ARTIFICIAL INTELLIGENCE AND EXPERT SYSTEMS

CT-361

ASSIGNMENT 2

Name: Filza Tanveer

Roll no: CT-22057

Department: CSIT

Section: B

Filza Tanveer (CT-22057)

(IMPLEMENTING TIC TAC TOE GAME)

Requirements:

- Implement the core logic of the Tic-Tac-Toe game.
- Implement the Minimax algorithm to create an AI player that plays optimally.
- Implement the Alpha-Beta Pruning optimization to improve the efficiency of the Minimax algorithm.
- Compare the performance of the standard Minimax and the Alpha-Beta Pruning optimized Minimax.
- Submit the GitHub repo containing the code file and demo video.

(MINIMAX ALGORITHM EXPLAINED)

1. In game theory and artificial intelligence, the minimax algorithm is a decision-making method that determines a player's optimum course of action while the other player is also playing optimally. Zero-sum games, two-player games, and other games usually use the method.
2. By simulating every move that both players could make, the algorithm begins with the present game state and creates every conceivable following game state. It then creates all potential future states for each of those states, and so on, until it reaches a maximum search depth or a terminal state/leaf node (win, loss, draw, etc.).
3. The score of a min node, which represents the opponent's move, is the least of the scores of its children, while the score of a max node, which represents the current player's move, is the highest of the scores of its children.
4. Time Complexity: The minimax algorithm explores the entire game tree, evaluating all possible moves and their consequences. This leads to a time complexity of O(b^m). (b=branching factor and m= max depth).

(ALPHA BETA ALGORITHM EXPLAINED)

1. Alpha Beta Pruning is a Minimax algorithm optimization method. By using its parameters Alpha(α) and Beta(β) to prune unnecessary branches of a game tree, this technique overcomes the Minimax algorithm's limit of exponential time and space complexity.
2. Alpha(α) is the optimal option (highest value) and Beta(β) is the optimal option (lowest value).
3. Time Complexity: In the best case, it effectively reduces the branching factor to the square root of 'b', resulting in a time complexity of O(b^(m/2)).

Filza Tanveer (CT-22057)

(TIC TAC TOE GAME CODE USING MINIMAX ALGORITHM)

CODE EXPLAINATION:

In this game AI is ('X') and a human player is ('O') compete using the Minimax algorithm.

Main Features of code:

TicTacToe Class consists of following elements:

1. __init__: This function will create a blank board and keeps track of the nodes visited and the winner.
2. print_board: This will show the three-by-three board.
3. The list of accessible movements (empty squares) is returned by the available_moves function.
4. empty_squares: Determines whether any more blank squares exist.
5. make_move: This will allow a player to make a move at a specified location.
6. winner: This function will verify whether a player has won following a move.

Minimax Function:

1. It attempts every move both players may make in a recursive manner.
2. Moreover it reduces the opponent's (O) score while increasing the AI's (X) score.
3. Further is chooses the course of action that will yield the best results.
4. It keeps track of the number of nodes (moves) it made when making decisions.

Play Function:

1. This function initiates the game.
2. AI ('X') uses Minimax to play first.
3. A human ('O') manually enters their move.
4. Game continues until and unless someone wins o game is draw.

Performance:

Slower as compared to optimized tic tac toe game using alpha beta as this will evaluate all the nodes, not suitable for larger game trees.

Filza Tanveer (CT-22057)

```python
import copy

class TicTacToe:
    def __init__(self):
        self.board = [' ' for _ in range(9)]
        self.current_winner = None
        self.nodes_visited = 0

    def print_board(self):
        for row in [self.board[i*3:(i+1)*3] for i in range(3)]:
            print('| ' + ' | '.join(row) + ' |')

    def available_moves(self):
        return [i for i, spot in enumerate(self.board) if spot == ' ']

    def empty_squares(self):
        return ' ' in self.board

    def make_move(self, square, letter):
        if self.board[square] == ' ':
            self.board[square] = letter
            if self.winner(square, letter):
                self.current_winner = letter
            return True
        return False
```

```python
    def winner(self, square, letter):
        row_ind = square // 3
        row = self.board[row_ind*3:(row_ind+1)*3]
        if all([s == letter for s in row]):
            return True

        col_ind = square % 3
        column = [self.board[col_ind+i*3] for i in range(3)]
        if all([s == letter for s in column]):
            return True

        if square % 2 == 0:
            diagonal1 = [self.board[i] for i in [0,4,8]]
            if all([s == letter for s in diagonal1]):
                return True
            diagonal2 = [self.board[i] for i in [2,4,6]]
            if all([s == letter for s in diagonal2]):
                return True

        return False

def minimax(state, player, maximizing_player):
    state.nodes_visited += 1

    max_player = maximizing_player
    other_player = 'O' if player == 'X' else 'X'
```

