# ARTIFICIAL INTELLIGENCE PROJECT

**Course Code: UCS411**

**Computer Science and Engineering Department**

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

**Thapar Institute of Engineering and Technology, Patiala, Punjab**

## Submitted by:

- Aseem Goel (102103729)
- Akshita Gupta (102103741)
- Ashmeet Kaur (102103742)
- Pranet Singh Kalra (102103747)

**Group: 2CO26**

Submitted to:
Dr. Swati Kumari

**Problem Statement:** To design a Chess Game in Python using Minimax Algorithm and Alpha Beta Pruning.

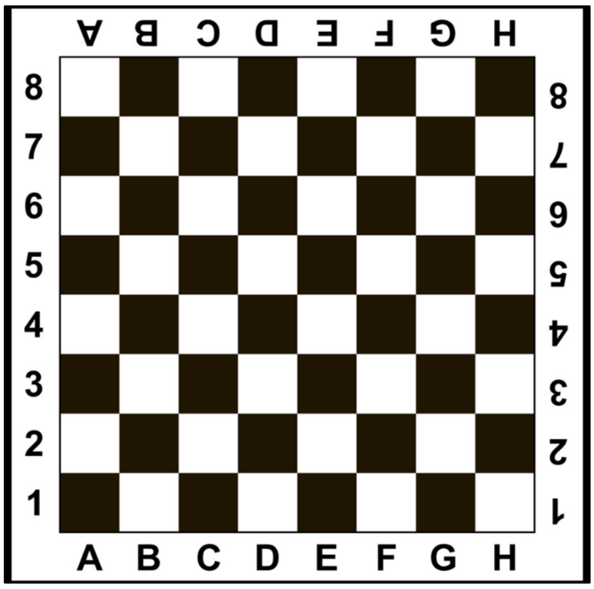
## Description Of Problem:

The main functions and entities that can be used are:

- Board: Board is the one entity represents an actual board on which you play this game.
- Cell: A board consists of a grid of cells.
- Player: Someone who is actually playing right.
- Piece: There are various types of pieces as explained below.

**Pieces and their moves:**

- King: Key entity in chess. If your king is killed then you lose. Its also called checkmate.
- Queen: It can move any number of steps in a single move and in any direction.
- Rook: It only moves in horizontal and vertical direction but can move any number of steps in single move.
- Bishop: It only moves in diagonal direction but can move any number of steps in single move.
- Knight: It makes L shaped moves. Check online for more details about it.
- Pawn: It can move 1 step forward vertically. If it is its first turn, then it can also choose to make 2 steps in single move.
  Note: All pieces except Knight cannot jump any other piece. Knight can jump over other pieces.

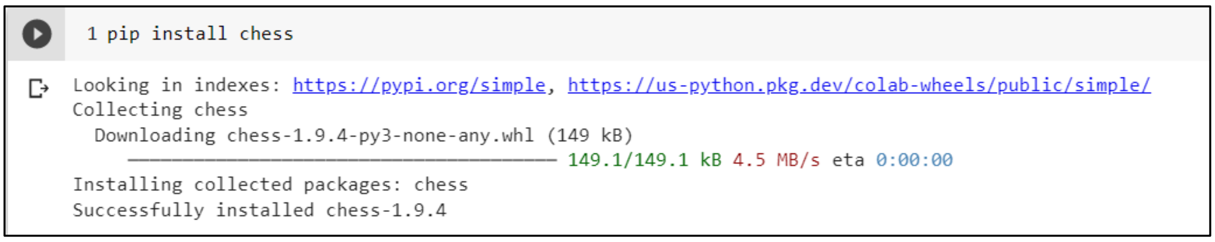


## Link To Google Colaboratory:

https://colab.research.google.com/drive/1_yMgVfAM9roMD8QsHrtg2TCsLtVSWKPr?usp=sharing

## Codes And Explanation:

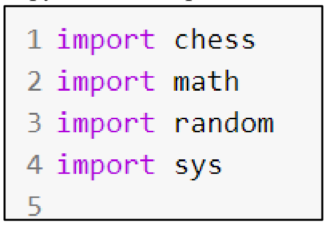
## Modules Imported:



```
pip install chess
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting chess
  Downloading chess-1.9.4-py3-none-any.whl (149 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 149.1/149.1 kB 4.5 MB/s eta 0:00:00
Installing collected packages: chess
Successfully installed chess-1.9.4
```

1. Importing chess library
   The chess module is a pure Python chess library with move generation, move validation and support for common formats. We can play chess with it. It will help us to move the king queen, pawn, bishops and knights. We need to know the basics of chess to play chess with it. This module does every task in python that is possible in the real game.



```
import chess
import math
import random
import sys

```

2. Math module
   Sometimes when working with some kind of financial or scientific projects it becomes necessary to implement mathematical calculations in the project. Python provides the **math module** to deal with such calculations. Math module provides functions to deal with both basic operations such as addition (+), subtraction (-), multiplication (*), division (/) and advance operations like trigonometric, logarithmic, exponential functions.
3. Random module
   Python Random module is an in-built module of Python which is used to generate random numbers. These are pseudo-random numbers means these are not truly random. This module can be used to perform random actions such as generating random numbers, print random a value for a list or string, etc.
4. Sys module
   The **sys** module in Python provides various functions and variables that are used to manipulate different parts of the Python runtime environment. It allows operating on the interpreter as it provides access to the variables and functions that interact strongly with the interpreter

   For implementing Chess, we made use of two algorithms
   - MINIMAX ALGORITHM
   - APLHA BETS PRUNING

   We made two chess programs using these algorithms.

