Building and Analyzing a Chess AI

Ekin Ercetin

April 30, 2022

Abstract

Building a chess AI using search algorithms like Minimax and Alpha Beta pruning has been a famous topic for a lot of researchers over the past few years. Research into AI in chess has produced several contributions to AI. Famous chess engines like Stockfish are great examples of how certain search algorithms can be used to create great programs that use AI. There are 4 total steps beginners can follow to build a chess engine like Stockfish: board representation, board evaluation, move selection, and testing.

1 Introduction

This report will talk about creating a simple chess AI and understanding how search algorithms like Minimax search and Alpha Beta pruning work to solve a two player turn-based, decision making problem. This report will also delve into testing the chess AI in several ways to analyze the smartness of the AI and determine if it can detect a checkmate and make the right moves to get to that board state.

Chess is a self-contained world that can be explored and it's one of the model organisms for AI researchers because it illustrates many of the phenomena that's seen in AI. Chess engines have impacted the game of chess in a positive manner. The influence of AI on chess is an overall drastic improvement, and further research and advancements are yet to be made for challenging the professional players. Which means there are many ways chess engines can be improved upon. For example, opening theory, opening and closing move databases and datasets will allow developers to further their implementations of chess AI.

Chess AIs are also very interesting because advanced level chess engines are able to tell what level of a player it's playing against and what is a good move for a player at that level. Chess AIs can also tell where, when and how many mistakes a player has made and what that player could have done differently to change the outcome of a game.

This project will help us understand the basics of designing and programming a chess AI and learning the simple theories and tricks to implementing a chess engine.

For this project, to build a chess AI we have followed the steps given in the article named "Let's Create a Chess AI" [Gai20]. But we modified the steps and the given code to create a chess class in python and add functions to that class that deal with board representation, board evaluation and, move selection. The estimated state-space complexity of the algorithm is 10^{46} and the game tree complexity is 10^{123}. Average branching factor is 35 and the average game length is 80 ply.

2 Literature Review

2.1 History and Importance of Chess AI

Chess is a strategy game that has been around for centuries. It's also called a "game of intelligence" which makes it an excellent domain for conducting AI research. It has a rich problem-solving domain, people are able to record chess competitions to monitor progress and use it to develop algorithms, and lastly it's an "excellent test bed for uncertainty management schemes - the basis of most expert problem-solving."[LHS+91] [Ber78]

Research into AI in chess has produced several contributions to AI:

1. Chess has demonstrated that sometimes brute-force approaches to solving problems are as effective as the rest. [Gil72]

2. Ideas that have been developed for iterative deepening search, and alpha-beta search are also applicable to other AI domains.

While there has been a great amount of research done for chess AI, there still hasn't been many scientific advances in knowledge-based search algorithms, knowledge representation and acquisition, error analysis, or tool development in this domain in particular. And these are some of the reasons why a computer won't be able to surpass human capabilities when it comes to a chess match. [Has17]

2.2 Famous Chess Engines

Psychological evidence suggests that chess players "search very few positions and base their positional assessments on structural/perceptual patterns learned through experience." Famous chess engines have been developed to be more consistent with the cognitive models. They also have clever board representations which operate on bits, where bit-wise operations reveal information about the board positions very efficiently. "Chess AI: Competing Paradigms for Machine Intelligence" [MPT21] tests and compares two famous chess engines (Stockfish and LCZero) using Plaskett's Puzzle, a famous endgame study.

2.2.1 Stockfish

Stockfish uses alpha-beta pruning where the search terminates early when it reaches a certain ply (turn taken by a player). Stockfish applies two main heuristics to reduce search space: reduction and forward pruning. The evaluation function is hard-coded based on position of the pieces, game phase, and piece activity. [MPT21]

2.2.2 LCZero

LCZero uses Monte Carlo Tree Search to identify best moves and board positioning. The search depth increases with every simulation and it directs future simulations towards the best outcome. The engine has layered residual blocks and it supports endgame tablebases. [MPT21]

2.2.3 Comparison

When both engines are tasked with solving the Plaskett's Puzzle, Stockfish proves superior. It searches through 1.9 billion different positions to identify the best piece position/move. The algorithm works much more efficiently compared to LCZero, and finds the more unpredictable solution. [MPT21]

LCZero has a less efficient algorithm because the algorithm prefers positional advantage over everything which means giving away pieces and material isn't a big deal. That's why the engine's human like approach fails the edge cases like recognizing positional potential of a knight sacrifice.