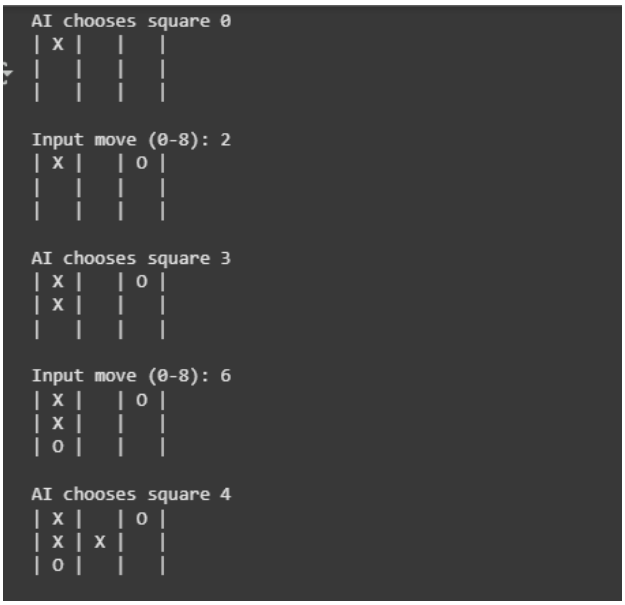
Filza Tanveer (CT-22057)

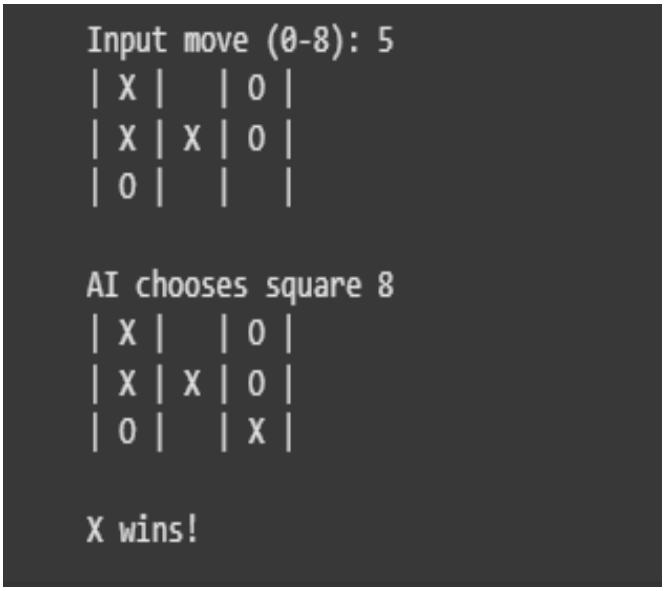
```
    if state.current_winner == other_player:
        return {'position': None, 'score': 1 * (len(state.available_moves()) + 1)
         if other_player == max_player else -1 * (len(state.available_moves()) + 1)}
    elif not state.empty_squares():
        return {'position': None, 'score': 0}

    if player == max_player:
        best = {'position': None, 'score': -float('inf')}
    else:
        best = {'position': None, 'score': float('inf')}

    for possible_move in state.available_moves():
        state.make_move(possible_move, player)
        sim_score = minimax(state, other_player, max_player)

        state.board[possible_move] = ' '
        state.current_winner = None
        sim_score['position'] = possible_move

        if player == max_player:
            if sim_score['score'] > best['score']:
                best = sim_score
        else:
            if sim_score['score'] < best['score']:
                best = sim_score
```

```
    return best

def play():
    game = TicTacToe()
    letter = 'X'

    while game.empty_squares():
        if letter == 'O':
            square = int(input('Input move (0-8): '))
        else:
            move = minimax(game, letter, letter)
            square = move['position']
            print(f"AI chooses square {square}")

        if game.make_move(square, letter):
            game.print_board()
            print('')

            if game.current_winner:
                print(f"{letter} wins!")
                return
            letter = 'O' if letter == 'X' else 'X'

    print("Game Draw")
```

