Laboratory 2

Variant 2

Group 12

By Andrea Amato and Dimitrios Georgouisis

Introduction

Task description:
The task was to develop a program that plays the Connect Four game against a user, employing the Minimax algorithm with alpha-beta pruning. Connect Four is a two-player connection game in which the players choose a color and then take turns dropping colored discs from the top into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the lowest available space within the column. The objective is to be the first to form a horizontal, vertical, or diagonal line of four of one's own discs.

Algorithm Description:
The Minimax algorithm is a decision rule used for minimizing the possible loss for a worst-case scenario. It is widely used in two-player games, such as tic-tac-toe, chess, and Connect Four, where players take turns. The algorithm simulates all possible moves in the game tree, looking ahead to determine a move that will lead to an outcome with the highest score for the AI, assuming optimal play by the opponent. The alpha-beta pruning enhancement is used to reduce the number of nodes evaluated in the game tree by the Minimax algorithm, effectively skipping the evaluation of moves that would not be chosen by the minimax algorithm.

Advantages and Disadvantages:
The Minimax algorithm with alpha-beta pruning has several advantages, including its ability to ensure that the AI can make the most optimal move considering the player will also play optimally. This makes the AI challenging and improves the gameplay experience. However, its disadvantages include high computational cost for games with large branching factors or depth, as it involves exploring many possible moves and counter-moves.

Implementation

Initialization: The game board is initialized as a 2D grid filled with spaces representing empty cells. Players are represented by "X" and "O". The first player to start is assigned the "X" mark.

```
self.board = [[EMPTY] * COLS for _ in range(ROWS)]
self.current_player = PLAYER_X
```

Fig.1 – Game initialization

Player choice: A message is displayed on the console that asks the user whether they want to start first or second. After the input is received, the players are set accordingly with the set_players function.

```
print("\nWelcome to Connect Four!\n")
while True:
    try:
        # our code: set the player
        print("Enter '1' to play first or '2' to play second")
        turn = int(input()) - 1
        if turn not in[0, 1]:
            print("\nInvalit number\n")
            continue
        game.set_players(turn)
        break
    except ValueError:
        print("\nInvalid input. Please enter a number\n")
```

Fig.2 – Player choice, inside the main function

```
def set_players(self, turn):
    """
    Set the current player

    Parameters:
    - turn: turn of the player (0 means player goes first,
      1 means computer goes first)

    """

    self.human_player = PLAYER_X if turn == 0 else PLAYER_O

    self.computer_player = PLAYER_O if turn == 0 else PLAYER_X
```

Fig.3 – Implementation of set_player function

Gameplay loop: After players are set, the program enters in the gameplay loop, managing alternation of turns between human player and the AI, processing inputs, updating the game state, and determining the game's outcome. We will show the related code and then comment upon it.

```
while True:
  game.print_board()
  print()

  if game.current_player == game.human_player:
    print("Human turn")
    while True:
      try:
        col = int(input("Enter column (0-6): "))
        if col < 0 or col >= COLS:
```

```python
                print("\nColumn must be between 0 and 6\n")
                continue
            elif not game.is_valid_move(game.board, col):
                print("\nColumn is full. Try a different one\n")
                continue
            row = game.get_open_row(game.board, col)
            game.drop_piece(game.board, row, col, game.human_player)
            break
        except ValueError:
            print("\nInvalid input. Please enter a number\n")
            continue

    if game.is_winner(game.board, game.human_player):
        game.print_board()
        print("Human wins!\n")
        break
    game.current_player = game.computer_player

else:
    print("Computer turn")
    _, col = game.minimax(game.board, 5, True, float('-inf'),
                          float('inf'))
    if game.is_valid_move(game.board, col):
        row = game.get_open_row(game.board, col)
        game.drop_piece(game.board, row, col, game.computer_player)
        if game.is_winner(game.board, game.computer_player):
            game.print_board()
            print()
            print("Computer wins!\n")
            break
        game.current_player = game.human_player

# check if the game is over
```

```
    if game.is_terminal(game.board):
        game.print_board()
        print("It's a tie!")
        break
```

