A PROJECT REPORT

On

"CHESS ENGINE IN PYTHON"

Submitted to

KIIT Deemed to be University

In Partial Fulfilment of the Requirement for the Award of

BACHELOR'S DEGREE IN
INFORMATION TECHNOLOGY

BY

ROHIT GHOSH 1929038

UNDER THE GUIDANCE OF
DR. ABHISHEK RAY

SCHOOL OF COMPUTER ENGINEERING
KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY
BHUBANESWAR, ODISHA - 751024
May 2023

A PROJECT REPORT
On

"CHESS ENGINE IN PYTHON"

Submitted to
KIIT Deemed to be University

In Partial Fulfilment of the Requirement for the Award of

BACHELOR'S DEGREE IN
INFORMATION TECHNOLOGY

BY

ROHIT GHOSH 1929038

UNDER THE GUIDANCE OF
DR. ABHISHEK RAY

SCHOOL OF COMPUTER ENGINEERING
KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY
BHUBANESWAE, ODISHA -751024
May 2023

KIIT Deemed to be University

School of Computer Engineering
Bhubaneswar, ODISHA 751024

CERTIFICATE

This is certify that the project entitled

"CHESS ENGINE IN PYTHON"

Submitted by
ROHIT GHOSH 1929038

Is a record of bonafide work carried out by them, in the partial fulfilment of the requirement for the award of Degree of Bachelor of Engineering (Computer Science & Engineering OR Information Technology) at KIIT Deemed to be university, Bhubaneswar. This work is done during year 2022-2023, under our guidance.

Date: 30/04/2023

(Dr. Abhishek Ray)
Project Guide

Acknowledgements

I am profoundly grateful to **DR ABHISHEK RAY** of **KIIT UNIVERSITY** for his expert guidance and continuous encouragement throughout to see that this project rights its target since its commencement to its completion. .....

ROHIT GHOSH

ABSTRACT

The aim of this project is to develop a sophisticated chess engine in Python that can play chess at a high level. The chess engine will be developed using various algorithms, such as the minimax algorithm, alpha-beta pruning, and various heuristics. The engine will be able to analyse different chess positions, generate possible moves, and evaluate the best move based on the current state of the board.

In addition, the engine will be designed to use a graphical user interface, allowing users to play against the engine and observe the engine's thought process. The performance of the chess engine will be evaluated based on its ability to play against human players, as well as its ability to defeat other chess engines.

Contents

1	Introduction			7
2	Basic Concepts			8
	2.1	Driver File		8
	2.2	Chess Engine Basic		9
	2.3	Chess Engine Advanced		10
	2.4	Chess Bot		12
3	Problem Statement / Requirement Specifications			14
	3.1	System Design		14
		3.3.1	Design Constraints	14
4	Implementation			15
	4.1	Methodology		15
	4.2	Screenshots		15
5	Conclusion and Future Scope			16
	6.1	Conclusion		16
	6.2	Future Scope		16
References				17
Individual Contribution				18
Plagiarism Report				19

List of Figures

4.2 FIGURE 1 15

4.2 FIGURE 2 15

Chapter 1

Introduction

Chess is one of the oldest and most popular board games in the world, played by millions of people worldwide. Chess engines are computer programs that can play chess at a high level, often surpassing even the best human players. In this project, we will be creating our own chess engine using Python, a popular programming language.

There are several reasons why creating a chess engine is a valuable project. Firstly, it allows us to explore the field of artificial intelligence and machine learning. Chess engines use a variety of techniques, such as brute-force search algorithms, heuristics, and machine learning, to make decisions about which moves to play. By building our own chess engine, we can learn about these techniques and how they can be applied in other domains.

This project will cover a wide range of concepts in programming, including:

1. Data structures: We will need to represent the chess board and game state using appropriate data structures, such as arrays, lists, and dictionaries.
2. Algorithms: We will be using a variety of search algorithms, such as minimax and alpha-beta pruning, to find the best move for our engine.
3. Object-oriented design: We will be designing our engine using object-oriented principles, creating classes for pieces, the board, and the game.

Overall, this project will be an exciting and challenging endeavour that will provide us with a deeper understanding of programming and artificial intelligence while contributing to the open-source community.

Chapter 2
Basic Concepts

This section contains the explanation of the various parts of the process of building the chess engine.