## 1. Using Minimax Algorithm

```python
def minimaxRoot(depth, board,isMaximizing):
    possibleMoves = board.legal_moves
    bestMove = -9999
    secondBest = -9999
    thirdBest = -9999
    bestMoveFinal = None
    for x in possibleMoves:
        move = chess.Move.from_uci(str(x))
        board.push(move)
        value = max(bestMove, minimax(depth - 1, board, not isMaximizing))
        board.pop()
        if( value > bestMove):
            print("Best score: " ,str(bestMove))
            print("Best move: ",str(bestMoveFinal))
            print("Second best: ", str(secondBest))
            thirdBest = secondBest
            secondBest = bestMove
            bestMove = value
            bestMoveFinal = move
    return bestMoveFinal
```

The function minimaxRoot() takes three arguments: the current depth in the search tree, the current state of the board, and a boolean flag indicating whether the current player is maximizing or not. It then generates all possible moves for the current player, and for each move, it applies the minimax algorithm with alpha-beta pruning to determine the best score that can be achieved from this move.

The bestMove, secondBest, and thirdBest variables are used to keep track of the top three moves seen thus far, in case the best move becomes unavailable later. The variable bestMoveFinal is used to maintain track of the best move seen thus far, which will be returned as the final outcome.

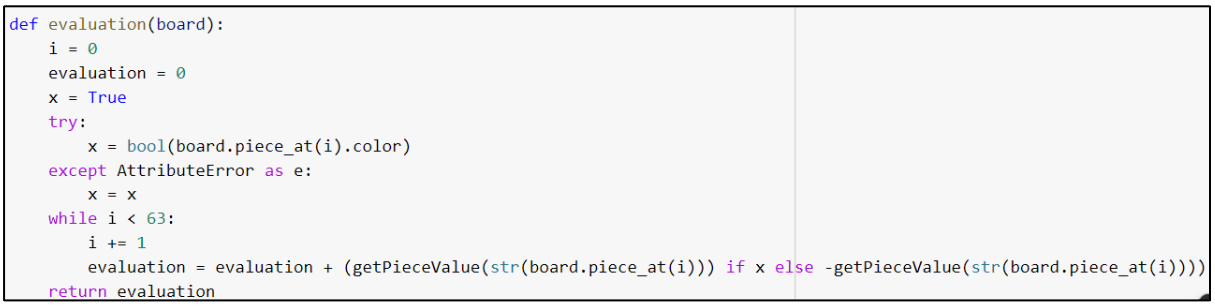
The board.push() and board.pop() methods are used to update the board's state with each move and subsequently undo the change, allowing the search to resume with the original board state. The chess.Move.from_uci() method is used to convert a move from UCI notation to the chess library's own move representation.

```python
def minimax(depth, board, is_maximizing):
    if(depth == 0):
        return -evaluation(board)
    possibleMoves = board.legal_moves
    if(is_maximizing):
        bestMove = -9999
        for x in possibleMoves:
            move = chess.Move.from_uci(str(x))
            board.push(move)
            bestMove = max(bestMove,minimax(depth - 1, board, not is_maximizing))
            board.pop()
        return bestMove
    else:
        bestMove = 9999
        for x in possibleMoves:
            move = chess.Move.from_uci(str(x))
            board.push(move)
            bestMove = min(bestMove, minimax(depth - 1, board, not is_maximizing))
            board.pop()
        return bestMove
```

This code is an implementation of the minimax algorithm for a two-player game, where one player is trying to maximize the score, and the other player is trying to minimize it. The minimax() function takes three arguments: the current depth in the search tree, the current state of the board, and a boolean flag indicating whether the current player is maximizing or not.

If the depth is zero, the function returns the negative evaluation of the board, which assumes that the other player will make the best move in response.. If the current player is maximizing, the function initializes bestMove to a very small negative value and then iterates over all possible moves. For each move, it pushes the move onto the board, calls the minimax() function recursively with the depth reduced by one and the is maximizing flag flipped, and then pops the move from the board.