Fig.4 – Implementation of the gameplay loop inside the main function

The loop always starts by printing the board and then checks if it is the human player's turn, in which case the game prompts the player to enter the column number where they want to drop their piece. The user input is type checked to confirm it can be casted to an integer and validated to ensure it is a valid move (the column exists and is not full). The player's piece is then placed in the lowest available row in the chosen column with the use of two helper functions: get_open_row and drop_piece.

After each time a piece is dropped, the game checks if the move led to a win (four in a row horizontally, vertically, or diagonally) or a tie (the board is full) using the functions is_winner *(Fig.8)* and is_terminal *(Fig.9)*. If there's a win, it announces the winner and ends the game. If it is a tie, it declares so and concludes the game.

On the AI's turn, instead, the game employs the Minimax algorithm with alpha-beta pruning to calculate the best move, considering a search to a depth of five moves ahead. Analogous code runs on this workflow, with move validation, and checks for win and end of game.

After each turn, control switches between the human player and the AI until the game ends. The game loop ensures continuous play, alternating turns and updating the board after each move, providing a dynamic and interactive gameplay experience.

```
def is_valid_move(self, board, col):
    """
    Check if the move is valid
    Parameters:
    - board: 2d matrix representing the state, each cell contains
      either ' ' (empty cell), 'X' (player1), or 'O' (player2)
    - col: column where the player wants to put the piece

    Returns:
    - True if the move is valid, False otherwise

    """

    return board[0][col] == EMPTY
```

Fig.5 – Implementation of is_valid_move function

```
def get_open_row(self, board, col):
    """
    Get the open row

    Parameters:
    - board: 2d matrix representing the state, each cell contains
      either ' ' (empty cell), 'X' (player1), or 'O' (player2)
    - col: column where the player wants to put the piece

    Returns:
    - row where the piece will be placed

    """

    # we prefer the deepest row, so we start from the bottom
    for r in range(ROWS - 1, -1, -1):
        if board[r][col] == EMPTY:
            return r
```