2.1 Driver File:-

I have created a driver file called ChessMain.py which contains the functions necessary to handle user input and draw the game visuals. It works in tandem with the engine file to drive the chess game via python.
The program begins by importing several modules and packages, including `**sys**`, `**pygame**`, `**ChessEngine**`, `**ChessBot**`, and `**config**`. The program then initializes the Pygame library with `**p.init()**`, sets some constants such as `**BOARD_WIDTH**`, `**BOARD_HEIGHT**`, `**DIMENTION**`, and `**SQ_SIZE**`, and loads the images for each chess piece using `**loadImages()**` function.
The main function `**main()**` is then called, which handles the main game loop. The function begins by initializing the game state with a `**GameState()**` object from the `**ChessEngine**` module. It then sets some variables such as `**validMoves**`, `**moveMade**`, `**animate**`, `**sqSelected**`, `**playerClicks**`, `**playerOne**`, `**playerTwo**`, and `**gameOver**`. These variables are used to keep track of the current state of the game, including which player is currently taking a turn and which moves are valid.
The game loop runs until the user quits the program by clicking the close button on the window. During each iteration of the loop, the program handles user input by calling `**p.event.get()**`, which returns a list of events that have occurred since the last time the function was called. The program checks each event and takes appropriate actions based on the event type.
If the event is a `**QUIT**` event, the program sets `**running**` to `**False**`, which exits the game loop and quits the program. If the event is a `**MOUSEBUTTONDOWN**` event, the program checks whether the game is over and whether it is currently the human player's turn. If these conditions are met, the program gets the position of the mouse cursor and calculates the row and column of the square that was clicked. It then checks whether the square clicked is a valid move and updates the game state accordingly. If the event is a `**KEYDOWN**` event, the program checks whether the `**z**` or `**r**` key was pressed. If `**z**` was pressed, the program undoes the last move. If `**r**` was pressed, the program resets the game state to the initial state.
If it is not the human player's turn, the program calls the **findBestMoveMinMax()**` function from the `**ChessBot**` module to find the best move for the AI player. If no valid move can be found, the program sets `**gameOver**` to `**True**` and the game loop ends.

Overall, the program uses a modular design with separate functions for handling user input, managing the game state, validating moves, and rendering the game board. This makes the code easier to read and maintain, and allows for easy modification and extension of the game's functionality.

2.2 Chess Engine Basic: -

I have used 2 different algorithms to make an AI. The first one is the brute force algorithm used in the ChessEngine.py file and the optimized algorithm in the ChessEngineAdv.py to calculate the next moves for the computer.

The ChessEngine.py works in the following way: -

The code has several instance variables like **board**, **whiteToMove**, **moveLog**, **whiteKingLocation**, **blackKingLocation**, **checkMate**, **staleMate**, **enPassantPossible**, **currentCastlingRights** and **castleRightsLog** that are used to store the current state of the game.

board is a 2D list that stores the current state of the chessboard where each element of the list is a 2 character string representing the color and the piece present on the corresponding position of the board. If a position is empty, '--' is stored in the corresponding cell of the 2D list.

whiteToMove is a boolean variable that indicates if it's white's turn to move.**moveLog** is a list of all the moves made in the game so far.

whiteKingLocation and **blackKingLocation** store the positions of the white and black kings respectively.

checkMate and **staleMate** are boolean variables that indicate if the game is in a checkmate or a stalemate situation respectively.

enPassantPossible is a tuple that stores the position of a pawn that can be captured via an en-passant move.

currentCastlingRights is an instance of the **CastleRights** class that stores the current castling rights of the game.

castleRightsLog is a list that stores all the castling rights of the game so far.

The class **GameState** has several methods that are responsible for managing the state of the game.

__init__ initializes all the instance variables of the **GameState** class.

makeMove is responsible for making a move on the chessboard and updating the state of the game accordingly. It takes a **Move** object as an argument and modifies the **board**, **whiteKingLocation**, **blackKingLocation**, **enPassantPossible**, **currentCastlingRights**, **castleRightsLog**, and **whiteToMove** variables.

undoMove is responsible for undoing the last move made in the game. It modifies the **board**, **whiteKingLocation**, **blackKingLocation**, **enPassantPossible**, **currentCastlingRights**, **castleRightsLog**, and **whiteToMove** variables.

getValidMoves is responsible for returning a list of all the valid moves that a piece can make given a position on the board.

updateCastlingRights is responsible for updating the castling rights of the game after a move has been made.
getKingMoves, **getQueenMoves**, **getBishopMoves**, **getKnightMoves**, **getRookMoves**, and **getPawnMoves** are responsible for returning a list of all the valid moves that a piece of the corresponding type can make given a position on the board.