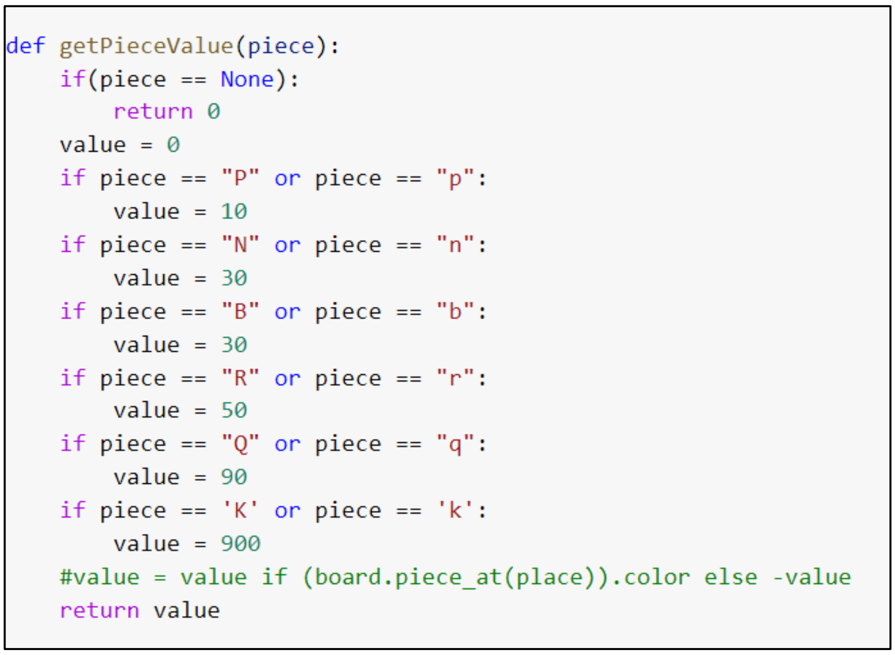
The max() function is used to update bestMove with the maximum value seen so far. If the current player is minimizing, the function initializes bestMove to a very large positive value and follows a similar procedure, but with the min() function used to update bestMove with the minimum value seen so far.



```
def evaluation(board):
    i = 0
    evaluation = 0
    x = True
    try:
        x = bool(board.piece_at(i).color)
    except AttributeError as e:
        x = x
    while i < 63:
        i += 1
        evaluation = evaluation + (getPieceValue(str(board.piece_at(i))) if x else -getPieceValue(str(board.piece_at(i))))
    return evaluation
```

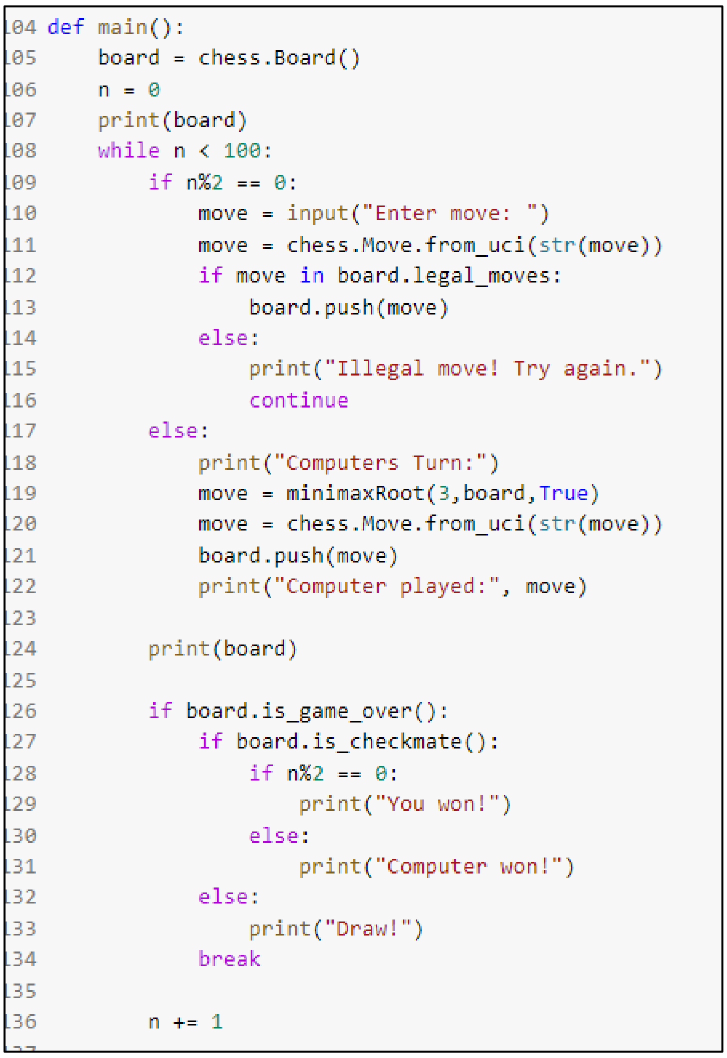
The function evaluation(board) takes a board object as input and returns an evaluation score for the board based on the piece values assigned in the getPieceValue function. The function initializes a counter i to 0 and sets the initial evaluation score to 0. It also sets a boolean variable x to True.

The try-except block attempts to determine the color of the piece at index i, and sets x to the color of the piece if successful. If there is no piece at index i, the try block will raise an AttributeError, but the except block catches this and does not modify the function then enters a loop that iterates through each index of the board from 0 to 63. For each index, the function adds the value of the piece at that index (as determined by the getPieceValue function) to the evaluation score. The value of the piece is negated if x is False. Finally, the function returns the evaluation score.



```
def getPieceValue(piece):
    if(piece == None):
        return 0
    value = 0
    if piece == "P" or piece == "p":
        value = 10
    if piece == "N" or piece == "n":
        value = 30
    if piece == "B" or piece == "b":
        value = 30
    if piece == "R" or piece == "r":
        value = 50
    if piece == "Q" or piece == "q":
        value = 90
    if piece == 'K' or piece == 'k':
        value = 900
    #value = value if (board.piece_at(place)).color else -value
    return value
```

The function getPieceValue(piece) takes a piece string as input (representing a chess piece) and returns an integer value based on the type of piece. The function first checks if piece is None, and returns 0 if so. Otherwise, it sets an initial value of 0 and checks the type of piece using a series of if statements. The function assigns values of 10, 30, 50, 90, and 900 to pawns, knights, bishops, rooks, queens, and kings respectively. The function then returns the value assigned to the piece.



