NAME : Vedansh Agarwal

ROLL NO. : 202401100300276(56)

BRANCH : CSE-AI(D)

PROBLEM STATEMENT : Noughts and Crosses with Alpha-Beta Pruning

Introduction on Noughts and Crosses with Alpha-Beta Pruning

The project aims to develop an AI that plays Noughts and Crosses in an optimal way with the Alpha-Beta Pruning algorithm. Noughts and Crosses is a two-player turn-based game on a 3x3 grid. Player One is the one who selects an X for a mark while Player Two selects an O for a mark. Each player gets to alternate turns selecting an empty cell in the 3x3 grid and filling it with their specific mark. A win is gained by a player who marks the same squares or cells that stand on either three consecutive rows, columns, or diagonals. If all squares or cells have been filled without any player winning, the game results in a draw. If the AI is to become smarter, this project incorporates Alpha-Beta Pruning, which is an optimization of the Minimax algorithm. Alpha-Beta Pruning essentially prunes those branches from the decision tree where it is impossible for their outcomes to have any effect on the final outcome, prevents repetitive computations methodologically. In this way, it makes decisions faster and better without compromising optimal performance. Thus, through a mix of good game play strategies, and good decisions, the project provides an overview of how the algorithm Alpha-Beta Pruning can be used to tackle real life problems within gaming.

Methodology

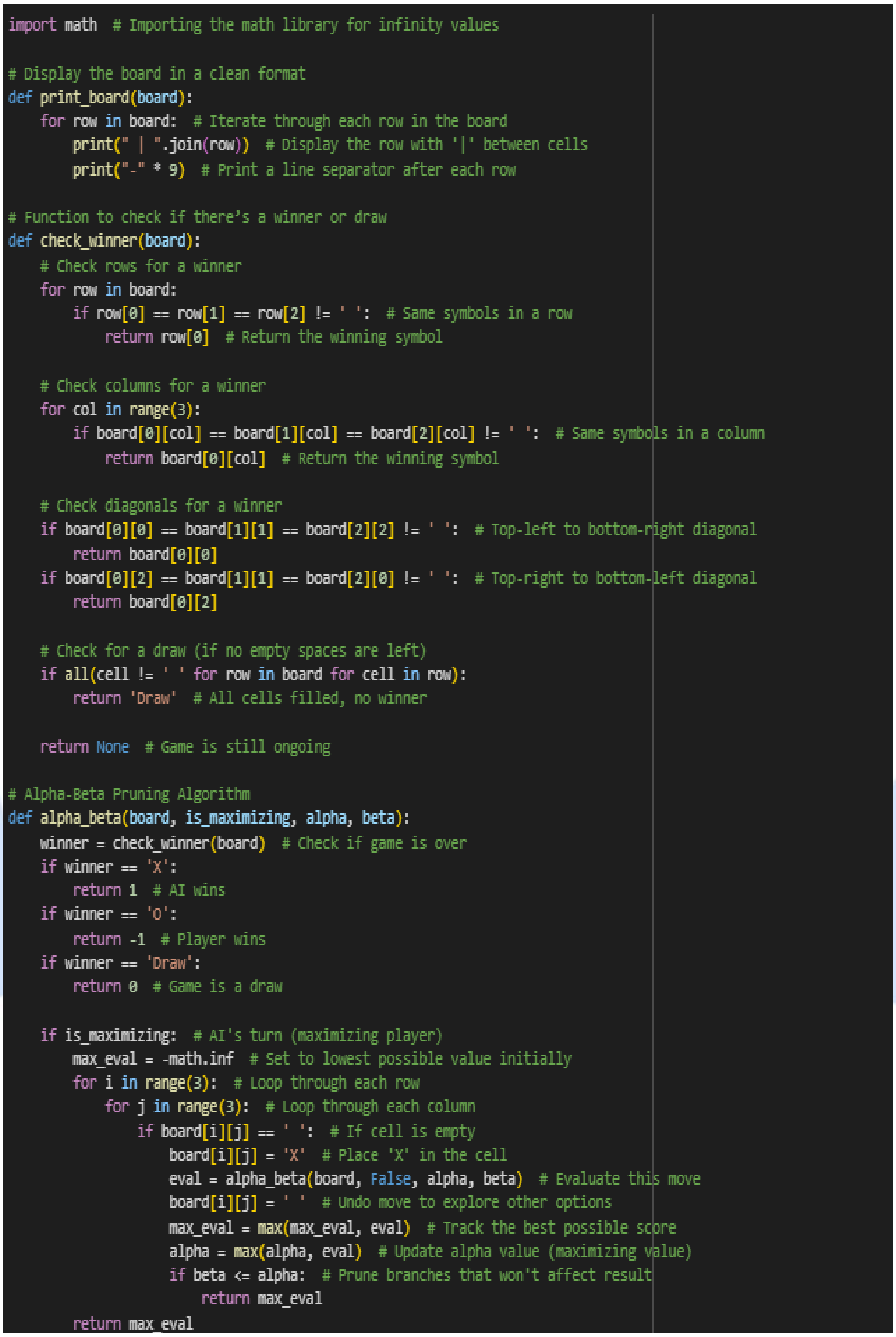
Alpha-Beta Pruning works by the AI making optimal decisions with more efficiency during the game. An advanced type of the Minimax algorithm was used, as it pruned unnecessary moves and therefore reduced calculation times without loss in quality.

