ARTIFICIAL INTELLIGENCE

(UCS411)

DEPARTMENT OF

COMPUTER SCIENCE AND ENGINEERING

PROJECT REPORT

SLOTH CHESS ENGINE

THAPAR INSTITUTE

OF ENGINEERING AND TECHNOLOGY,

PATIALA-147004, PUNJAB

SUBMITTED BY:

HEMANT (102217141)

GARIMA SHARMA (102217171)

SUBMITTED TO:

DR. ASHISH BAJAJ

INTRODUCTION

Chess has been a game of strategy and skill for centuries, captivating players of all ages and backgrounds. With the rise of artificial intelligence (AI), chess engines have become increasingly advanced, providing formidable opponents for even the most skilled players. In this report, we introduce "Sloth" a slow yet a classical chess engine that operates on the fundamentals of AI.

Sloth is designed to analyze chess positions and make informed decisions based on a variety of factors, including board evaluation, piece mobility, and potential future moves. By utilizing cutting-edge AI algorithms, Sloth is capable of quickly analyzing a large number of potential moves and determining the best course of action.

Overall, Sloth represents a significant step forward in the development of AI-powered chess engines. Its advanced decision-making capabilities and ability to learn and adapt make it a formidable opponent for even the most skilled human players.

LITERATURE REVIEW

HERE IS A BRIEF REGARDING THE CHESS ENGINES THAT HAVE BEEN DEVELOPED OVER TIME AND HAVE MADE SIGNIFICANT CHANGES IN HOW THE GAME OF CHESS IS PERCEIVED

1. "Deep Blue" by Feng-Hsiung Hsu: This paper details the development of the chess engine that famously defeated world champion Garry Kasparov in 1997, using a combination of brute-force search and heuristic evaluation.
2. "AlphaZero" by David Silver et al.: This groundbreaking paper describes the development of an AI-powered chess engine that taught itself to play chess at a superhuman level through reinforcement learning, without any prior human knowledge of the game.
3. "Stockfish" by Tord Romstad et al.: This paper details the development of the popular open-source chess engine, which uses a combination of alpha-beta search, transposition tables, and sophisticated evaluation functions to play at a world-class level.
4. "Komodo" by Don Dailey and Larry Kaufman: This paper describes the development of a chess engine that uses a combination of Monte Carlo tree search and sophisticated evaluation functions to play at a superhuman level.
5. "Leela Chess Zero" by Gary Linscott: This paper details the development of a neural network-based chess engine that

learned to play the game through self-play, and has since gone on to achieve superhuman performance.

6. "Pulsar" by Julien Kloetzer et al.: This paper describes the development of a chess engine that uses a neural network-based evaluation function, and achieved strong results in the 2017 TCEC competition.
7. "AllieStein" by Timo Haupt and Ryan Hayward: This paper details the development of a chess engine that uses a neural network-based evaluation function and Monte Carlo tree search, achieving impressive results in both the TCEC and CCC competitions.
8. "LCZero" by Alexander Lyashuk: This paper describes the development of a neural network-based chess engine that uses a combination of supervised and reinforcement learning, achieving superhuman performance in self-play and tournaments.
9. "Fat Fritz 2.0" by Albert Silver: This paper details the development of a neural network-based chess engine that uses a combination of supervised and unsupervised learning, achieving superhuman performance in self-play and strong tournament results.
10. "RofChade" by Gerd Isenberg et al.: This paper describes the development of a chess engine that uses a neural network-based evaluation function and Monte Carlo tree search, achieving impressive results in the TCEC competition.

METHODOLOGY

Took Motivation/help from:

http://liamvallance.com/img/Exploring%20python%20chess.pdf

https://www.chess.com

https://lichess.org

ALGORITHM

A total of three algorithms were used namely:

1) **MIN-MAX ALGORITHM:** The Minimax algorithm is a decision-making algorithm that considers all possible moves in a two-player game and selects the best move that minimizes the maximum loss.
2) **NEGA-MAX ALGORITHM:** The Nega-Max algorithm is a variation of the Minimax algorithm that simplifies the search process by negating the evaluation function and only considering the maximum of the negated values.
3) **ALPHA BETA PRUNING:** Alpha-beta pruning is a technique used in computer algorithms for gameplaying that reduces the number of nodes searched by discarding those that cannot affect the final decision.