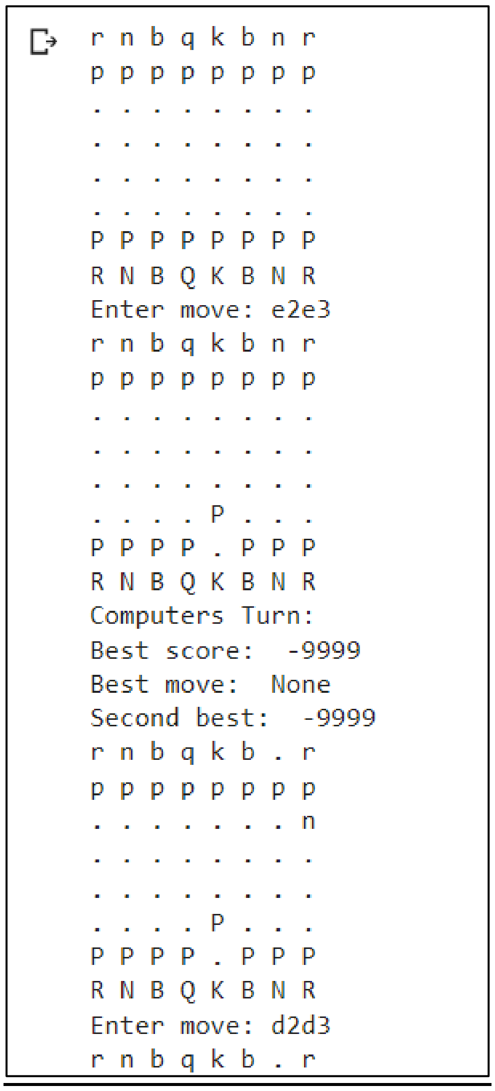
```
def main():
    board = chess.Board()
    n = 0
    print(board)
    while n < 100:
        if n%2 == 0:
            move = input("Enter move: ")
            move = chess.Move.from_uci(str(move))
            if move in board.legal_moves:
                board.push(move)
            else:
                print("Illegal move! Try again.")
                continue
        else:
            print("Computers Turn:")
            move = minimaxRoot(3,board,True)
            move = chess.Move.from_uci(str(move))
            board.push(move)
            print("Computer played:", move)

        print(board)

        if board.is_game_over():
            if board.is_checkmate():
                if n%2 == 0:
                    print("You won!")
                else:
                    print("Computer won!")
            else:
                print("Draw!")
            break

        n += 1
```

The main() function initializes the chess board, sets the number of turns to zero, and prints the initial board. It then enters a loop that continues until the game is over or 100 turns have been played. Inside the loop, it alternates between the user and computer's turn. For the user's turn (when n%2 == 0), the code prompts the user to enter a move in UCI notation. If the move is legal, it is applied to the board with board.push(move). If the move is illegal, the code prints an error message and continues to the next iteration of the loop.

For the computer's turn (when n%2 == 1), the code calls the minimaxRoot() function with a depth of 3 and the current board state as input. This function uses the minimax algorithm with alpha-beta pruning to determine the best move for the computer. The selected move is then applied to the board with board.push(move).

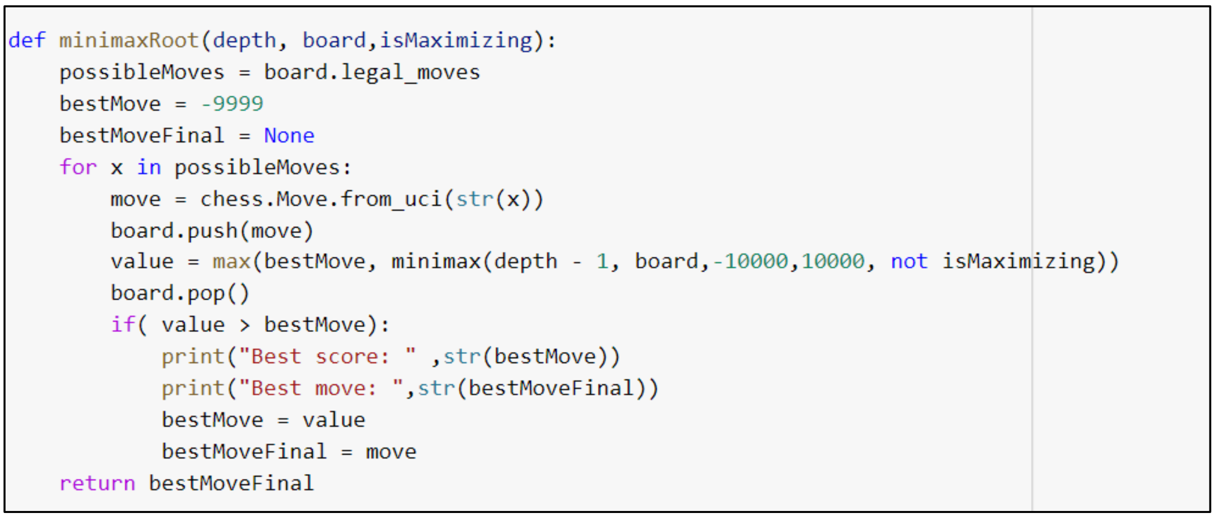
After each turn, the code prints the updated board. If the game is over (due to checkmate, stalemate, or a draw), the code determines the winner and prints the result. The loop exits and the program terminates.