"After annotating 40 games between Stockfish and LCZero, FIDE Master Bill Jordan concludes that "Stockfish represents calculation" and "LCZero represents intuition"" [MPT21]

2.3 Simple Chess AI Implementation

According to Cornell Computer Science [Uni04], there are eight steps to follow to implement an AI chess algorithm.

2.3.1 Board Representation

The chessboard is represented by an 8 by 8 matrix. Initially all spaces contain a "blank" piece which represents the empty board spaces. Side castling and en-passant capture move availability are checked by flag variables. Each move contains information about piece movements and capturing enemy pieces, which makes reversing a move on the board a simple process.

2.3.2 Using Minimax and Alpha-Beta Pruning

The core of writing a simple chess AI is the minimax search algorithm. Minimax search is used to minimize the opponent's score and maximize the player's own. At each depth the algorithm examines all possible valid moves and calls the static board evaluation function to determine the scores on each leaf of the tree. The algorithm looks ahead multiple moves to determine the optimal score. [Str11]

Later on Alpha-Beta pruning is used to improve the minimax search space. Alpha-Beta pruning basically keeps track of the best and worse moves for each player and avoids searching branches that contain worse results. [FGG+73]

Cornell University also uses a heuristic called the "null move heuristic" where it simply improves the beginning of the search algorithm. Initially instead of using negative and positive infinity for the best and worst move values, this heuristic finds a lower bound on the best moves. This works by letting the opponent play two moves in sequence and computing the score right after.

Lastly Quiescence searching is used for moves like capturing a piece where it affects the score by a lot. Quiescence search is used because the algorithm searches for the positions that don't affect the current positions too much, so that it won't make moves where next move ends in a disaster. For example, the minimax algorithm will chose the move where a knight is captured since it seems to be a great move, but because of that, in the next move the opponent might have the chance to capture the queen. The search only looks at the moves that become available because of the current move.

2.3.3 Static Board Evaluation Function

The evaluation function helps determine how good or how bad the given board positioning is. The function returns a numerical value so that it can compare the value of other board positions with it. The simplest evaluation function returns values looking at only piece possessions. Every piece is given a value, where the order from least to most is pawn, bishop, knight, rook, and queen.

Additional features added to Cornell's chess program include "pawn advancement", "piece mobility", "piece threats", and "piece protects".

Lastly Cornell optimized the board evaluation function using genetic algorithms. They created a module where the computer plays chess tournaments against itself with different evaluation functions. First it generates random evaluation functions, then the evaluation functions get preserved or mutated depending on their performance. After observing the tournament results, Cornell realized that the higher value pieces have higher "mobility and threats" weights while those pieces also have lower "protects" weights.

2.3.4 Opening Move Database

As the years have passed, a lot of opening moves developed by chess masters have been collected into a database. The program uses this database at the start of every game to check if that initial move fits into a given opening strategy. "The database contains over 2000 openings, stored in a binary, allowing for constant time access." [Uni04]

3 Description of Approach: Representation and Algorithms

There are 4 steps to create a chess AI that can make logical moves to beat an opponent.

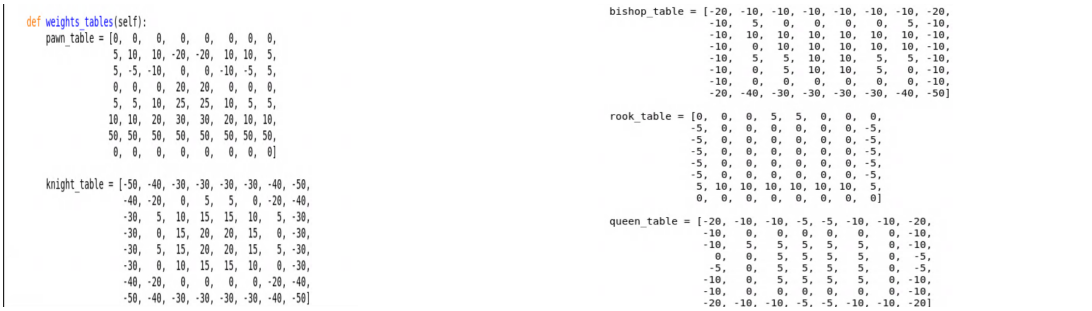
1) Board Representation
This beginning step is one of the most important parts of the AI that will allow the algorithm to understand and evaluate the chess board. First the python-chess library has to be installed and imported into the AIMA code's games.py file, which provides all possible move generation and validation and assigns every possible legal move to each and every chess piece. This is essential for the program to understand the logic behind chess pieces. This library will provide essential functions such as board.pop() which undoes the last move, board.push() which appends a move, or board.ischeckmate() which checks to see if there's a checkmate.