STEPS OF ALGORITHM:

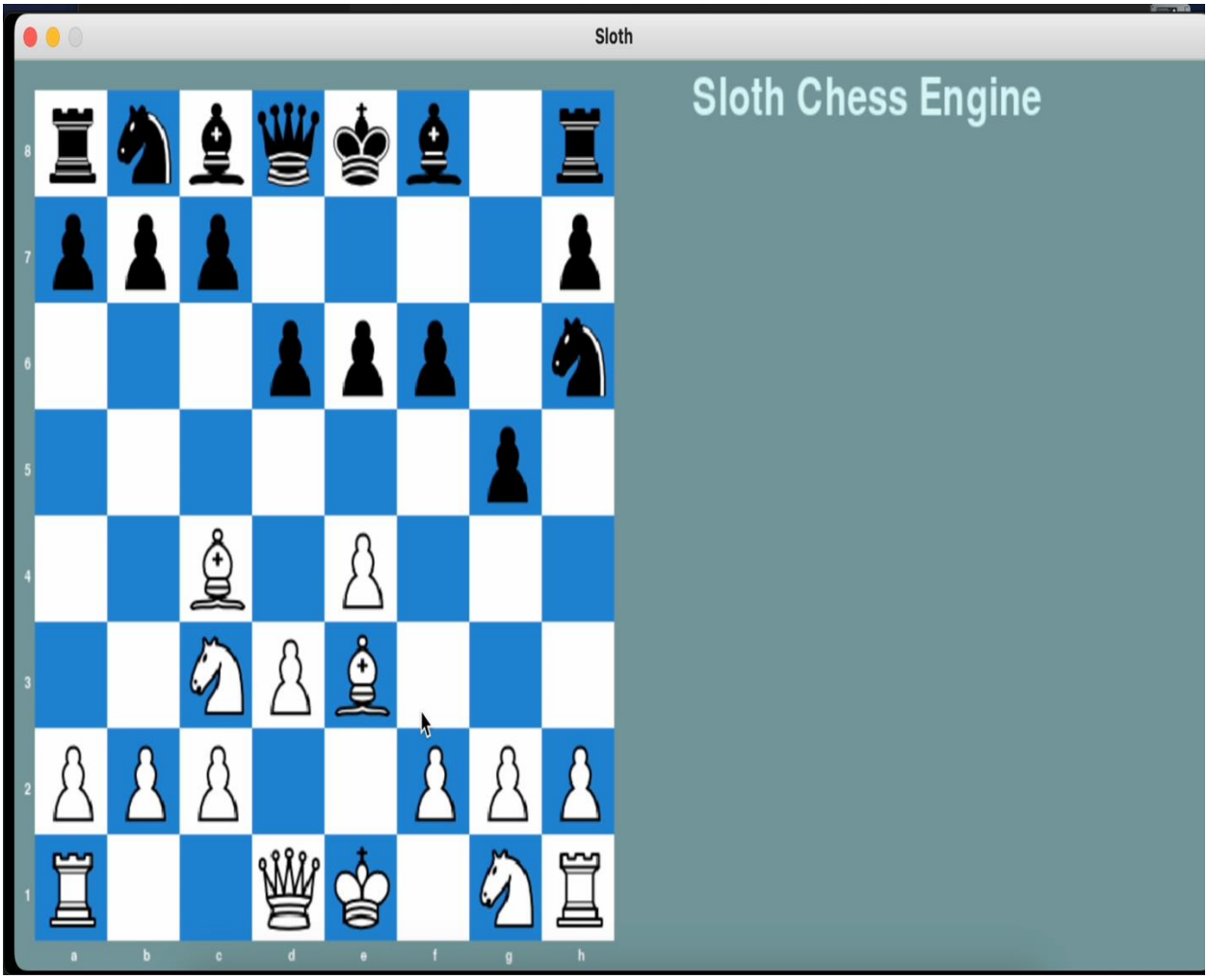
Initially min max algorithm worked and created a tree with time complexity exponential to the depth of the game tree, which resulted in quite a vast search space.

Next the NegaMax algorithm was implemented which is not necessarily better than the min-max algorithm but results in a more concise and faster implementation of the algorithm by using the property that in two-player zero-sum games, the value of the game from the perspective of one player is the negation of the

value of the game from the perspective of the other player. This allows the algorithm to search only one branch of the game tree and negate the result rather than searching both branches separately as in Minimax. However, the problem of a very vast search space still existed.

This is when the alpha-beta pruning algorithm was used which is the most efficient out of the three given algorithms. It significantly reduces the time complexity as much as to **O(b^d/2)**[b -> branching factor of the tree, d -> depth of the tree].