**OUTPUT**



```
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
P P P P P P P P
R N B Q K B N R
Enter move: e2e3
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . P . . .
P P P P . P P P
R N B Q K B N R
Computers Turn:
Best score:  -9999
Best move:  None
Second best:  -9999
r n b q k b . r
p p p p p p p p
. . . . . . . n
. . . . . . . .
. . . . . . . .
. . . . P . . .
P P P P . P P P
R N B Q K B N R
Enter move: d2d3
r n b q k b . r
```

## 2. Using Alpha Beta Pruning



```python
def minimaxRoot(depth, board,isMaximizing):
    possibleMoves = board.legal_moves
    bestMove = -9999
    bestMoveFinal = None
    for x in possibleMoves:
        move = chess.Move.from_uci(str(x))
        board.push(move)
        value = max(bestMove, minimax(depth - 1, board,-10000,10000, not isMaximizing))
        board.pop()
        if( value > bestMove):
            print("Best score: " ,str(bestMove))
            print("Best move: ",str(bestMoveFinal))
            bestMove = value
            bestMoveFinal = move
    return bestMoveFinal
```

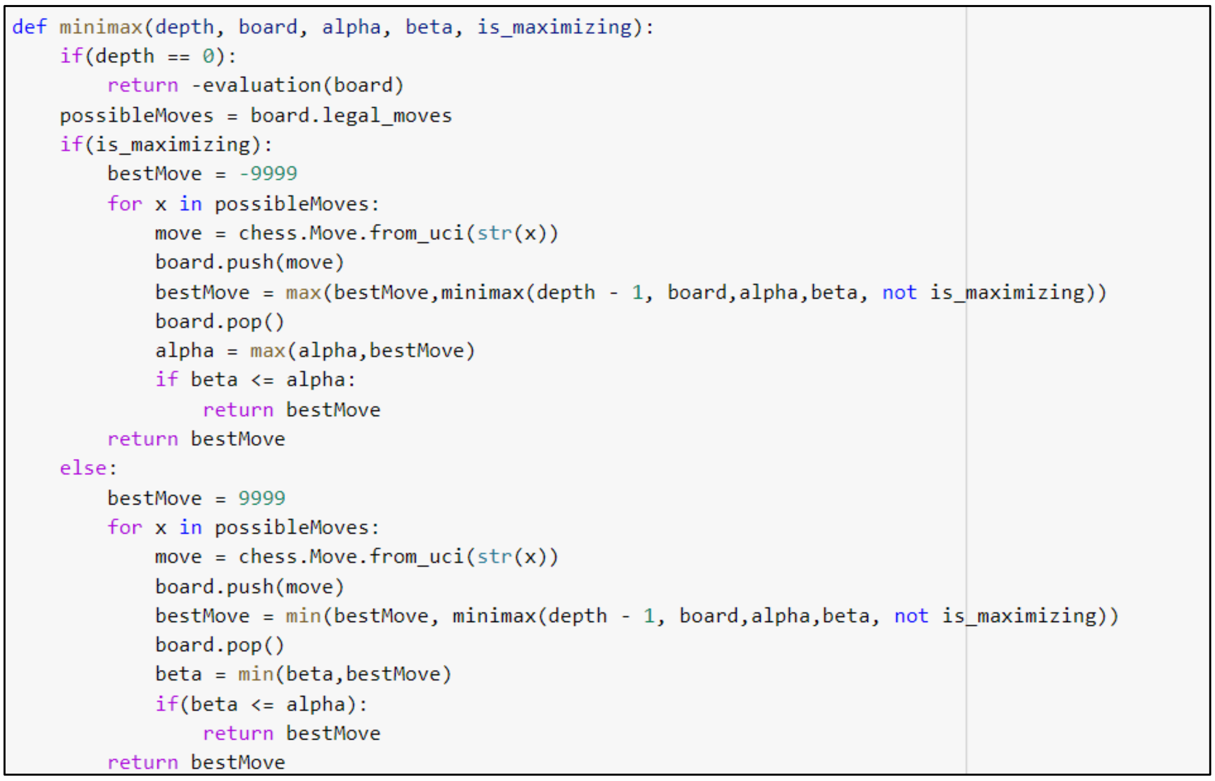
The function takes in a few parameters:

depth: The depth of the search tree to explore.

board: The current state of the chess board.

isMaximizing: A boolean variable that indicates whether the function is currently maximizing or minimizing the score.

