Applied Artificial Intelligence | AI-4007 | Assignment

APRIL 1, 2023

Daniyal Khan : 20i-1847

Assignment:03 | Applied AI

Alpha Beta pruning and MIN-MAX

Table of Contents

Requirements:

Problem Statement

In this assignment, you will be implementing a chess game using the Min-Max Algorithm with Alpha-Beta Pruning. Chess is a two-player board game that is played on an 8x8 chessboard. The game involves moving pieces across the board in an effort to capture the opponent's king. The player who successfully captures the opponent's king wins the game.

Requirements:

1. Implement a chess game that can be played against the computer.
2. The computer will use the Min-Max Algorithm with Alpha-Beta Pruning to make its
1. moves.
2. The game should display the chessboard and pieces after each move.
3. The game should keep track of the moves made by both players.
4. The game should detect checkmate and declare the winner.

5. The game should detect stalemate and declare a draw.
6. The game should have a command line-based user interface to display the chessboard and pieces.

Solution:

A player can play chess against the computer using the provided code. Players can enter moves into the game using UCI notation, and the computer will then randomly generate playable moves. To choose the best move for the computer player, the code also uses an alpha-beta pruning minimax algorithm.

1. The code's first function is get_piece_symbol (piece). The symbol of a chess piece is returned by this function. It accepts a chess piece object as an argument and uses the piece type attribute to determine what Unicode character belongs to which piece to return.

```
def get_piece_symbol(piece):
    """
    Using symbols insted of pieces
    """
    if piece.piece_type == chess.PAWN:
        return "♟"
    elif piece.piece_type == chess.KNIGHT:
        return "♞"
    elif piece.piece_type == chess.BISHOP:
        return "♝"
    elif piece.piece_type == chess.ROOK:
        return "♜"
    elif piece.piece_type == chess.QUEEN:
        return "♛"
    elif piece.piece_type == chess.KING:
        return "♚"
```

2. The chess library is used to create the board object. It simulates a chessboard and keeps track of every piece's location. The moves list is designed to keep track of every move made during the course of the game.

```
board = chess.Board()
moves = []
```

3. Till the game is over, the while loop is in effect. The board is shown inside the loop, with rows numbered 1 through 8 and columns labelled a to h. The symbol of each piece on the board is shown using the function get piece symbol(piece).
4. The player is prompted to enter their move in UCI notation if it is their turn to move. The push uci(move) method is then used to push the input to the board. An error message is shown if the move is invalid.
5. When it's the computer's turn, the random.choice() function generates a random legal move. The push(move) method is then used to push the move to the board. The move is not selected until passed from Alpha and min max algo.

```
while not board.is_game_over():
    #adding side helping material
    print("    a b c d e f g h")


    #printing rows and columns
    for row in range(8):
        print(f"{8 - row} {'|'}", end=" ")
        for col in range(8):
            square = chess.square(col, 7 - row)
            piece = board.piece_at(square)
            if piece is None:
                print(".", end=" ")
            else:
                print(get_piece_symbol(piece), end=" ")
        print(f"{'|'} {8 - row}")

    print("    a b c d e f g h")

    #Computer's / User's turn, appending the board
    if board.turn:
        move = input("Enter your move in UCI notation (e.g. e2e4): ")
        try:
            board.push_uci(move)
            moves.append(move)
        except ValueError:
            print("Invalid move, try again.")
    else:
```

```
        # Generate a random legal move for the computer
        #Returns a set of legal moves
        legal_moves = list(board.legal_moves)
        computer_move = random.choice(legal_moves)
        print("Computer's move:", computer_move.uci())
        board.push(computer_move)
        moves.append(computer_move.uci())

    # Display previous moves
    print("Previous moves:")
    for i, move in enumerate(moves):
        if i % 2 == 0:
            print(f"{i // 2 + 1}. {move}", end=" ")
        else:
            print(move)

    # Give player hints
    #I tried to select hints using Alpha beta or minmax, but there seem to be an
error
    #anyways, not the requiremtns
    if board.turn:
        print("It's your turn. You can:")
        for move in board.legal_moves:
            print(move.uci(), end=" ")
        print()
```

6. The score of a given chess position is determined using the evaluate board(board) function. The function returns infinity or negative infinity depending on which player is in checkmate if the situation is a checkmate. The function returns zero if the situation is a deadlock or there isn't enough material to win. If not, the function calculates the total value of every piece on the board and returns it to the player.

```
def evaluate_board(board):
    if board.is_checkmate():
        if board.turn:
            return float('-inf')
        else:
            return float('inf')
    elif board.is_stalemate() or board.is_insufficient_material():
        return 0
```

```
    else:
        material_score = sum([value for _, value in board.piece_map().items()])
        if board.turn:
            return material_score
        else:
            return -material_score
```

7. The implementation of the minimax algorithm with alpha-beta pruning is the minimax(board, depth, alpha, beta, maximizing player) function. A board object, depth (an integer indicating the maximum depth of the search tree), alpha and beta values (to prune the search tree), and maximizing player (a boolean value indicating whether it is the computer's turn to maximise its score) are all inputs required by the function. The function returns the player's current score and the best move.

```
def minimax(board, depth, alpha, beta, maximizing_player):
    """
    The minimax algorithm with alpha-beta pruning to determine the best move for the
computer player
    """
    if depth == 0 or board.is_game_over():
        return evaluate_board(board), None

    if maximizing_player:
        max_eval = float("-inf")
        best_move = None
        for move in board.legal_moves:
            board.push(move)
            eval = minimax(board, depth - 1, alpha, beta, False)[0]
            board.pop()
            if eval > max_eval:
                max_eval = eval
                best_move = move
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval, best_move
    else:
        min_eval = float("inf")
        best_move = None
        for move in board.legal_moves:
            board.push(move)
```

```
            eval = minimax(board, depth - 1, alpha, beta, True)[0]
            board.pop()
            if eval < min_eval:
                min_eval = eval
                best_move = move
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval, best_move
```

8. The primary function that enables a player to compete against a computer is play(). It accepts user input for their moves and computes the computer's moves using the minimax algorithm. An error message is shown and the player is encouraged to try again if they enter an invalid move. The board and previous moves are shown after each move.

```
def play():
    """
    Main function to play the game against the computer
    """
    board = chess.Board()
    while not board.is_game_over():
        print(board)
        if board.turn == chess.WHITE:
            move = input("Enter your move in UCI notation (e.g. e2e4): ")
            while chess.Move.from_uci(move) not in board.legal_moves:
                print("Illegal move, try again")
                move = input("Enter your move in UCI notation (e.g. e2e4): ")
            board.push(chess.Move.from_uci(move))
            print("Your move: ", move)
        else:
            eval, move = minimax(board, 3, float("-inf"), float("inf"), True)
            board.push(move)
            print("Computer's move: ", move)
            print("Computer's evaluation: ", eval)
        print("Previous moves: ", list(board.move_stack))
    print("Game over, winner: ", board.result())


play()
```

```
print(board.result())
```