HOW IT LOOKS:



CODE AND EXPLANATION

GUI:

```
def loadImages():
    pieces = np.array(["bR", "bN", "bB", "bK", "bQ", "bB", "bB", "bN", "bR","bP",
              "wR", "wN", "wB", "wK", "wQ", "wB", "wB", "wN", "wR","wP"])
    for piece in pieces:
        IMAGES[piece] = p.transform.scale(p.image.load("images/"+piece+".png"),(CELL_SIZE,CELL_SIZE))
        #now we can access the IMAGES dictionary

# GUI
def drawGameState(screen,gameState,validMoves,sqSelected):
    drawBoardAndPieces(screen,gameState)       #draw cells & pieces
    highlightCells(screen, gameState, validMoves, sqSelected)

 def drawRanksAndFiles(screen):
     title = "BEAVER CHESS ENGINE"
     titleBox = HEADING.render(title,True,HEADINGCOL)
     titleBox.get_width()
     screen.blit(titleBox, (WIDTH *1.1+ (2*INDENT), CELL_SIZE ))
     for i in range(DIMENSION):
         rank = str(DIMENSION-i)
         rankBox = FONT.render(rank, True, WHITE)
         screen.blit(rankBox, (INDENT//2, i*CELL_SIZE + INDENT +CELL_SIZE//2))
     files = {7: "a",6:  "b",5: "c",4: "d",3: "e",2: "f",1: "g",0: "h"}
     for j in range(DIMENSION):
         file = files[DIMENSION-j-1]
         fileBox = FONT.render(file, True, WHITE)
         screen.blit(fileBox, (j*CELL_SIZE + INDENT +CELL_SIZE//2,HEIGHT+ INDENT+INDENT//4))


 def drawBoardAndPieces(screen,gameState):
     for i in range(DIMENSION):
         for j in range(DIMENSION):
             color = WHITE if (i+j)%2==0 else GREY
             x = j * CELL_SIZE + INDENT
             y = i * CELL_SIZE + INDENT
             square =p.Rect(x,y,CELL_SIZE,CELL_SIZE)
             p.draw.rect(screen,color,square,width=CELL_SIZE)
             piece = gameState.board[i][j]
             if piece != "--":
                 x = j * CELL_SIZE + INDENT
                 y = i * CELL_SIZE + INDENT
                 screen.blit(IMAGES[piece],(x,y))
```

```
def highlightCells(screen,gameState,validMoves,sqSelected): #Highlights the moves
    if sqSelected !=():
        row,col = sqSelected
        me = "w" if gameState.whiteToMove else "b"
        if gameState.board[row][col][0] == me:
            #highlight selected square
            sq = p.Surface((CELL_SIZE,CELL_SIZE))
            sq.set_alpha(50)
            sq.fill(HIGHLIGHT1)
            x = col * CELL_SIZE + INDENT
            y = row * CELL_SIZE + INDENT
            screen.blit(sq,(x,y))
            # Highlight moves
            for move in validMoves:
                if move.startRow == row and move.startCol == col:
                    sq.set_alpha(120)
                    sq.fill(HIGHLIGHT2)
                    x = move.endCol * CELL_SIZE + INDENT
                    y = move.endRow * CELL_SIZE + INDENT
                    screen.blit(sq, (x, y))
```

The code above is a Python script that creates a chess board in Pygame, which is a popular Python library used for creating games and multimedia applications.

The script first imports the **`np.array`** module and initializes it. It also defines some colors that will be used to draw the squares on the board.

Next, it sets the dimensions of the board and creates a Pygame **`Surface`** object called **`board_surface`** with the same dimensions. This surface will be used to draw the squares on the board.

The script then creates an 8x8 chess board array called **`board`**, which is a two-dimensional array of **`Rect`** objects. Each **`Rect`** object represents a square on the board and is added to the **`board_surface`** surface using the **`p.array.draw.rect`**() function. The **`Rect`** objects are also added to the **`board`** array for later use.

The script then creates a Pygame **`Display`** object called **`screen`** with the same dimensions as the board and displays the board by blitting the **`board_surface`** onto the **`screen`** using the **`screen.blit()`** function.