2) Board Evaluation
Board evaluation is essential for the chess AI program to determine the relative value of a position

which is the chances of winning the game. There are some points which will be used while writing the final evaluation function. These important points include: "avoid exchanging one minor piece for three pawns, always have the bishop in pairs, avoid exchanging two minor pieces for a rook and a pawn". After translating these points into equations, we get:

- Bishop > 3 pawns and a knight > 3 pawns
- Bishop > knight
- Bishop + knight > rook + pawn

Before writing the evaluation function, a chess class is created in games.py, which includes an init() function which initializes the chess board. After that an 8x8 matrix is used to represent piece square tables for the evaluation function. Piece square tables will have lower values at non-favorable places and higher values at favorable positions. For example, a queen piece will have higher chances of dominating the board if it's placed in the middle of the board and will be given higher points since chess is all about defending the king piece and dominating the center of the board. We'll then add a function to the class called "weightstables()" to create a total of 6 piece square tables and put them into a list to use them later for calculations.



```
def weights_tables(self):
    pawn_table = [0,  0,  0,  0,  0,  0,  0,  0,
                  5, 10, 10, -20, -20, 10, 10,  5,
                  5, -5, -10,  0,  0, -10, -5,  5,
                  0,  0,  0, 20, 20,  0,  0,  0,
                  5,  5, 10, 25, 25, 10,  5,  5,
                 10, 10, 20, 30, 30, 20, 10, 10,
                 50, 50, 50, 50, 50, 50, 50, 50,
                  0,  0,  0,  0,  0,  0,  0,  0]

    knight_table = [-50, -40, -30, -30, -30, -30, -40, -50,
                    -40, -20,  0,  5,  5,  0, -20, -40,
                    -30,  5, 10, 15, 15, 10,  5, -30,
                    -30,  0, 15, 20, 20, 15,  0, -30,
                    -30,  5, 15, 20, 20, 15,  5, -30,
                    -30,  0, 10, 15, 15, 10,  0, -30,
                    -40, -20,  0,  0,  0,  0, -20, -40,
                    -50, -40, -30, -30, -30, -30, -40, -50]
```

(a) Pawn and Knight weights

```
bishop_table = [-20, -10, -10, -10, -10, -10, -10, -20,
                -10,   5,   0,   0,   0,   0,   5, -10,
                -10,  10,  10,  10,  10,  10,  10, -10,
                -10,   0,  10,  10,  10,  10,  10, -10,
                -10,   5,   5,  10,  10,   5,   5, -10,
                -10,   0,   5,  10,  10,   5,   0, -10,
                -10,   0,   0,   0,   0,   0,   0, -10,
                -20, -40, -30, -30, -30, -30, -40, -50]

rook_table = [0,  0,  0,  5,  5,  0,  0,  0,
              -5,  0,  0,  0,  0,  0,  0, -5,
              -5,  0,  0,  0,  0,  0,  0, -5,
              -5,  0,  0,  0,  0,  0,  0, -5,
              -5,  0,  0,  0,  0,  0,  0, -5,
              -5,  0,  0,  0,  0,  0,  0, -5,
               5, 10, 10, 10, 10, 10, 10,  5,
               0,  0,  0,  0,  0,  0,  0,  0]

queen_table = [-20, -10, -10, -5, -5, -10, -10, -20,
               -10,   0,   0,  0,  0,   0,   0, -10,
               -10,   5,   5,  5,  5,   5,   0, -10,
                 0,   0,   5,  5,  5,   5,   0,  -5,
                -5,   0,   5,  5,  5,   5,   0,  -5,
               -10,   0,   5,  5,  5,   5,   0, -10,
               -10,   0,   0,  0,  0,   0,   0, -10,
               -20, -10, -10, -5, -5, -10, -10, -20]
```