2.3 Chess Engine Advanced:-

The difference in the advanced chess engine is that the **`GameState`** class has been further implemented with methods to handle en-passant and pawn-promotion moves in the `makeMove` method.

The **`enPassantPossible`** attribute has been added to the **`GameState`** class which is initialized as an empty tuple. If a pawn moves two squares from its starting position and lands beside an opponent's pawn on the fifth rank, the opponent's pawn can capture the first pawn as if it had moved only one square forward. The **`enPassantPossible`** attribute keeps track of the square where an en-passant capture can be made in the next move.

The **`makeMove`** method has been updated to handle en-passant captures. If a move is an en-passant capture, then the **`enPassant`** attribute of the `Move` object will be set to True. In that case, the method captures the opponent's pawn, which is on the same row as the pawn being moved, and located beside the pawn's destination.

If a pawn move results in it reaching the opposite end of the board, then pawn promotion occurs. The user is prompted to choose a piece to which the pawn should be promoted to. If the current player is an AI player, the piece is automatically promoted to a queen.

There is a type of move called **`pin`** in Chess that may force a piece in front of the king to not move if there is a piece of the opponent that places the king in check. To handle this, I have implemented a new function which handles this dilemma.
The method first initializes empty lists 'pins' and 'checks', and a boolean variable 'inCheck' to False. It then determines the color of the enemy and ally pieces and the starting row and column of the current player's King.

The method then iterates over a set of eight possible directions, which represent the directions a piece can move from the King's location, and for each direction, it checks if there are any pieces that are pinning or checking the King. If an ally piece is found, the method checks if there is only one ally piece in that direction, in which case it is a possible pin. If an enemy piece is found, the method checks if it can attack the King in that direction, and if it can, it adds the attacking piece's position to the 'checks' list. If a possible pin is found, it adds the pin's position and direction to the 'pins' list.
The method then checks for Knight checks by checking if there is a Knight piece attacking the King, and if there is, it adds the Knight's position to the 'checks' list.
Finally, the method returns a tuple of three values: the boolean 'inCheck' indicating whether the King is in check, the 'pins' list indicating any pieces that are pinned, and the 'checks' list indicating any pieces that are checking the King.
Each movement of the pieces now check for pins and check automatically to prevent the problem.

One more functions is added to check that the king is available to move in a direction that is not under attack by any opposing piece preventing check or checkmate.
The method takes two arguments, **r** and **c**, which represent the row and column of a square on the chessboard, and returns a boolean indicating whether that square is under attack by an opposing piece.
The method uses an optimized algorithm to determine if the square is under attack. It first determines the color of the player whose turn it is (**allyColor**) and the color of the opponent (**enemyColor**). It then defines a list of eight directions that pieces can move in (**directions**). For each direction, it iterates over each square in that direction, starting from the square one square away from the square being checked and moving outward.
If the square being checked is on the board and contains an opponent's piece, the code checks the type of the piece and the direction it is attacking from. If the piece is a rook or bishop attacking orthogonally or diagonally, respectively, or a queen or king attacking in any direction, the method returns **True**, indicating that the square is under attack. If the piece is a pawn attacking one square diagonally, the code checks the direction of the attack based on the opponent's color and returns **True** if the attack is in the correct direction. If the square being

checked contains a piece of the player whose turn it is, the method breaks out of the loop for that direction.
After checking all eight directions, the method checks for knight attacks by iterating over a list of knight moves and checking the square one square away from the square being checked in each direction. If the square being checked is on the board and contains an opponent's knight, the method returns **True**, indicating that the square is under attack.
If no attacks are detected, the method returns **False**, indicating that the square is not under attack.

2.3:- Chess Bot:-

I have designed a bot that uses material score of pieces in order to move and capture opponent pieces. The **pieceScore** dictionary assigns a score to each type of piece, with the King given a score of 0, since it cannot be captured. The Knight, Bishop, Queen, and Rook are assigned scores of 3, 3, 9, and 5, respectively.
The **knightScores**, **bishopScores**, **queenScores**, and **rookScores** matrices assign a score to each square on the board for the corresponding piece. These scores are based on the positional strength of the pieces. For example, the Bishop is generally stronger when it has open diagonals to attack along, so the score is higher for squares near the centre of the board.
The **whitePawnScores** and **blackPawnScores** matrices assign a score to each square on the board for white and black pawns, respectively. The scores are based on the pawn structure and the potential for the pawns to advance and control space.

