENGTH)]
            score += self.evaluate_window(window, piece)

    for r in range(ROW_COUNT - 3):
        for c in range(BOARD_WIDTH - 3):
            window = [board[r + 3 - i][c + i] for i in range(WINDOW_LENGTH)]
            score += self.evaluate_window(window, piece)

    return score
```

- The `heuristic` method takes a `model` object as input and returns the difference between the scores obtained by the AI player (`AI_PIECE`) and the human player (`PLAYER_PIECE`). It calls the `score_position` method for both players and subtracts the score obtained by the human player from the score obtained by the AI player.

- The `evaluate_window` method takes a `window` list and a `piece` (player's game piece) as input and returns a score based on the contents of the window. The window represents a section of the game board, such as a row, column, or diagonal. It checks for different patterns in the window and assigns scores accordingly. For example, if the window contains 4 pieces of the same player (`piece`), the score is increased by 100. If the window contains 3 pieces of the same player and one empty space (`EMPTY`), the score is increased by 5. Similar scoring rules are applied for other combinations of player's pieces and empty spaces. Additionally, if the window contains 3 pieces of the opponent's player (`opp_piece`) and one empty space, the score is decreased by 4.
- The `score_position` method takes the game `board` and a `piece` as input and calculates the score for the given position on the board. It iterates over different sections of the board (columns, rows, and diagonals) and calls the `evaluate_window` method to calculate the score for each window. The scores are accumulated and returned as the final score for the given position.

Note that some parts of this report is written with gpt's