The function first gets a list of all possible moves that can be made from the current state of the board. It then initializes some variables to keep track of the best move and score seen so far. For each possible move, the function generates the move, applies it to the board, and calls the minimax function (which is presumably defined elsewhere) with the new board state and updated depth. The minimax function returns a score for the new board state. The function then reverts the board to its previous state (by calling pop()), and if the score returned by minimax is better than the current best score, it updates the best score and move seen so far. Finally, the function returns the best move found.



```python
def minimax(depth, board, alpha, beta, is_maximizing):
    if(depth == 0):
        return -evaluation(board)
    possibleMoves = board.legal_moves
    if(is_maximizing):
        bestMove = -9999
        for x in possibleMoves:
            move = chess.Move.from_uci(str(x))
            board.push(move)
            bestMove = max(bestMove,minimax(depth - 1, board,alpha,beta, not is_maximizing))
            board.pop()
            alpha = max(alpha,bestMove)
            if beta <= alpha:
                return bestMove
        return bestMove
    else:
        bestMove = 9999
        for x in possibleMoves:
            move = chess.Move.from_uci(str(x))
            board.push(move)
            bestMove = min(bestMove, minimax(depth - 1, board,alpha,beta, not is_maximizing))
            board.pop()
            beta = min(beta,bestMove)
            if(beta <= alpha):
                return bestMove
        return bestMove
```

The function takes in a few parameters:

depth: The depth of the search tree to explore.

board: The current state of the chess board.

alpha: The alpha value used for alpha-beta pruning.

beta: The beta value used for alpha-beta pruning.

is_maximizing: A boolean variable that indicates whether the function is currently maximizing or minimizing the score.

The function first checks if the depth has reached 0. If so, it returns the negative of the evaluation of the board. This is because the algorithm assumes that the current player is trying to maximize the score, so if the depth has reached 0, the algorithm assumes that it's the opponent's turn, and thus the negative of the evaluation should be returned. If the depth has not reached 0, the function generates a list of all possible moves from the current board state.

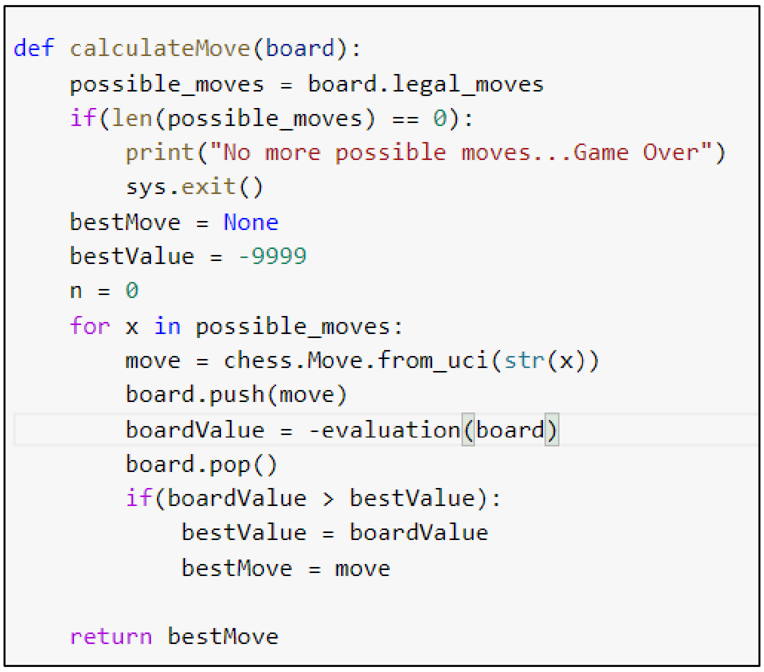
If is_maximizing is true, the function initializes bestMove to a very low value (-9999), and then iterates through each possible move. For each move, the function generates the move, applies it to the board, and recursively calls minimax with the updated board state, decreased depth, and the opposite value of is_maximizing (since it's now the opponent's turn). The minimax function returns the score for that move.

After the recursive call to minimax, the function reverts the board to its previous state, updates the bestMove value if the new score is higher than the current best score, and updates the alpha value if the new bestMove is greater than the current alpha value. If beta is less than or equal to alpha, the function returns the current bestMove.

If is_maximizing is false, the function initializes bestMove to a very high value (9999), and then iterates through each possible move. For each move, the function generates the move, applies it to the board, and recursively calls minimax with the updated board state, decreased depth, and the opposite value of is_maximizing (since it's now the maximizing player's turn). The minimax function returns the score for that move.

After the recursive call to minimax, the function reverts the board to its previous state, updates the bestMove value if the new score is lower than the current best score, and updates the beta value if the new bestMove is less than the current beta value. If beta is less than or equal to alpha, the function returns the current bestMove.

The function returns the bestMove value at the end.

