Submitted By :

1. Sunil Mittal (BITS ID : 2021SC04968)
2. Indira Saha
3. Vikram Panwar
4. Kirti Karki

**Question 1 – Game Problem statement**

Simulate the working of chess with the below sample smaller game board. A player with any piece/coin standing where its opponent sacrifices all the piece/coin is a win for the first player. Soldier/Pawn (S) can move only one straight step ahead. Only if opponent coins are available, it can move in diagonally one step forward to attack the opponent coin. Horses/Knight(H) can move either forward/backward in L–shaped strides covering exactly 3 tiles. It is the only piece that can jump over other coins Elephant/Rook (E) can move either vertically or horizontally up to maximum length of the board.

A picture containing text

Description automatically generated

1. You are free to choose your own static evaluation function. Justify your choice of static evaluation value design and explain with a sample game state. Do not use any machine learning model for the evaluation function.
2. Similar to the virtual lab example, one of the players must be a human ie., it must get dynamic inputs from us. The other layer must be simulated using the program.
3. Implement Python code for the design under part a, using Minimax Algorithm.

**Answer :**

As, static evaluation function is a function used in AI, especially in game-playing programs, to estimate the value of a game state without actually searching through all possible future moves.

In game-playing programs, such as chess, it is often not feasible to search through all possible future moves from a given game state, as the branching factor can be very high. Therefore, a static evaluation function is used to estimate the value of a game state based on some features of the state, such as the material on the board, the position of the pieces, and the mobility of the pieces.

The static evaluation function is typically designed to be fast and efficient, as it is called many times during a search for the best move. The output of the static evaluation function is used to guide the search algorithm towards more promising branches of the game tree.The accuracy of the static evaluation function can greatly impact the performance of the game-playing program. A better static evaluation function can result in better move selection, leading to stronger play.

For a static evaluation function, I would assign a point value to each piece on the board. For example:

* Soldier/Pawn: 1 point (Solider/Pawn can Move only 1 step)
* Horse/Knight: 3 points (Horse can move 3 steps in L Shaped)
* Elephant/Rook: 2 points (It can move upto max width or length of the board which is 3 and max movement can be two steps in either direction)

This value represents the potential of the piece to capture opponent's pieces and to eventually lead to a win for the player. These values were chosen based on the strength and mobility of each piece in the game.

Example:  
At a given game state, player with white pieces (player 1) has 1 Soldier and 1 Horse and 1 Rook. The player with black pieces (player 2) has only 1 soldier and 1 Horse. The static evaluation value would be: Player 1: 6 points (1 point for Soldier + 3 for Horse + 2 points for Rook) Player 2: 4 points (1 point for Soldier + 3 points for Horse)

In this example, player 1 is ahead in the game as they have more points and a greater potential to win.

To implement a human-computer interface, we can use the standard input/output functions of Python to prompt the human player for their moves and display the current game state. The computer player can use the minimax algorithm to choose its moves as in given example Assume Player 1 Human Assume Player 2 Computer

|  |  |  |
| --- | --- | --- |
| Player 1 Soldier | Player 1 Horse | Player 1 Rook |
|  |  |  |
| Player 1 Soldier | **Player 2 Soldier** | **Player 2 Horse** |

Computer would then use the minimax algorithm to select its move. Let’s assume A (3,2) 🡪 A(2,2)

|  |  |  |
| --- | --- | --- |
| Player 1 Soldier | Player 1 Horse | Player 1 Rook |
|  | **Player 2 Soldier** |  |
|  |  | **Player 2 Horse** |

Program will then prompt the human player to enter their move, the program would then update the game state and display it again, along with a message indicating whose turn it is next:

|  |  |  |
| --- | --- | --- |
|  |  | Player 1 Rook |
|  | **Player 2 Soldier** |  |
| Player 1 Horse |  | **Player 2 Horse** |

The computer would then use the minimax algorithm to select its move, and the process would repeat until one of the players wins or the game ends in a draw.

### Minimax Algorithm:

Minimax algorithm is a decision-making algorithm used in games like chess to determine the best move for a player. It evaluates all possible moves and their outcomes, and chooses the move that gives the maximum advantage for the current player and the minimum advantage for the opponent.