(b) Bishop, Rook and Queen weights

Figure 1: Weight Tables

Then we create a function called "bstate()" which takes in the chess board, to check the state of the game. For example if there is a checkmate, the function decides who won and returns -1 or 1 accordingly.

```
king_table = [20,  30,  10,   0,   0,  10,  30,  20,
              20,  20,   0,   0,   0,   0,  20,  20,
             -10, -20, -20, -20, -20, -20, -20, -10,
             -20, -30, -30, -40, -40, -30, -30, -20,
             -30, -40, -40, -50, -50, -40, -40, -30,
             -30, -40, -40, -50, -50, -40, -40, -30,
             -30, -40, -40, -50, -50, -40, -40, -30,
             -30, -40, -40, -50, -50, -40, -40, -30]
```

(a) King weights

```
def b_state(self, board):#1
    if board.is_checkmate():
        if board.turn:
            return -1 #white wins
        else:
            return 1 #black wins
    if board.is_stalemate():
        return 0
    if board.is_insufficient_material():
        return 0
    else:
        return 2
```

(b) bstate

Figure 2: King weights table and bstate() function

Then two other functions are created to count the total number of black and white pieces and to calculate the material and individual piece scores. The second function returns the summation of the individual scores and material scores for white. When it's black's turn the function negates the value and returns it. Lastly the evaluation function is created which checks the state of the board and returns -9999 if it's white's turn or 9999 if it's black's turn.

```
w_table = [pawn_table, knight_table, bishop_table, rook_table, queen_table, king_table]
return w_table
```

Figure 3: Weights list

```
def total_pieces(self, board):
    wp = len(board.pieces(chess.PAWN, chess.WHITE)) #white pawns
    bp = len(board.pieces(chess.PAWN, chess.BLACK)) #black pawns
    wk = len(board.pieces(chess.KNIGHT, chess.WHITE)) #white knights
    bk = len(board.pieces(chess.KNIGHT, chess.BLACK)) #black knights
    wb = len(board.pieces(chess.BISHOP, chess.WHITE)) #white bishops
    bb = len(board.pieces(chess.BISHOP, chess.BLACK)) #black bishops
    wr = len(board.pieces(chess.ROOK, chess.WHITE)) #white rooks
    br = len(board.pieces(chess.ROOK, chess.BLACK)) #black rooks
    wq = len(board.pieces(chess.QUEEN, chess.WHITE)) #white queens
    bq = len(board.pieces(chess.QUEEN, chess.BLACK)) #black queens

    piece_table = [wp, bp, wk, bk, wb, bb, wr, br, wq, bq]
    return piece_table
```

(a) totalpieces()

```
def score_eval(self, board):
    #sum of all piece's weights multiplied by the difference between the number of that piece between white
    piece_table = self.total_pieces(board)
    wp = piece_table[0]
    bp = piece_table[1]
    wk = piece_table[2]
    bk = piece_table[3]
    wb = piece_table[4]
    bb = piece_table[5]
    wr = piece_table[6]
    br = piece_table[7]
    wq = piece_table[8]
    bq = piece_table[9]

    material_score = 100*(wp-bp) + 320*(wk - bk) + 330*(wb-bb) + 500*(wr-br) + 900*(wq-bq)

    #sum of piece-square values of positions where the piece is present at that instance
    w_list = self.weights_tables()
    p_table = w_list[0]
    k_table = w_list[1]
    b_table = w_list[2]
    r_table = w_list[3]
    q_table = w_list[4]
    king_table = w_list[5]

    pawnsq = sum([p_table[i] for i in board.pieces(chess.PAWN, chess.WHITE)])
    pawnsq = pawnsq + sum([-p_table[chess.square_mirror(i)] for i in board.pieces(chess.PAWN, chess.BLACK)])
```

(b) scoreeval()