Fig.6 – Implementation of get_open_row function

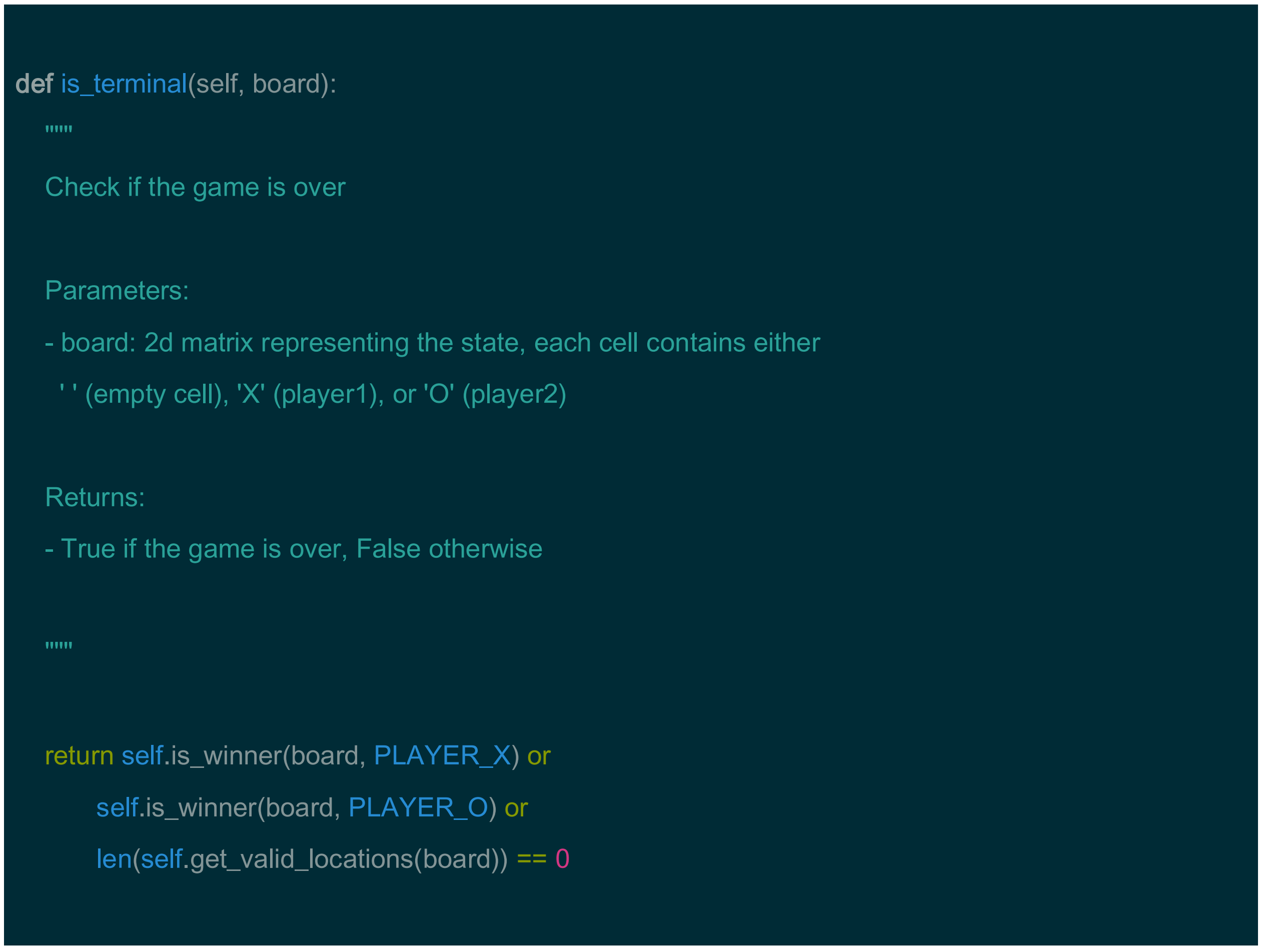
```
def drop_piece(self, board, row, col, piece):
    """
    Drop the piece to the board

    Parameters:
    - board: 2d matrix representing the state, each cell contains either
      ' ' (empty cell), 'X' (player1), or 'O' (player2)
    - row: row where the piece will be placed
    - col: column where the player wants to put the piece
    - piece: PLAYER_X or PLAYER_O depending on which player's position we
      evaluate

    """

    board[row][col] = piece
```

Fig.7 – Implementation of drop_piece function

```
def is_winner(self, board, piece):
    """
    Check if the player has won

    Parameters:
    - board: 2d matrix representing the state, each cell contains either
      ' ' (empty cell), 'X' (player1), or 'O' (player2)
    - piece: PLAYER_X or PLAYER_O depending on which player's position we
      evaluate
```

```
Returns:
- True if the player has won, False otherwise

"""

# Check horizontal locations for win
for c in range(COLS - 3):
    for r in range(ROWS):
        if board[r][c:c + 4].count(piece) == 4:
            return True

# Check vertical locations for win
for c in range(COLS):
    for r in range(ROWS - 3):
        if [board[r + i][c] for i in range(4)].count(piece) == 4:
            return True

# Check positively sloped diagonals
for r in range(ROWS - 3):
    for c in range(COLS - 3):
        if [board[r + i][c + i] for i in range(4)].count(piece) == 4:
            return True

# Check negatively sloped diagonals
for r in range(ROWS - 3):
    for c in range(COLS - 3):
        if [board[r + 3 - i][c + i] for i in range(4)].count(piece)== 4:
            return True

return False
```