RULES SETUP:

```
def makeMove(self,gameMove): # Not works for castling, en passant and pawn promotion
    self.board[gameMove.startRow][gameMove.startCol] = "--"
    self.board[gameMove.endRow][gameMove.endCol] = gameMove.pieceMoved
    self.moveLog.append(gameMove)
    #swap player turns
    self.whiteToMove = not self.whiteToMove
    if gameMove.pieceMoved == "wK":
        self.whiteKingLoc = (gameMove.endRow,gameMove.endCol)
    elif gameMove.pieceMoved == "bK":
        self.blackKingLoc = (gameMove.endRow,gameMove.endCol)

def undoMove(self):
    if len(self.moveLog)!=0:
        lastMove = self.moveLog.pop()
        self.board[lastMove.startRow][lastMove.startCol] = lastMove.pieceMoved
        self.board[lastMove.endRow][lastMove.endCol] = lastMove.pieceCaptured
        self.whiteToMove = not self.whiteToMove
        if lastMove.pieceMoved == "wK":
            self.whiteKingLoc = (lastMove.startRow, lastMove.startCol)
        elif lastMove.pieceMoved == "bK":
            self.blackKingLoc = (lastMove.startRow, lastMove.startCol)
        self.checkmate = False
        self.stalemate = False
```

```python
def getAllPossibleMoves(self): #All moves with checks
    moves = []
    for i in range(len(self.board)): #rows
        for j in range(len(self.board[0])): #cols
            pieceColor = self.board[i][j][0]
            if(self.whiteToMove and pieceColor == "w") or (not self.whiteToMove and pieceColor == "b"):
                piece = self.board[i][j][1]
                if piece == 'P':
                    self.getPawnMoves(i, j, moves)
                elif piece == 'R':
                    self.getRookMoves(i, j, moves)
                elif piece == 'N':
                    self.getKnightMoves(i, j ,moves)
                elif piece == 'B':
                    self.getBishopMoves(i ,j ,moves)
                elif piece == 'K':
                    self.getKingMoves(i, j, moves)
                elif piece == 'Q':
                    self.getQueenMoves(i, j, moves)
    return moves
```

The following code describes and denotes all the possible moves that a chess piece can make along with stating that when a piece can make a move and also undo said move.

ALGORITHMS USED AND POSTIONAL TABLE:

```
def RandomAI(validMoves):
    return random.choice(validMoves)

def ScoreMaterial(board):
    score = 0
    for row in board:
        for ele in row:
            if ele[0] == "w":
                score += pieceScore[ele[1]]
            elif ele[0] == "b":
                score -= pieceScore[ele[1]]
    return score

def ScoreBoard(gameState):
    if gameState.checkmate:
        if gameState.whiteToMove:
            return -CHECKMATE   #blackwins
        else:
            return CHECKMATE    #blackwins
    elif gameState.stalemate:
        return STALEMATE        #score 0
    score = 0
    for row in range(len(gameState.board)):
        for col in range(len(gameState.board[0])):
            piece = gameState.board[row][col]
            if piece !="--":
                #transpositional score
                positionScore = POSITIONAL_WEIGHT * positionalTableMap[piece[1]][row][col]
                if piece[0]=="w":
                    score+= pieceScore[piece[1]] + positionScore
                elif piece[0]=="b":
                    score-= pieceScore[piece[1]] + positionScore
    return score
```

```
def DepthTwoMinMaxAI(gameState,validMoves):
    turnSign = 1 if gameState.whiteToMove else -1
    MinMaxScore = CHECKMATE  # init the worst possible score
    bestAIMove = None
    random.shuffle(validMoves)
    for aiMove in validMoves:
        gameState.makeMove(aiMove)
        # FIND OPPONENTS MAX SCORE
        oppMoves = gameState.getValidMoves()
        oppMaxScore = -CHECKMATE
        for oppMove in oppMoves:
            gameState.makeMove(oppMove)
            if gameState.checkmate:
                score = CHECKMATE
            elif gameState.stalemate:
                score = STALEMATE
            else:
                score = -turnSign * ScoreBoard(gameState.board)
            if (score > oppMaxScore):
                oppMaxScore = score
            gameState.undoMove()

        # FIND YOUR MIN SCORE
        if oppMaxScore < MinMaxScore:
            MinMaxScore = oppMaxScore
            bestAIMove = aiMove
        gameState.undoMove()
    return bestAIMove

def MinMaxAI(gameState,validMoves):
    global nextMove
    nextMove = None
    random.shuffle(validMoves)
    RecursiveMinMax(gameState,validMoves,DEPTH,gameState.whiteToMove)
    return nextMove
```

```python
def NegaMaxAI(gameState,validMoves):
    global nextMove, counter
    nextMove = None
    random.shuffle(validMoves)
    # counter =0
    RecursiveNegaMax(gameState, validMoves, DEPTH, (1 if gameState.whiteToMove else -1))
    # print(counter)
    return nextMove


def RecursiveNegaMax(gameState,validMoves,depth,turnMultiplier):
    global nextMove, counter
    # counter +=1
    if depth == 0:
        return turnMultiplier * ScoreBoard(gameState)
    maxScore = -CHECKMATE # init with the worst possible value
    for move in validMoves:
        gameState.makeMove(move)
        nextMoves = gameState.getValidMoves()
        score = -1 *  RecursiveNegaMax(gameState, nextMoves, depth - 1,-1*turnMultiplier)
        if score > maxScore:
            maxScore = score
            if depth == DEPTH:
                nextMove = move
        gameState.undoMove()
    return maxScore
```

```python
def AlphaBetaPruningAI(gameState,validMoves):
    global nextMove, counter
    nextMove = None
    random.shuffle(validMoves)
    # counter = 0
    RecursiveAlphaBetaPruning(gameState, validMoves,DEPTH,-CHECKMATE,CHECKMATE, (1 if gameState.whiteToMove else -1))
    # print(counter)
    return nextMove


def RecursiveAlphaBetaPruning(gameState,validMoves,depth,alpha,beta,turnMultiplier):
    #alpha is max rn and beta is min score rn
    global nextMove, counter
    # counter +=1
    if depth == 0:
        return turnMultiplier * ScoreBoard(gameState)
    # order moves - implement later for better efficiency
    maxScore = -CHECKMATE # init with the worst possible value
    for move in validMoves:
        gameState.makeMove(move)
        nextMoves = gameState.getValidMoves()
        score = -1 *  RecursiveAlphaBetaPruning(gameState, nextMoves,
                                                depth=depth - 1,
                                                alpha=-beta,
                                                beta=-alpha,
                                                turnMultiplier=-1*turnMultiplier)
        if score > maxScore:
            maxScore = score
            if depth == DEPTH:
                nextMove = move
        gameState.undoMove()
        if maxScore > alpha:
            alpha = maxScore
        if alpha >=beta:
            break
    return maxScore
```