Figure 4: totalpieces() and scoreeval() functions

```
def eval_f(self, board):
    state = self.b_state(board)
    if state == -1:
        return -9999
    if state == 1:
        return 9999
    if state == 0:
        print("stalemate or insufficient material")
        return 0
    else:
        return self.score_eval(board)
```

Figure 5: Evaluation function

3) Move Selection

After representing the board using the chess library and evaluating the board using multiple functions, the program needs a way to decide which move is the best move to make given the current board positioning and status. And the best way to select the best legal move is to use the Negamax implementation of the Minimax Algorithm and then optimize it using Alpha-Beta pruning to reduce execution time. At each step the Minimax Algorithm will assume that player 1 maximizes its chances of winning, and in the next turn player 2 will try to minimize the chances of winning. Negamax implementation of the Minimax Algorithm is the same as Minimax but instead of implementing the algorithm using two separate subroutines for the max and min players, it passes on the negated score so one player's loss is equal to another player's gain.

Next alpha-beta pruning is used to optimize the Negamax function so that the unnecessary iterations can be eliminated.

Lastly Quiescence search is used to evaluate the positions where there are no winning tactical moves to be made. For example, there might be a situation where the last move considered will be a queen capturing a pawn but after searching one move deeper the queen is captured by another pawn. So Quiescence search makes sure that the program is only evaluating quiet positions and is avoiding the horizon effect caused by depth limitation.

```python
def alphabeta(self, alpha, beta, depthleft, board):
    bestscore = -9999
    if (depthleft == 0):
        return self.quiesce(alpha, beta, board)
    for move in board.legal_moves:
        board.push(move)
        score = -self.alphabeta(-beta, -alpha, depthleft-1, board)
        board.pop()
        if (score >= beta):
            return score
        if (score > bestscore):
            bestscore = score
        if (score > alpha):
            alpha = score
    return bestscore
```

(a) Alpha Beta pruning

```python
def quiesce(self, alpha, beta, board):
    stand_pat = self.eval_f(board)
    if (stand_pat >= beta):
        return beta
    if (alpha < stand_pat):
        alpha = stand_pat
    for move in board.legal_moves:
        if board.is_capture(move):
            board.push(move)
            score = -self.quiesce(-beta, -alpha, board)
            board.pop()
            if score >= beta:
                return beta
            if score > alpha:
                alpha = score
    return alpha
```

(b) Quiesce search

Figure 6: Alpha Beta and Quiesce functions

```python
def negamax(self, depth, board):
    bestMove = chess.Move.null()
    bestValue = -99999
    alpha = -100000
    beta = 100000
    for move in board.legal_moves:
        board.push(move)
        boardValue = -self.alphabeta(-beta, -alpha, depth-1, board)
        if boardValue > bestValue:
            bestValue = boardValue
            bestMove = move
        if (boardValue > alpha):
            alpha = boardValue
        board.pop()
    return bestMove
```

Figure 7: Negamax algorithm

4 Experiments and Results

To test the AI we wrote 4 types of test cases. All of the test cases contain a test chess match starting from different board positions. The goal of the first two experiments is to see if the chess AI can foresee a checkmate that's a couple of moves away and make the right move to reach the goal state. The initial state of these two tests will be "1 move required for the white player (AI) to win" and "2 moves required for the white player (AI) to win". The third experiment plays a game of chess with a real player from the start using the input() function. The last experiment will make two AIs play against each other to measure the lower boundary of turns until one AI wins or draws.

Test case 1: One move until checkmate
For this experiment we used Fool's Mate which is the fastest checkmate possible in chess. In order for the Fool's Mate to be performed, white must move one of their g and f pawns up two or one squares in the first two consecutive moves.

Board state before AI makes its move:

```python
e4 = chess.Move.from_uci("e2e4")
g5 = chess.Move.from_uci("g7g5")
bc3 = chess.Move.from_uci("b1d3")
f5 = chess.Move.from_uci("f7f5")

board.push(e4)
board.push(g5)
board.push(bc3)
board.push(f5)

print(board)
mov = test.negamax(3, board)
print(mov)
board.push(mov)
```

Figure 8: 1 move until checkmate

Board state after AI makes its move:

```
PASS
r n b q k b n r
p p p p p . . p
. . . . . . . .
. . . . . p p .
. . . . P . . .
. . . N . . . .
P P P P . P P P
R . B Q K B N R

"d1h5"
```

Figure 9: Board state of 1 move until checkmate

Test case 2: Two moves until checkmate
For this experiment we used Fool's Mate again but changed the initial state by starting the AI two moves away from checkmate.