Fig.8 – Implementation of is_winner function



```
def is_terminal(self, board):
    """
    Check if the game is over

    Parameters:
    - board: 2d matrix representing the state, each cell contains either
      ' ' (empty cell), 'X' (player1), or 'O' (player2)

    Returns:
    - True if the game is over, False otherwise

    """

    return self.is_winner(board, PLAYER_X) or
           self.is_winner(board, PLAYER_O) or
           len(self.get_valid_locations(board)) == 0
```

Fig.9 – Implementation of is_terminal function

Minimax algorithm and position evaluation: The minimax function *(Fig.10)* is the core of the decision-making process of the AI. We start by first setting the players and look for all the columns where dropping a piece is valid (get_valid_locations – *Fig.11*). The algorithm then checks whether it has reached a predetermined depth (i.e. depth is equal to zero), therefore limiting how far ahead the AI looks, or if it reaches a terminal state (a win, a loss, or a tie). If the latter occurs, the function returns 0 if it is a tie, or a very big positive, or negative score respectively for a win or a loss. Otherwise, if depth is zero, it calls evaluate_position function *(Fig.12)*, returning a score representing the state of the game from the perspective of the current player. This score is then used to inform the AI's decision-making for subsequent moves.

If none of the previous conditions are verified, the algorithm proceeds to evaluate potential moves. This is done differently based on whether it is the AI's turn to maximize the score or the human player's turn to minimize it.

For the maximizing player, which represents the AI's turn, the function initializes the best score, i.e. value, to negative infinity and iterates through all valid moves (columns where a piece can be dropped). For each move, it simulates dropping a piece in that column by creating a copy of the board and evaluates the move's potential using a recursive call to minimax, this time with decreased depth and reversed roles (minimizing player's turn). It updates the best score if the evaluated score for a move is higher than the current best. This process also involves updating the alpha value for alpha-beta pruning, where if the current value is greater than or equal to beta (the opponent's best avoided score), the loop breaks early, as further exploration of moves is unnecessary.

When it is the minimizing player's turn, indicating the opponent, the process is similar but aims to minimize the score instead. The best score, value, is initialized to positive infinity. As the AI simulates different moves by the opponent, it seeks to find the move that would lead to the lowest score, representing the worst outcome for the AI. Here, beta represents the minimum score that the maximizing player is assured of and is updated if a lower is found. If beta is less than or equal to alpha at any point, it indicates that the maximizing player has a better alternative elsewhere, prompting an early break from the loop.

```
def minimax(self, board, depth, maximizing_player, alpha, beta):
    """
    Minimax with alpha-beta pruning algorithm

    Parameters:
    - board: 2d matrix representing the state, each cell contains either
      ' ' (empty cell), 'X' (player1), or 'O' (player2)
```

```
- depth: depth
- maximizing_player: boolean which is equal to True when the player
  tries to maximize the score
- alpha: alpha variable for pruning
- beta: beta variable for pruning

Returns:
- Best value
- Best move found

"""

# determine the current player and the opponent
player = self.current_player
opponent = PLAYER_X if player == PLAYER_O else PLAYER_O

# get the valid locations, used later to determine the best move
valid_locations = self.get_valid_locations(board)
# check if the game is over
is_terminal = self.is_terminal(board)

if depth == 0 or is_terminal:
    if is_terminal:
        if self.is_winner(board, player):
            return (1_000_000_000, None) # a really big score
        elif self.is_winner(board, opponent):
            return (-1_000_000_000, None) # a similarly big negative
                                    score
        else: # Game is over, no more valid moves
            return (0, None)

    else: # Depth is zero
        return (self.evaluate_position(board, player), None)
```

```
if maximizing_player: # Maximizing player
    value = float('-inf')
    column = random.choice(valid_locations) # start with a random column
    for col in valid_locations:
        row = self.get_open_row(board, col) # get the row where the
                                              piece will fall
        board_copy = copy.deepcopy(board) # create a copy of the board
        self.drop_piece(board_copy, row, col, player) # drop the piece
                                                       to the board
        new_score = self.minimax(board_copy, depth - 1, False, alpha,
                          beta)[0]
        if new_score > value:
            value = new_score
            column = col
        alpha = max(alpha, value) # keep increasing alpha
        if alpha >= beta:
            break
    return value, column

else: # Minimizing player
    value = float('inf')
    column = random.choice(valid_locations)
    for col in valid_locations:
        row = self.get_open_row(board, col)
        board_copy = copy.deepcopy(board)
        self.drop_piece(board_copy, row, col, opponent)
        new_score = self.minimax(board_copy, depth - 1, True, alpha,
                          beta)[0]
        if new_score < value:
            value = new_score
            column = col
        beta = min(beta, value) # keep decreasing beta
```

```
            if alpha >= beta:
                break
        return value, column
```