The Decision Tree: Each possible move creates a branch in the decision tree, where all potential outcomes are taken into consideration. Alpha and Beta: Alpha does a promise of how much the maximizing player (our AI) might ensure for himself while Beta does that for the minimizing player (the human). Making use of these values allows it to find out more which branches are to be pruned and which ones will have to be analysed further.

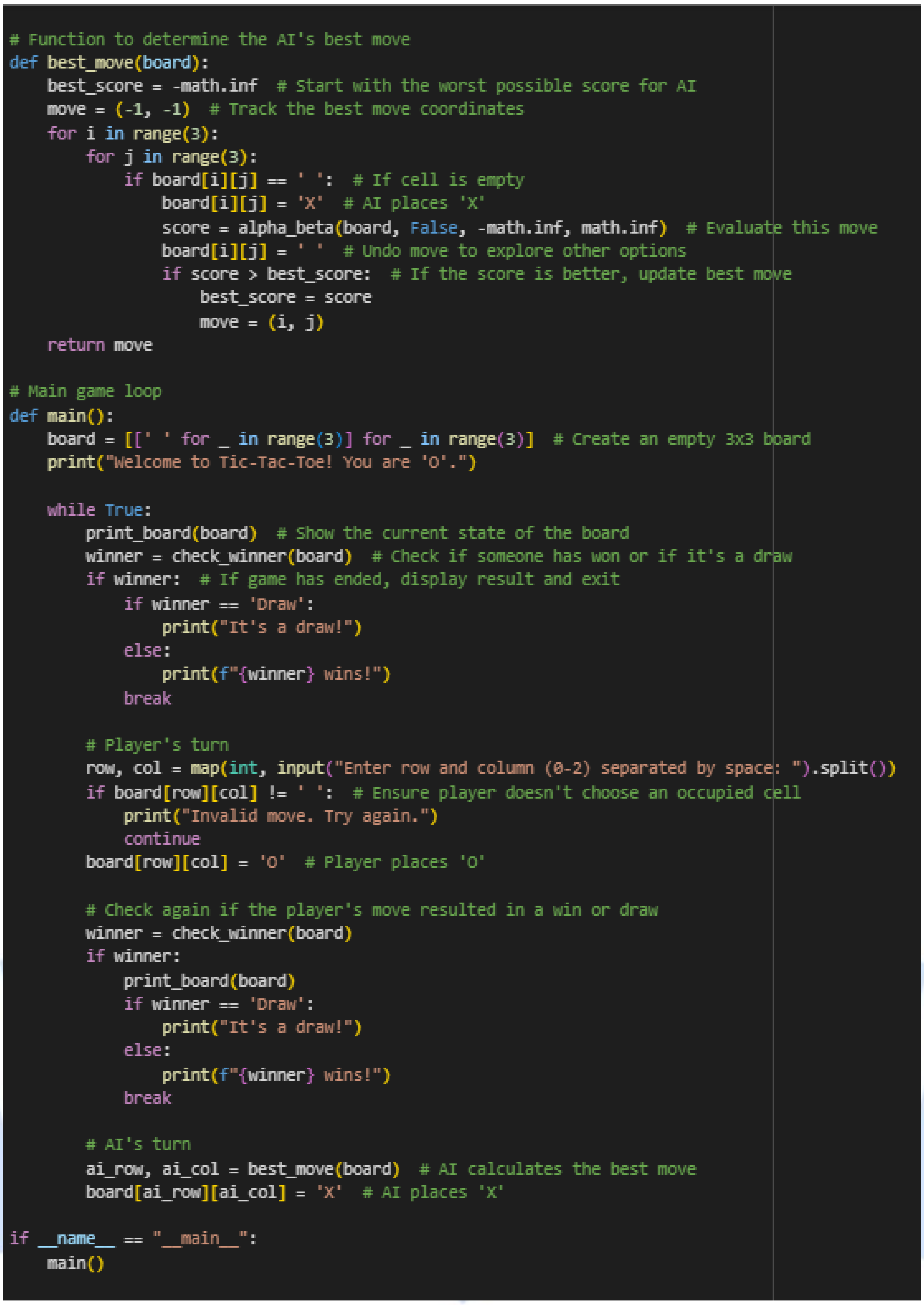
The pruning process: As the AI keeps scanning for better moves, the alpha and beta values get constantly updated. If there occurs a move that promises a less favourable outcome than what has been noted so far as the best option, that branch will be pruned to save computation time.