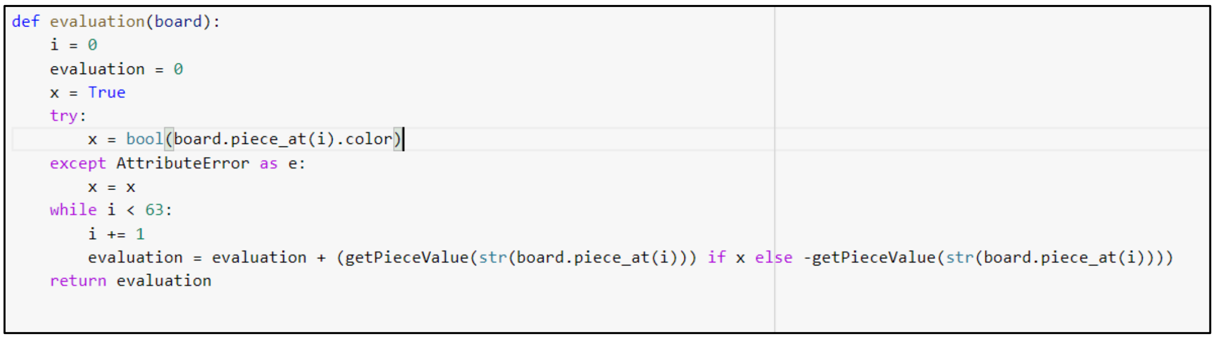


```python
def calculateMove(board):
    possible_moves = board.legal_moves
    if(len(possible_moves) == 0):
        print("No more possible moves...Game Over")
        sys.exit()
    bestMove = None
    bestValue = -9999
    n = 0
    for x in possible_moves:
        move = chess.Move.from_uci(str(x))
        board.push(move)
        boardValue = -evaluation(board)
        board.pop()
        if(boardValue > bestValue):
            bestValue = boardValue
            bestMove = move

    return bestMove
```

The function takes in a single parameter board, which represents the current state of the chess board. The function first generates a list of all possible legal moves from the current board state. If there are no possible moves, the function prints a message and exits the program.

The function then initializes the bestMove variable to None and bestValue to a very low value (-9999). It also initializes n to 0. The function then iterates through each possible move. For each move, the function generates the move, applies it to the board, and calculates the negative evaluation of the new board state using the evaluation function. The function then reverts the board to its previous state. If the calculated boardValue is greater than the current bestValue, the function updates bestValue to the new value and bestMove to the current move.

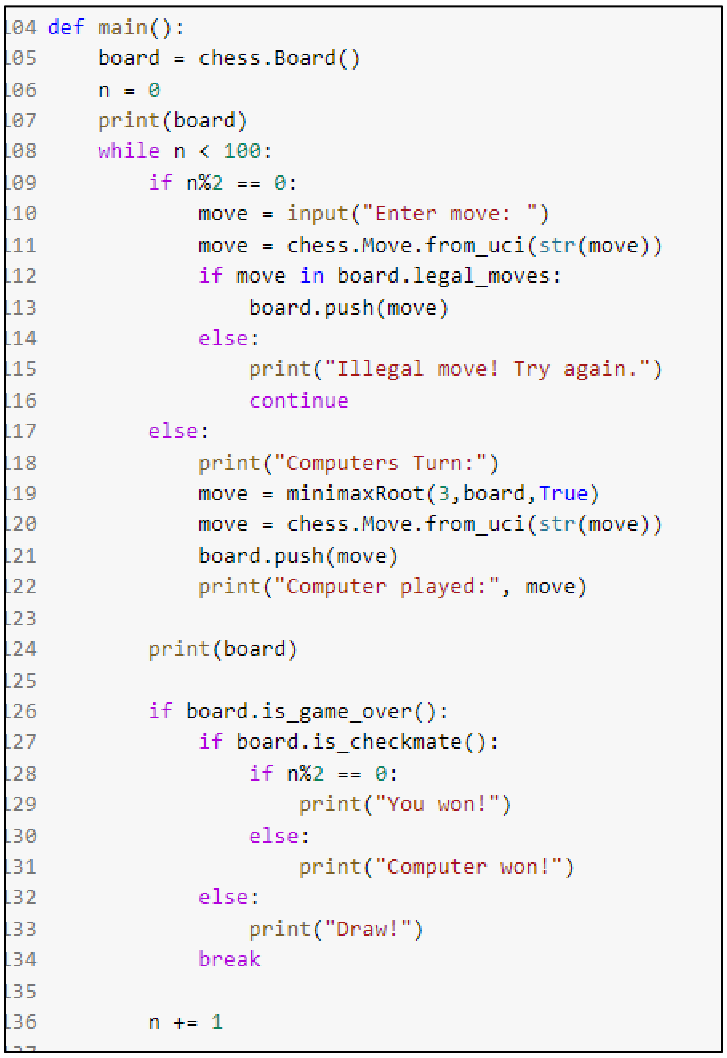
After all possible moves have been considered, the function returns bestMove, which represents the best move to make based on the negative evaluation of the resulting board states.



```python
def evaluation(board):
    i = 0
    evaluation = 0
    x = True
    try:
        x = bool(board.piece_at(i).color)
    except AttributeError as e:
        x = x
    while i < 63:
        i += 1
        evaluation = evaluation + (getPieceValue(str(board.piece_at(i))) if x else -getPieceValue(str(board.piece_at(i))))
    return evaluation
```

This is a Python function that evaluates a given chess board state and returns a numerical score representing how favorable the board is for the player whose turn it is. The function takes in a single parameter board, which represents the current state of the chess board.