Board state before AI makes its move:

```
#white move
e4 = chess.Move.from_uci("e2e4")
board.push(e4)
#black move
g5 = chess.Move.from_uci("g7g5")
board.push(g5)
#AI move for white
mov = test.negamax(3, board)
print(mov)
board.push(mov)
#black move
f5 = chess.Move.from_uci("f7f5")
board.push(f5)
print(board)
#AI move for white
mov = test.negamax(3, board)
print(mov)
board.push(mov)
```

(a) 2 moves until checkmate

```
AI PLAYS: g1f3
r n b q k b n r
p p p p p . . p
. . . . . . . .
. . . . . p p .
. . . . P . . .
. . . . . N . .
P P P P . P P P
R N B Q K B . R
AI PLAYS: e4f5
```

(b) Board state

Figure 10: 2 moves

Board states until AI wins:

```
#AI move for white
mov = test.negamax(3, board)
board.push(mov)
#black move
a7 = chess.Move.from_uci("a7a5")
board.push(a7)
print(mov)
#AI move for white
mov = test.negamax(3, board)
board.push(mov)
#black move
b7 = chess.Move.from_uci("b7b5")
board.push(b7)
print(mov)
#AI move for white
mov = test.negamax(3, board)
board.push(mov)
print(board)
```

(a) 2 moves until checkmate continuation

```
g1f3
e4f5
f3g5
d1h5
r n b q k b n r
. . p p p . . p
. . . . . . . .
p p . . . P N Q
. . . . . . . .
. . . . . . . .
P P P P . P P P
R N B . K B . R
```

(b) Board state

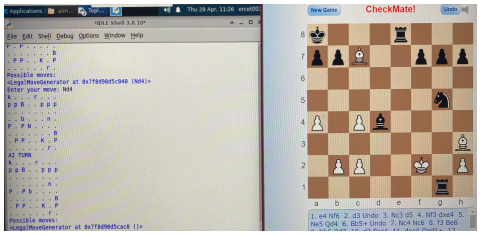
Figure 11: 2 moves continuation

Test case 3: Human vs AI:
For this experiment we used the input() function inside a for loop to prompt the player to play a continuous game of chess with an AI. Two players played against the chess AI program and the AI was able to win against the beginner level player while the intermediate 2 level player was able to outsmart the AI.

Beginner level vs AI final board state:

```
HUMAN VS AI
for i in range(40):
    print("Possible moves:")
    print(board.legal_moves)
    player_move = input("Enter your move: ")
    board.push_san(player_move)
    print(board)
    print("AI TURN")
    mov = test.negamax(3, board)
    board.push(mov)
    print(board)
```

(a) Human vs AI code



(b) Human vs AI result

Figure 12: Human vs AI

Test case 4: AI vs AI

Lastly for this experiment we used a for loop and two AI prompts to make the AIs play against each other. Maximum number of turns was limited to 40 since the AIs started taking a long time to decide which move to make after 20 turns.

Final state of the board after 40 turns:

```
 AI VS AI
count = 1
for i in range(40):
    print("AI 1 TURN")
    print(count)
    mov = test.negamax(3, board)
    board.push(mov)
    print(board)
    print("AI 2 TURN")
    print(count)
    count +=1
    mov2 = test.negamax(3, board)
    board.push(mov2)
    print(board)

AI 2 TURN
40
. . . . . k .
. . p . . p . p
. . . . r . . .
. . . . . . . .
. . . B . . P .
. P . . . . . .
r . . . . P . P
. . b . . . K .
```

Figure 13: AI vs AI

5 Analysis of Results

Analysis of test case 1:

Looking at the results of test case 1, it is clear that the AI knows which move to make when it has one move until checkmate. After running the program multiple times and using Scholar's Mate as another experiment, the AI still pushes the right move onto the board and wins the game with a checkmate. This shows that the weights for capturing the King where the King has no way of escaping is very high compared to weights of other legal moves.