Scoring: +1-a win for AI -1-a win for human 0-draw. By minimizing irrelevant branches and unnecessary computations, it now becomes possible for the AI to find the best possible move through pruning, thus assuring the game will be faster and smarter now.

CODE

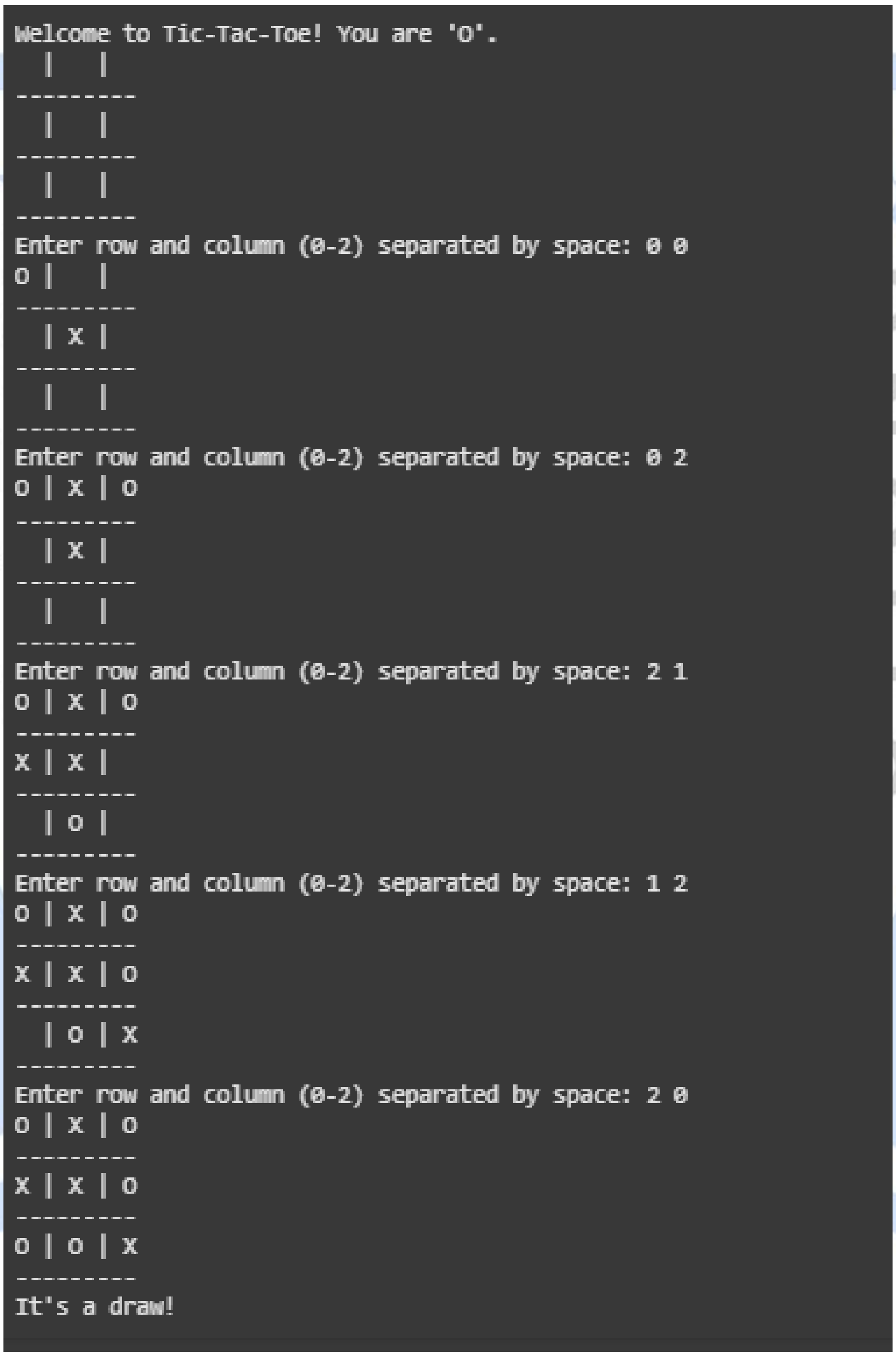


```python
import math  # Importing the math library for infinity values

# Display the board in a clean format
def print_board(board):
    for row in board:  # Iterate through each row in the board
        print(" | ".join(row))  # Display the row with '|' between cells
        print("-" * 9)  # Print a line separator after each row

# Function to check if there's a winner or draw
def check_winner(board):
    # Check rows for a winner
    for row in board:
        if row[0] == row[1] == row[2] != ' ':  # Same symbols in a row
            return row[0]  # Return the winning symbol

    # Check columns for a winner
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != ' ':  # Same symbols in a column
            return board[0][col]  # Return the winning symbol

    # Check diagonals for a winner
    if board[0][0] == board[1][1] == board[2][2] != ' ':  # Top-left to bottom-right diagonal
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != ' ':  # Top-right to bottom-left diagonal
        return board[0][2]

    # Check for a draw (if no empty spaces are left)
    if all(cell != ' ' for row in board for cell in row):
        return 'Draw'  # All cells filled, no winner

    return None  # Game is still ongoing

# Alpha-Beta Pruning Algorithm
def alpha_beta(board, is_maximizing, alpha, beta):
    winner = check_winner(board)  # Check if game is over
    if winner == 'X':
        return 1  # AI wins
    if winner == 'O':
        return -1  # Player wins
    if winner == 'Draw':
        return 0  # Game is a draw

    if is_maximizing:  # AI's turn (maximizing player)
        max_eval = -math.inf  # Set to lowest possible value initially
        for i in range(3):  # Loop through each row
            for j in range(3):  # Loop through each column
                if board[i][j] == ' ':  # If cell is empty
                    board[i][j] = 'X'  # Place 'X' in the cell
                    eval = alpha_beta(board, False, alpha, beta)  # Evaluate this move
                    board[i][j] = ' '  # Undo move to explore other options
                    max_eval = max(max_eval, eval)  # Track the best possible score
                    alpha = max(alpha, eval)  # Update alpha value (maximizing value)
                    if beta <= alpha:  # Prune branches that won't affect result
                        return max_eval
        return max_eval
```



```python