The function first initializes the i variable to 0 and the evaluation variable to 0. It also initializes the x variable to True. The function then enters a loop that iterates through all 64 squares on the chess board. For each square, the function calculates the value of the piece on the square using the getPieceValue function, and adds this value to the evaluation variable. The function also checks whether the color of the piece is the same as the player whose turn it is (using the bool(board.piece_at(i).color) expression) and multiplies the value by 1 if it is, or by -1 if it is not. After all squares have been considered, the function returns the final evaluation value, which represents the overall score of the board state.



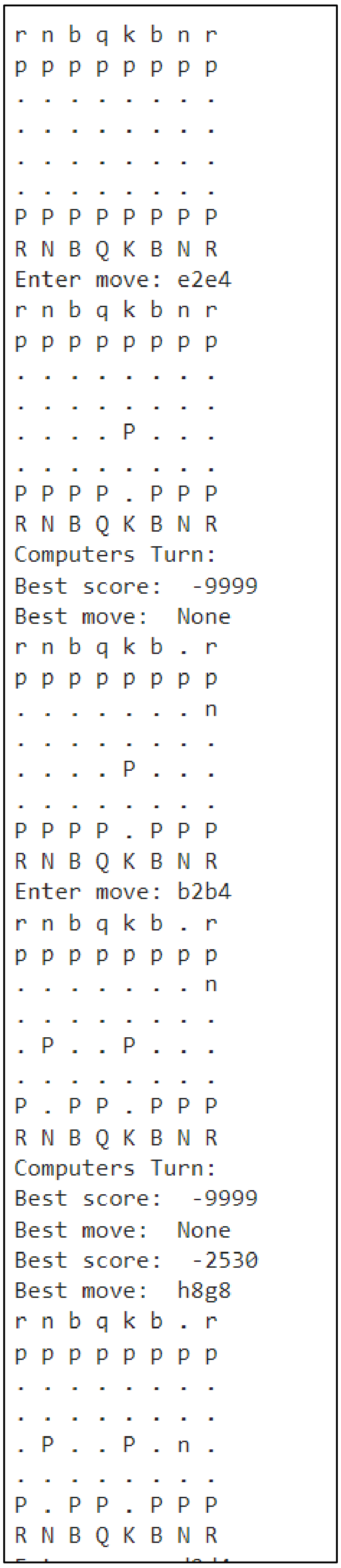
```python
def main():
    board = chess.Board()
    n = 0
    print(board)
    while n < 100:
        if n%2 == 0:
            move = input("Enter move: ")
            move = chess.Move.from_uci(str(move))
            if move in board.legal_moves:
                board.push(move)
            else:
                print("Illegal move! Try again.")
                continue
        else:
            print("Computers Turn:")
            move = minimaxRoot(3,board,True)
            move = chess.Move.from_uci(str(move))
            board.push(move)
            print("Computer played:", move)

        print(board)

        if board.is_game_over():
            if board.is_checkmate():
                if n%2 == 0:
                    print("You won!")
                else:
                    print("Computer won!")
            else:
                print("Draw!")
            break

        n += 1
```

The main() function initializes the chess board, sets the number of turns to zero, and prints the initial board. It then enters a loop that continues until the game is over or 100 turns have been played. Inside the loop, it alternates between the user and computer's turn. For the user's turn (when n%2 == 0), the code prompts the user to enter a move in UCI notation. If the move is legal, it is applied to the board with board.push(move). If the move is illegal, the code prints an error message and continues to the next iteration of the loop.

For the computer's turn (when n%2 == 1), the code calls the minimaxRoot() function with a depth of 3 and the current board state as input. This function uses the minimax algorithm with alpha-beta pruning to determine the best move for the computer. The selected move is then applied to the board with board.push(move).

After each turn, the code prints the updated board. If the game is over (due to checkmate, stalemate, or a draw), the code determines the winner and prints the result. The loop exits and the program terminates.

**OUTPUT:**



```
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
P P P P P P P P
R N B Q K B N R
Enter move: e2e4
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . P . . .
. . . . . . . .
P P P P . P P P
R N B Q K B N R
Computers Turn:
Best score:  -9999
Best move:  None
r n b q k b . r
p p p p p p p p
. . . . . . . n
. . . . . . . .
. . . . P . . .
. . . . . . . .
P P P P . P P P
R N B Q K B N R
Enter move: b2b4
r n b q k b . r
p p p p p p p p
. . . . . . . n
. . . . . . . .
. P . . P . . .
. . . . . . . .
P . P P . P P P
R N B Q K B N R
Computers Turn:
Best score:  -9999
Best move:  None
Best score:  -2530
Best move:  h8g8
r n b q k b . r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. P . . P . n .
. . . . . . . .
P . P P . P P P
R N B Q K B N R
```

## CONCLUSION:

Alpha Beta Pruning is an optimization method to the minimax algorithm that allows us to disregard some branches in the search tree. This helps us evaluate the minimax search tree much deeper, while using the same resources. Alpha-beta pruning is based on the situation where we can stop evaluating a part of the search tree if we find a move that leads to a worse situation than a previously discovered move. Alpha-beta pruning does not influence the outcome of the minimax algorithm — it only makes it faster. Alpha-beta algorithm also is more efficient if we happen to visit first those paths that lead to good moves.

Thank You