Fig.10 – Implementation of minimax function

```
def get_valid_locations(self, board):
    """
    Get valid locations

    Parameters:
    - board: 2d matrix representing the state, each cell contains either
      ' ' (empty cell), 'X' (player1), or 'O' (player2)

    Returns:
    - list of valid locations

    """

    valid_locations = []
    for col in range(COLS):
        if self.is_valid_move(board, col):
            valid_locations.append(col)
    return valid_locations
```

Fig.11 – Implementation of get_valid_locations function

Through this iterative process of simulating moves and recursively evaluating the game tree with alpha-beta pruning, the minimax function effectively navigates the vast array of possible futures to select the move that best secures a favorable outcome for the AI, balancing between aggressive plays to win and defensive moves to block the opponent’s strategies.

We previously mentioned the evaluate_position function. We now present its implementation code and then comment upon it.

```
def evaluate_position(self, board, piece):
    """
    Evaluation of position
    Parameters:
    - board: 2d matrix representing evaluated state of the board
    - piece: PLAYER_X or PLAYER_O depending on which player's position we evaluate

    Returns:
    - score of the position

    """

    score = 0

    # Score center column
    # We give extra weight to the center column, as it is the column that gives most options
    # for creating winning patterns. Thus we also include a multiplier to the count of our pieces
    # in it.
    center_array = [board[r][COLS // 2] for r in range(ROWS)]
    center_count = center_array.count(piece)
    score += center_count * 3

    # Score Horizontal
    for r in range(ROWS):
        row_array = [board[r][c] for c in range(COLS)]
        for c in range(COLS - 3):
            window = row_array[c:c + 4]
            score += self.evaluate_window(window, piece)
```

```
# Score Vertical
for c in range(COLS):
    col_array = [board[r][c] for r in range(ROWS)]
    for r in range(ROWS - 3):
        window = col_array[r:r + 4]
        score += self.evaluate_window(window, piece)

# Score positive sloped diagonal
for r in range(ROWS - 3):
    for c in range(COLS - 3):
        window = [board[r + i][c + i] for i in range(4)]
        score += self.evaluate_window(window, piece)

# Score negative sloped diagonal
for r in range(ROWS - 3):
    for c in range(COLS - 3):
        window = [board[r + 3 - i][c + i] for i in range(4)]
        score += self.evaluate_window(window, piece)



return score
```

Fig.12 – Implementation of evaluate_position function

The core objective of evaluate_position is to provide a heuristic score representing the desirability of the board's current state from the AI's perspective. It systematically examines the board to identify patterns that are beneficial or detrimental to the AI's chance of winning, applying scores based on the configuration of pieces in each window.
We assigned additional points for each piece the AI has in the central column, since based on the layout and rules of Connect Four, control over the center

offers strategic advantages, allowing for more opportunities to connect four pieces vertically, horizontally and diagonally.