A transpositional table is a data structure used in a chess engine to store previously evaluated positions and their associated scores. Since there are many different paths that can lead to the same position in chess, storing the scores for each possible path can be very memory-intensive. Instead, a transpositional table allows the engine to store the score for a position once and then retrieve it later if the same position is reached again through a different path.

The table is typically implemented as a hash table, which allows for fast lookup and retrieval of previously stored positions. Each entry in the table represents a unique position on the chess board, and includes information such as the position's score, the depth at which it was evaluated, and other relevant data such as the best move found from that position.

By using a transpositional table, the chess engine can avoid evaluating the same position multiple times, which can lead to significant improvements in both speed and accuracy of the search.

The Minimax algorithm is a decision-making algorithm that considers all possible moves in a two-player game and selects the best move that minimizes the maximum loss.

The Nega-Max algorithm is a variation of the Minimax algorithm that simplifies the search process by negating the evaluation function and only considering the maximum of the negated values.

Alpha-beta pruning is a technique used in computer algorithms for gameplaying that reduces the number of nodes searched by discarding those that cannot affect the final decision.

RESULTS

1) Basic chess playing abilities achieved.
2) Game prediction now possible.
3) Basic opening analysis.
4) Basic game annotation.

FUTURE SCOPE

1. Educational tool: A weak chess engine could be used as an educational tool for beginners or casual players looking to improve their game. It could provide helpful hints and tips, and allow users to practice against an opponent that is more forgiving than a strong engine.
2. Research and development: A weak chess engine could be used as a testbed for new AI algorithms and techniques. Researchers could experiment with different approaches and see how they perform against a relatively simple opponent.
3. Gaming and entertainment: A weak chess engine could be used as an opponent in video games or other entertainment applications. It could provide a fun and challenging opponent that is accessible to players of all skill levels.
4. Personal assistant: A weak chess engine could be integrated into personal digital assistants or chatbots, providing users with a fun and interactive way to pass the time or learn a new skill.
5. Data analysis: A weak chess engine could be used as a tool for data analysis, particularly in the areas of pattern recognition and machine learning. Its ability to recognize patterns and make predictions based on those patterns could be useful in a wide range of applications.

Overall, while a weak chess engine may not be as powerful as a strong one, it still has the potential to be a useful tool in a variety of contexts, both educational and commercial.

Project Repository

https://github.com/hemant7199/Chess_Engine