Analysis of test case 2:

Looking at the results of test case 2, it can be observed that the AI fails to choose the right set of moves that would allow the chess game to be over in less than 2 moves. Instead of moving any pawn the AI moves the knight to a position where it blocks the queen from getting the chance to move and capture the king. However when it continues to play the game, the AI wins it with a checkmate in 4 moves. This is 2 moves more than the ideal Fool's Mate checkmate. This shows that the algorithm prioritizes center control and knight positioning, and capturing pieces rather than ending a game as fast as possible.

Analysis of test case 3:
Looking at the results of test case 3, it can be observed that the chess AI is smart enough to beat a beginner level player in less than 30 moves. Compared to its game against the intermediate 2 level player, the AI also took less time on its turns and was around 2-3 moves ahead of the beginner level player. This shows that the chess engine's smartness level is more than an intermediate level player but not more than an intermediate 2 level player.

Analysis of test case 4:
Looking at the results and running the AI vs AI program multiple times, we come to the conclusion that the AIs lose similar pieces and end up with around the same number of total pieces by the end of 40 turns. In the experiment above, white AI had a total of 3 pawns, 2 rooks, and a bishop while black AI had 4 pawns and a bishop. Looking at the board state it's not too far fetched to say that white AI would be able to win the game after a few more turns since it still has both of its rook pieces.

6 Conclusion

In conclusion, designing and building a chess AI consists of 4 steps. Board representation, board evaluation, move selection, and testing. The chessboard is represented by using the python-chess library which provides useful functions like board.pop() which can reverse a move and board.ischeckmate() which can check if the board state is a checkmate. Board evaluation is done by writing a python chess class and using 8x8 matrices to represent the piece square tables for the evaluation tables. Then lastly 5 functions are added to get our evaluation score: weightstables(), bstate(), totalpieces(), scoreeval(), and evalf(). For selecting the best moves, the Negamax algorithm with Alpha Beta pruning is used. Lastly Quiescence search is used to evaluate the positions where there are no winning tactical moves to be made. To test the chess engine, we used 4 test cases to determine how smart and efficient our chess AI is. First test case consisted of a chess match where there was only 1 move left until checkmate. For the first test, the chess engine was able to make the right move in a reasonable amount of time. The second test case consisted of a chess match where there were 2 moves left until checkmate. For the second test, the chess engine wasn't able to make the right move right away but it was able to finish the game in 4 moves instead of 2. Third test case consisted of a chess game between a player and the AI. A beginner level player wasn't able to beat the AI but an intermediate 2 level player was. Lastly the final test case consisted of a chess game between an AI and another AI, which ended in a draw due to long waiting times for each turn.

In the future we would like to continue adding more functionalities to our chess engine for it to be able to beat higher level players. For example, we would like to use a dataset or a database that consists of famous opening moves and implement that in our chess class. For testing the AI we would like to have more test cases where we look into other checkmate situations to see if the AI makes the right move.

References

[Ber78] Hans J Berliner. A chronology of computer chess and its literature. *Artificial Intelligence*, 10(2):201–214, 1978.

[FGG+73] Samuel H Fuller, John G Gaschnig, JJ Gillogly, et al. *Analysis of the alpha-beta pruning algorithm*. Department of Computer Science, Carnegie-Mellon University, 1973.

[Gai20] Ansh Gaikwad. Let's create a chess ai, Oct 2020.

[Gil72] James J Gillogly. The technology chess program. *Artificial Intelligence*, 3:145–163, 1972.

[Has17] Demis Hassabis. Artificial intelligence: chess match of the century. *Nature*, 544(7651):413–414, 2017.

[LHS+91] Robert Levinson, Feng-hsiung Hsu, Jonathan Schaeffer, T Anthony Marsland, and David E Wilkins. The role of chess in artificial intelligence research. *ICGA Journal*, 14(3):153–161, 1991.

[MPT21] Shiva Maharaj, Nick Polson, and Alex Turk. Chess ai: Competing paradigms for machine intelligence. *arXiv preprint arXiv:2109.11602*, 2021.

[Str11] Glenn Strong. The minimax algorithm. *Trinity College Dublin*, 2011.

[Uni04] Cornell University. Ai chess algorithms, 2004.