# Function to determine the AI's best move
def best_move(board):
    best_score = -math.inf  # Start with the worst possible score for AI
    move = (-1, -1)  # Track the best move coordinates
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':  # If cell is empty
                board[i][j] = 'X'  # AI places 'X'
                score = alpha_beta(board, False, -math.inf, math.inf)  # Evaluate this move
                board[i][j] = ' '  # Undo move to explore other options
                if score > best_score:  # If the score is better, update best move
                    best_score = score
                    move = (i, j)
    return move

# Main game loop
def main():
    board = [[' ' for _ in range(3)] for _ in range(3)]  # Create an empty 3x3 board
    print("Welcome to Tic-Tac-Toe! You are 'O'.")

    while True:
        print_board(board)  # Show the current state of the board
        winner = check_winner(board)  # Check if someone has won or if it's a draw
        if winner:  # If game has ended, display result and exit
            if winner == 'Draw':
                print("It's a draw!")
            else:
                print(f"{winner} wins!")
            break

        # Player's turn
        row, col = map(int, input("Enter row and column (0-2) separated by space: ").split())
        if board[row][col] != ' ':  # Ensure player doesn't choose an occupied cell
            print("Invalid move. Try again.")
            continue
        board[row][col] = 'O'  # Player places 'O'

        # Check again if the player's move resulted in a win or draw
        winner = check_winner(board)
        if winner:
            print_board(board)
            if winner == 'Draw':
                print("It's a draw!")
            else:
                print(f"{winner} wins!")
            break

        # AI's turn
        ai_row, ai_col = best_move(board)  # AI calculates the best move
        board[ai_row][ai_col] = 'X'  # AI places 'X'

if __name__ == "__main__":
    main()
```

<u>OUTPUT</u>



```
Welcome to Tic-Tac-Toe! You are 'O'.
  |   |
---------
  |   |
---------
  |   |
---------
Enter row and column (0-2) separated by space: 0 0
O |   |
---------
  | X |
---------
  |   |
---------
Enter row and column (0-2) separated by space: 0 2
O | X | O
---------
  | X |
---------
  |   |
---------
Enter row and column (0-2) separated by space: 2 1
O | X | O
---------
X | X |
---------
  | O |
---------
Enter row and column (0-2) separated by space: 1 2
O | X | O
---------
X | X | O
---------
  | O | X
---------
Enter row and column (0-2) separated by space: 2 0
O | X | O
---------
X | X | O
---------
O | O | X
---------
It's a draw!
```

<u>Credits</u>: Chatgpt