Filza Tanveer (CT-22057)

(OUTPUT)





(TIC TAC TOE GAME CODE USING ALPHA BETA PRUNING ALGORITHM)

CODE EXPLAINATION:

Main Features of code:

In this code **Tic Tac Toe class** consists of following elements:

1. __init__: This will create a nine-space board.
2. show_board():This function will print the board in three-by-three dimensions.
3. available_moves(): Returns the list of indexes (0–8) with an empty square.
4. function has_empty_squares: If at least one square is empty, it returns True.
5. make_move(player, position):If the location is empty, places the player's sign ('X' or 'O') there.
6. check_winner(player, position): This functions checks row and columns
   (Only if the move is on an even index, such as 0, 2, 4, 6, 8) both diagonals and the player wins if they get three in a row).

alpha_beta_pruning function: (game, player, ai_player, alpha, beta).

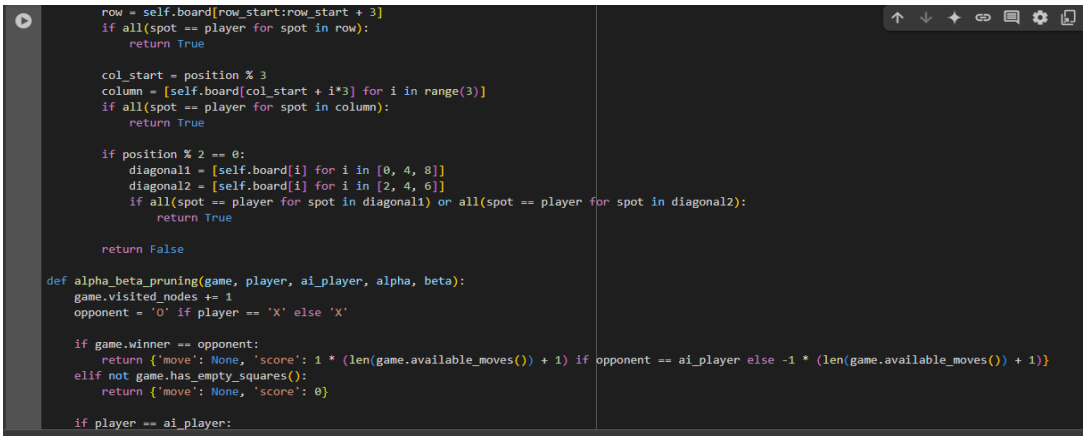
1. Like Minimax, but quicker as it eliminates branches that have little bearing on the ultimate choice.
2. Alpha: The highest score currently guaranteed by the maximizer.
3. Beta: The highest score the minimizer can currently ensure.
4. This tracks the best score for the best move and nodes visited (for analysis) in number.

Filza Tanveer (CT-22057)

Performance:

Faster decision making as it cuts unnecessary branches at the early stage of game.

```
class TicTacToe:
    def __init__(self):
        self.board = [' '] * 9
        self.winner = None
        self.visited_nodes = 0

    def show_board(self):
        for row in range(0, 9, 3):
            print('| ' + ' | '.join(self.board[row:row+3]) + ' |')

    def available_moves(self):
        return [i for i, spot in enumerate(self.board) if spot == ' ']

    def has_empty_squares(self):
        return ' ' in self.board

    def make_move(self, position, player):
        if self.board[position] == ' ':
            self.board[position] = player
            if self.check_winner(position, player):
                self.winner = player
            return True
        return False

    def check_winner(self, position, player):
        row_start = (position // 3) * 3
        row = self.board[row_start:row_start + 3]
```



```
        row = self.board[row_start:row_start + 3]
        if all(spot == player for spot in row):
            return True

        col_start = position % 3
        column = [self.board[col_start + i*3] for i in range(3)]
        if all(spot == player for spot in column):
            return True

        if position % 2 == 0:
            diagonal1 = [self.board[i] for i in [0, 4, 8]]
            diagonal2 = [self.board[i] for i in [2, 4, 6]]
            if all(spot == player for spot in diagonal1) or all(spot == player for spot in diagonal2):
                return True

        return False

def alpha_beta_pruning(game, player, ai_player, alpha, beta):
    game.visited_nodes += 1
    opponent = 'O' if player == 'X' else 'X'

    if game.winner == opponent:
        return {'move': None, 'score': 1 * (len(game.available_moves()) + 1) if opponent == ai_player else -1 * (len(game.available_moves()) + 1)}
    elif not game.has_empty_squares():
        return {'move': None, 'score': 0}

    if player == ai_player:
```