This function breaks down the board into "windows" of four adjacent cells, which is the exact configuration needed to win the game. It then calls evaluate_window *(Fig. 13)* on them, in order to evaluate each window's composition—counting the AI's pieces, the opponent's pieces, and empty spaces—and assigns scores based on the potential of these windows to contribute to a win. For instance, a window containing three AI pieces and one empty space is highly valuable, whereas a configuration that has three opponent's pieces is given negative points as we want to avoid that.
The evaluate_window function evaluates horizontal, vertical, and both positive and negative diagonal lines across the board, ensuring a comprehensive assessment of the game state, which enables the AI to properly detect winning opportunities and threats that need to be blocked.

```
def evaluate_window(self, window, piece):
    """
    Evaluation of given window. Helper function to evaluate the separate
    parts of the board called windows.
    A window is a list of 4 elements, which can be a row, column, or
    diagonal of the board.

    Parameters:
    - window: list containing values of evaluated window
    - piece: PLAYER_X or PLAYER_O depending on which player's position we
      evaluate

    Returns:
    - score of the window


    """
```

```
# window is the first 4 elements of the given list
window = window[:4]

player = piece
opponent = PLAYER_X if piece == PLAYER_O else PLAYER_O

score = 0

# the numeric values given bellow are arbitrary and can be changed
# We set them as seen in order to account for the different 'weights' of
  the different situations
# We prioritize winning, then positions that are close to winning, but
  we give them a much lower score.
# Finally, a position that is close to winning for the opponent is given
  a negative score, as we want to avoid that.
if window.count(player) == 4:
    score += 100

elif window.count(player) == 3 and window.count(EMPTY) == 1:
    score += 8

elif window.count(player) == 2 and window.count(EMPTY) == 2:
    score += 4

if window.count(opponent) == 3 and window.count(EMPTY) == 1:
    score -= 6

return score
```

Fig.13 – Implementation of evaluate_window function

Discussion

The evaluation function is a critical aspect of the AI's decision-making process. Its purpose is to assess the current state of the board and assign a numerical value that represents the desirability of that state for whichever player is currently making a move. We designed the evaluation function to systematically analyze the board for both advantageous and adverse patterns that affect the given players's chance of victory.

Our function scrutinizes each possible "window" of four cells—vertical, horizontal, and diagonal— and assigns scores based on the alignment of pieces. Our scoring logic is nuanced: a window containing three AI pieces and one empty cell is valued highly, as it positions the AI one move away from winning. Conversely, windows that present the opponent with similar opportunities are scored negatively, encouraging the AI to play defensively when necessary.

Moreover, special consideration is given to the central column of the board. Control of this column is weighted more heavily in our scoring system due to the multiple opportunities it provides for creating connect fours. Our AI is thus inclined to occupy the central spaces, mirroring strategic play often employed by experienced human players.

Strengths: As previously discussed, our function is designed in such a way that enables the AI to pursue active winning strategies, but also to detect threats and block the opponent's progress. Furthermore, the evaluation can also be adjusted for different strategies by simply changing the score weights for various patterns.

Weaknesses: The evaluation of every possible window on the board can be computationally intensive, affecting the AI's response time and efficiency. Additionally, the function evaluates the board state without considering the depth of the game tree, potentially overlooking deeper strategic opportunities or threats.

Conclusion

We have enhanced our understanding of recursive algorithms and learned the importance of efficient evaluation heuristics, as well as the value of using optimization techniques like alpha-beta pruning in search algorithms. The balancing act between depth of search and computation efficiency was a constant consideration, illustrating the practical constraints faced in AI development for applications requiring rapid response times.

Designing an evaluation function that was nuanced yet not overly complex was one of the major challenges we encountered.

One area for potential improvement could be the implementation of a more complex and dynamic evaluation strategy that adapts to the changing context of the game. Additionally, exploring iterative deepening or other algorithms could yield a more strategic AI capable of deeper foresight without a prohibitive computational cost.