The algorithm works by creating a game tree where each node represents a game state, and the children of the node represent the next possible moves from that state. The algorithm starts at the current state of the game and evaluates the outcome of each possible move. The algorithm continues to evaluate the game tree until it reaches a terminal state (a state where the game is over). The algorithm then chooses the move that gives the maximum advantage for the current player and the minimum advantage for the opponent.

The evaluation is done using a static evaluation function implementing the minimax algorithm, which assigns a point value to each piece based on its strength and mobility. The algorithm uses a technique called alpha-beta pruning to reduce the number of game states that need to be evaluated, making the algorithm more efficient.

*#Static Evaluation Function*

**def** minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):

**if** depth **==** 3:

**return** values[nodeIndex]

**if** maximizingPlayer:

best **=** **-**1000

**for** i **in** range(0, 2):

val **=** minimax(depth **+** 1, nodeIndex **\*** 2 **+** i, **False**, values, alpha, beta)

best **=** max(best, val)

alpha **=** max(alpha, best)

**if** beta **<=** alpha:

**break**

**return** best

**else**:

best **=** 1000

**for** i **in** range(0, 2):

val **=** minimax(depth **+** 1, nodeIndex **\*** 2 **+** i, **True**, values, alpha, beta)

best **=** min(best, val)

beta **=** min(beta, best)

**if** beta **<=** alpha:

**break**

**return** best

values **=** [3, 5, 6, 9, 1, 2, 0, **-**1]

print("The optimal value is : ", end**=**"")

print(minimax(0, 0, **True**, values, **-**1000, 1000))

The optimal value is : 5

This code implements the Minimax algorithm for a game with a small number of moves and a limited search depth (depth=3). The function minimax takes the current game state, the current player, and the search depth as inputs, and returns the best move for the current player. The alpha and beta values are used to implement alpha-beta pruning, which is a technique to reduce the number of game states that need to be evaluated by the algorithm.

**SUMMARY:**

Minimax algorithm is a decision-making method for two-player games such as chess. It evaluates all possible moves and their outcomes by creating a game tree and uses a static evaluation function to assign a value to each board state. The algorithm determines the best move for a player by choosing the one that gives the maximum advantage for the player and the minimum advantage for the opponent. The implementation of Minimax algorithm in Python involves using the minimax function and implementing alpha-beta pruning to optimize performance.

**Implement Python code for the design under part a, using Minimax Algorithm :**

This code implements the Minimax algorithm for a game with a small number of moves and a limited search depth (depth=3). The function minimax takes the current game state, the current player, and the search depth as inputs, and returns the best move for the current player. The alpha and beta values are used to implement alpha-beta pruning, which is a technique to reduce the number of game states that need to be evaluated by the algorithm.

Chess Board Key Combination to Play , kindly review the below table before playing the game.

A picture containing calendar

Description automatically generated

#https://python-chess.readthedocs.io/en/latest/

import chess

import random

# Define the minimax algorithm

def minimax(board, depth, maximizing\_player):

    if depth == 0 or board.is\_game\_over():

        return evaluate\_board(board)

    if maximizing\_player:

        max\_eval = float('-inf')

        for move in board.legal\_moves:

            board.push(move)

            eval = minimax(board, depth - 1, False)

            board.pop()

            max\_eval = max(max\_eval, eval)

        return max\_eval

    else:

        min\_eval = float('inf')

        for move in board.legal\_moves:

            board.push(move)

            eval = minimax(board, depth - 1, True)

            board.pop()

            min\_eval = min(min\_eval, eval)

        return min\_eval

# Define the evaluation function

def evaluate\_board(board):

    piece\_values = {

        chess.PAWN: 1,

        chess.KNIGHT: 3,

        chess.BISHOP: 3,

        chess.ROOK: 5,

        chess.QUEEN: 9,

        chess.KING: 0

    }

    score = 0

    for square in chess.SQUARES:

        piece = board.piece\_at(square)

        if piece is not None:

            value = piece\_values[piece.piece\_type]

            if piece.color == chess.WHITE:

                score += value

            else:

                score -= value

    return score

# Initialize the board

board = chess.Board()

