Artificial Intelligence & Expert Systems (CT-361)

Assignment 2

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.

Overview

Both the Minimax algorithm and the improved Alpha-Beta Pruning are used in this Tic Tac Toe game's decision-making process. With Minimax examining every move and Alpha-Beta Pruning enhancing performance by removing pointless actions, the game assesses every move to determine which is the greatest option in both strategies is. The AI selects the best moves using either algorithm while the user plays as the second player in the game, which alternates between two people.

Algorithms Explanation

Minimax Algorithm:

In game theory and decision-making, minimax is a type of backtracking algorithm that determines a player's best move, presuming that their opponent plays optimally as well. It is frequently utilized in two-player turn-based games like chess, tic tac toe etc.
The two participants in Minimax are referred to as maximizer and minimizer. While the minimizer seeks to obtain the lowest possible score, the maximizer seeks to obtain the highest possible score.

Time Complexity and Efficiency:

It's straightforward yet inefficient for larger games because it evaluates every possible game state, resulting in the time complexity of O(b^d), where d is the tree's depth and b is the branching factor.

Alpha-Beta Pruning Algorithm:

The minimax algorithm in AI games can be made more efficient by using the alpha-beta pruning algorithm. The method relies on the finding that some game tree branches can be safely pruned (ignored) in many games as they are always going to be worse than other branches.

It lowers the number of nodes by removing branches that have already been shown to be poorer than previously assessed branches using two criteria (alpha and beta).

Time Complexity and Efficiency:

For longer game trees, this outperforms Minimax in terms of speed and reduces the time complexity to O(b^(d/2)) in the best scenario.

Implementation of Tic Tac Toe Using Minimax:

```
import copy

class TicTacToe:

    """Tic Tac Toe game using Minimax"""


    def __init__(self):

        self.game_board = [' ' for _ in range(9)]

        self.winner = None

        self.visited = 0

    def show_board(self):

        for row in [self.game_board[i*3:(i+1)*3] for i in range(3)]:

            print('| ' + ' | '.join(row) + ' |')


    def get_empty_spots(self):

        empty = []

        for i in range(len(self.game_board)):

            if self.game_board[i] == ' ':

                empty.append(i)
```

```
        return empty

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

    def place_marker(self, pos, symbol):
        if self.game_board[pos] == ' ':
            self.game_board[pos] = symbol
            if self.check_winner(pos, symbol):
                self.winner = symbol
            return True
        return False

    def check_winner(self, pos, symbol):
        row = pos // 3
        row_vals = self.game_board[row*3:(row+1)*3]
        if all([spot == symbol for spot in row_vals]):
            return True
        col = pos % 3
        col_vals = [self.game_board[col+i*3] for i in range(3)]
        if all([spot == symbol for spot in col_vals]):
            return True
        if pos % 2 == 0:
            diag1 = [self.game_board[0], self.game_board[4], self.game_board[8]]
            diag2 = [self.game_board[2], self.game_board[4], self.game_board[6]]

            if all([spot == symbol for spot in diag1]) or all([spot == symbol for spot in diag2]):
```

```
            return True
        return False


def minimax(game_state, current_player, ai_player):
    game_state.visited += 1


    max_player = ai_player
    opponent = 'O' if current_player == 'X' else 'X'
    if game_state.winner == opponent:
        multiplier = 1 if opponent == max_player else -1
        return {'pos': None, 'score': multiplier * (len(game_state.get_empty_spots()) + 1)}
    if not game_state.has_empty_squares():
        return {'pos': None, 'score': 0}
    if current_player == max_player:
        best_move = {'pos': None, 'score': float('-inf')}
    else:
        best_move = {'pos': None, 'score': float('inf')}
    for move in game_state.get_empty_spots():


        saved_board = game_state.game_board[:]
        saved_winner = game_state.winner
        game_state.place_marker(move, current_player)
        score = minimax(game_state, opponent, max_player)


        game_state.game_board = saved_board
        game_state.winner = saved_winner
        score['pos'] = move
```

```
        if current_player == max_player:
            if score['score'] > best_move['score']:
                best_move = score
        else:
            if score['score'] < best_move['score']:
                best_move = score

    return best_move

def play_game():
    game = TicTacToe()
    current_player = 'X'

    while game.has_empty_squares():
        if current_player == 'O':
            try:
                move = int(input('Your turn! Enter square (0-8): '))
                while move not in game.get_empty_spots():
                    move = int(input('Invalid move! Try again (0-8): '))
            except ValueError:
                print("Please enter a valid number!")
                continue
        else:
            result = minimax(game, current_player, current_player)
            move = result['pos']
            print(f"AI plays square {move}")
```