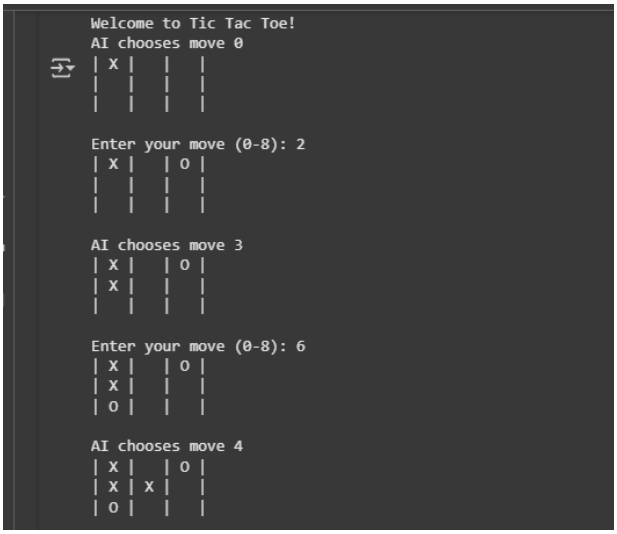
Filza Tanveer (CT-22057)

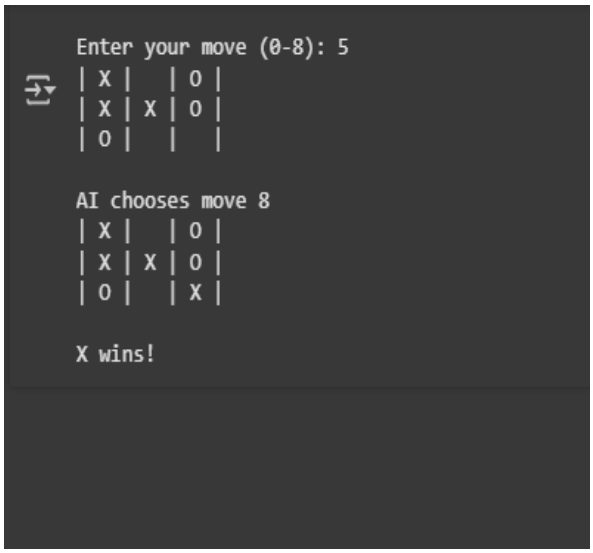
```
    if player == ai_player:
        best = {'move': None, 'score': float('-inf')}
    else:
        best = {'move': None, 'score': float('inf')}

    for move in game.available_moves():
        game.make_move(move, player)
        simulated = alpha_beta_pruning(game, opponent, ai_player, alpha, beta)
        game.board[move] = ' '
        game.winner = None
        simulated['move'] = move

        if player == ai_player:
            if simulated['score'] > best['score']:
                best = simulated
            alpha = max(alpha, simulated['score'])
        else:
            if simulated['score'] < best['score']:
                best = simulated
            beta = min(beta, simulated['score'])

        if beta <= alpha:
            break

    return best
```

```
def play_game():
    print("Welcome to Tic Tac Toe!")

    game = TicTacToe()
    current_player = 'X'

    while game.has_empty_squares():
        if current_player == 'O':
            try:
                move = int(input("Enter your move (0-8): "))
                if move not in game.available_moves():
                    print("Invalid move. Try again.")
                    continue
            except ValueError:
                print("Please enter a number between 0 and 8.")
                continue
        else:
            result = alpha_beta_pruning(game, current_player, current_player, -float('inf'), float('inf'))
            move = result['move']
            print(f"AI chooses move {move}")

        if game.make_move(move, current_player):
            game.show_board()
            print()
```

```
            if game.winner:
                print(f"{current_player} wins!")
                return

            current_player = 'O' if current_player == 'X' else 'X'

    print("Game Draw")

if __name__ == '__main__':
    play_game()
```

(OUTPUT)





(COMPARISION BETWEEN BOTH ALGORITHMS)

FEATURES	ALPHA BETA	MINIMAX
Working	Explore moves but prunes unnecessary branches.	Explores all possible moves and outcomes.
Speed	Fast	Slow
Use Case	Medium level to large game	Small level games
Scalability	Best	Poor

Memory	Less memory usage due to pruning	High memory usage due to whole tree expansion.