# Define the maximum depth of the minimax algorithm

max\_depth = 2

# Loop until the game ends

while not board.is\_game\_over():

    # Print the current board state

    print(board)

    # Human player's move

    if board.turn == chess.WHITE:

        move = None

        while move is None:

            try:

                #print(board.legal\_moves)

                move = input("Enter your move (in algebraic notation): ")

                move = chess.Move.from\_uci(move)

                if move not in board.legal\_moves:

                    raise ValueError

            except ValueError:

                print("Invalid move. Please try again.")

                move = None

        # Make the move and update the board

        board.push(move)

    # Computer player's move

    else:

        best\_move = None

        best\_eval = float('-inf')

        for move in board.legal\_moves:

            board.push(move)

            eval = minimax(board, max\_depth, False)

            board.pop()

            if eval > best\_eval:

                best\_eval = eval

                best\_move = move

        # Make the move and update the board

        print("Computer moves: " + str(best\_move))

        board.push(best\_move)

# Print the final board state and game result

print(board)

print("Game over. Result: ", board.result())

In this code, the chess module is imported to handle the board and game logic. The minimax function implements the minimax algorithm for the computer player, using the evaluate\_board function as the evaluation function for the leaf nodes of the game tree. The evaluate\_board function assigns a score to the current board state based on the total value of the pieces on the board.

The game loop works similarly to the previous example, but now includes a section for the computer player's move. The computer player uses the minimax algorithm to choose the best move based on the current board state and the maximum depth of the search tree. Once the best move is chosen, it is printed to the console and executed on.

**Question 2 – Logic Problem statement**

A sample decision tree for classifying the given software module as having defects or no-defects is given. Derive if-then rules from this tree and code these rules as Prolog rules. The Prolog code must take the attribute values as input and classify whether the module has defects using the decision tree as reference. The data-set for software defect prediction and its associated variable names are available in the announcement section for reference

Table

Description automatically generated

**Derive the rules from the given decision tree and code as Prolog rules. (35% marks)**

In this code, the defects predicate takes eight input parameters, which are the attribute values of the software module. The code follows the structure of the decision tree to determine whether the module has defects or not. The results are printed using the write predicate either TRUE or FALSE.

defect(N, EvG, IvG, I,i, Loc, D, VG) :- N =< 1, !, write('TRUE').

defect(N, EvG, IvG, I,i, Loc, D, VG) :- N > 1,

    (

    122.55 >= i ->

    (

        0.05 >= I ->

            (

                39.38 >= i ->

                (

                     EvG =< 1.4 ->

                     (

                        19.5 >= D  ->

                        (

                          IvG =< 2 ->

                          (

                            write('TRUE')

                          );

                          IvG > 2 ->

                            Loc =< 50 ->

                            (

                               write('FALSE')

                            );

                            Loc > 50 ->

                                write('TRUE')

                        );

                        D > 19.5 ->

                          write('TRUE')

                     )

                     ;

                     EvG > 1.4 ->

                        write('TRUE')

                )

                ;

                i > 39.38 ->

                    VG =< 3 ->

                    (

                        write('FALSE')

                    );

                    VG > 3 ->

                         IVG =< 8 ->

                         (

                           I =< 0.04 ->

                           (

                                 EvG =< 14 ->

                                 (

                                     write('FALSE')

                                 );

                                 EvG > 14 ->

                                     write('TRUE')

                           );

                           I > 0.04 ->

                            i =<71.94 ->

                                (

                                  D =< 21 ->

                                  (

                                    write('TRUE')

                                  );

                                  D > 21 ->

                                    write('FALSE')

                                );

                                i >71.94 ->

                                  write('FALSE')

                         );

                         IVG > 8 ->

                            write('TRUE')

            );

        I >0.05 ->

            write('FALSE')

    );

    i > 122.55 ->

        write('TRUE')

    ).

**Predicate can be called by looking at the datasheet and values can be taken from datasheet against the last column.**

**Below is the trace enabled output from the above Prolog program :**

**Case 1: when n >1**

**Text, letter

Description automatically generated**

**Case 2 : when N < 1**

**A picture containing text

Description automatically generated**