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

            if game.winner:
                if current_player == 'X':
                    print("AI wins!")
                else:
                    print("You win!")
                return

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

    print("Game Over - It's a tie!")

if __name__ == '__main__':
    print("Welcome to Tic Tac Toe!")
    print("You'll be O, AI will be X")
    print()
    play_game()
```

Output:

```
Welcome to Tic Tac Toe!
You'll be O, AI will be X

AI plays square 0
| X |   |   |
|   |   |   |
|   |   |   |

Your turn! Enter square (0-8):  5
| X |   |   |
|   |   | O |
|   |   |   |

AI plays square 2
| X |   | X |
|   |   | O |
|   |   |   |

Your turn! Enter square (0-8):  7
| X |   | X |
|   |   | O |
|   | O |   |

AI plays square 1
| X | X | X |
|   |   | O |
|   | O |   |

AI wins!
```

Working:

Without any optimizations mimimax algorithm is implemented over here:

- After recursively analyzing every move that might be made, the AI ("X") returns the highest score.
- The evaluation function check whether the player has won or lost for each state.
- The recursion's base case determines whether the game is over or whether there are no more slots left.
- To determine the optimal choice, the program models every scenario that could occur.

Performance Analysis:

O(b^d), where d is the game tree's depth (number of moves) and b is the branching factor (number of alternative movements per state). In the worst situation, it will explore every move that could be made until the game is over means it would evaluate large number of game states, which could be slow for larger game tree.

Game Flow:

- The game starts with the human player (O) and AI (X).
- The human enters their move.
- All potential future movements are simulated by the AI (using Minimax).
- The AI makes the move and then looks for a winner then it will show the outcome if the game is over (a win or a draw); if not, the loop keeps going.

Implementation Of Tic Tac Toe Using Alpha Beta Pruning:

```
class TicTacToe:
    """Tic Tac Toe Using Alpha-Beta Pruning"""
    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]
        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:
```

```
        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!")
    print("You are 'O'. AI is 'X'.")
    print("Positions are numbered from 0 (top-left) to 8 (bottom-right).\n")

    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("It's a tie!")
if __name__ == '__main__':
```

```
play_game()
```

Output:

```
Welcome to Tic Tac Toe!
You are 'O'. AI is 'X'.
Positions are numbered from 0 (top-left) to 8 (bottom-right).

AI chooses move 0
| X |   |   |
|   |   |   |
|   |   |   |

Enter your move (0-8):  3
| X |   |   |
| O |   |   |
|   |   |   |

AI chooses move 1
| X | X |   |
| O |   |   |
|   |   |   |

Enter your move (0-8):  2
| X | X | O |
| O |   |   |
|   |   |   |

AI chooses move 4
| X | X | O |
| O | X |   |
|   |   |   |

Enter your move (0-8):  7
| X | X | O |
| O | X |   |
|   | O |   |

AI chooses move 8
| X | X | O |
| O | X |   |
|   | O | X |

X wins!
```

Working:

- Minimax is optimized using the Alpha-Beta Pruning technique.
- The algorithm prunes useless portions of the tree using two parameters, alpha and beta:
    - The optimal value for the maximizer (AI) along the route is alpha.
    - The minimizer's (opponent's) optimal value along the path is beta.
    - The current branch gets pruned if it is unable to affect the ultimate choice (due to its inferiority to a branch that has already been explored).

Performance Analysis:

Its an optimized version of minimax algorithm as it prunes unnecessary branches that's why its faster especially for larger game trees. Further, when b is the branching factor and d is the depth, Alpha-Beta Pruning minimizes the time complexity to O(b^(d/2)). The approach can reduce the number of nodes to be evaluated in this situation by half.

Game Flow:

- The game starts with the human player (O) and AI (X).
- The human enters their move.
- To assess and choose the optimum action, the AI employs Alpha-Beta Pruning.
- The AI makes a move and then looks for a winner.
- The outcome (win or tie) is printed if the game is over; if not, the loop keeps going.

Comparison on the Basis of Performance Metrics

Metric	Minimax Algorithm	Alpha Beta Pruning
Scalability	Poor	Good
Execution Time	Slower (specially for deeper search trees)	Faster (due to pruning)
Decision	Optimal	Optimal
Nodes Evaluated	Evaluates each node in tree	Evaluates fewer nodes as it prunes unnecessary branches
Memory Utilization	Utilizes more memory as it needs to store all evaluated nodes.	Utilizes less memory as it prunes branches which reduces stored nodes.