The program also uses the following functions are used to evaluate the next move that the AI will make based on the current state of the game:

1. **findRandomMove(validMoves)**: This function is used to return a random move from the valid moves available at the current game state.
2. **findBestMove(gs, validMoves)**: This function is used to find the best move from the list of valid moves by analysing the game state and possible future moves. It uses the **gs** (game state) and **validMoves** parameters to determine the best move. The algorithm uses a nested for loop to analyse each possible move, and then returns the best move found.
3. **findBestMoveMinMax (gs, validMoves)**: This function is used to call **findMoveNegaMaxAlphaBetaPruning(gs, validMoves,**

DEPTH, -CHECKMATE, CHECKMATE, 1 if gs.whiteToMove else -1) to evaluate the best move for the current game state. It also shuffles the valid moves list to make the selection process more random.

4. **findMoveMinMax (gs, validMoves, depth, whiteToMove)**: This function is used by **findBestMoveMinMax(gs, validMoves)** to recursively evaluate the score of a given move based on the material of the pieces on the board. It uses the **gs** (game state), **validMoves**, **depth**, and **whiteToMove** parameters to determine the score of the move.
5. **findMoveNegaMax (gs, validMoves, depth, turnMultiplier)**: This function is used by **findBestMoveMinMax(gs, validMoves)** to evaluate the best move for the current game state using the NegaMax algorithm. It uses the **gs** (game state), **validMoves**, **depth**, and **turnMultiplier** parameters to determine the best move.
6. **findMoveNegaMaxAlphaBetaPruning (gs, validMoves, depth, alpha, beta, turnMultiplier)**: This function is used by **findBestMoveMinMax (gs, validMoves)** to evaluate the best move for the current game state using the NegaMax algorithm with Alpha-Beta pruning. It uses the **gs** (game state), **validMoves**, **depth**, **alpha**, **beta**, and **turnMultiplier** parameters to determine the best move. The algorithm uses move ordering to traverse the better moves first, which leads to more pruning and a more optimized algorithm.

The code also uses some global constants, such as **`DEPTH`**, **`CHECKMATE`**, and **`STALEMATE`**, to determine various aspects of the game state and moves.

Chapter 3

Problem Statement / Requirement Specifications

The problem statement for this project was :-

Design and implement a chess engine in Python that allows users to play chess against an AI opponent, with the AI making strategic moves based on various algorithms such as Minimax, Negamax, and Alpha-Beta pruning. The chess engine should be able to provide a user-friendly interface for playing chess

3.1 System Design

3.1.1 Design Constraints

The entire project was done on python programming language. In order to successfully run the project, the computer must have the latest version of python interpreter on it to run.

Chapter 4

Implementation

4.1 Methodology

To implement the entire project, I have used Python programming language. To use the main part , that is the AI against the player, I have used 2 algorithms.

1.The Brute Force Algorithm. It selects the best made move possible from all list of valid moves. It uses the minmax algorithm along with the material scores of the pieces to make the move.

2. The Optimised Algorithm. It uses the NegaMax algorithm along with Alpha Beta pruning to make the optimised move.

4.2 Screenshot:-

Here are few of the screenshots of the final product:-

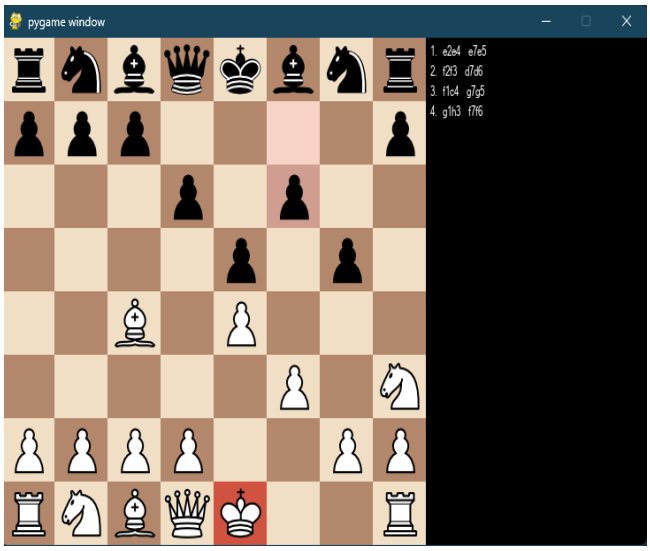

Figure 1

Figure 2

Chapter 5

Conclusion and Future Scope

5.1 Conclusion

In conclusion, the development of a Chess Engine in Python is a challenging but rewarding project. It requires a deep understanding of the game of chess, algorithmic thinking, and proficiency in the Python programming language.

The successful implementation of the project can provide valuable insights into the workings of chess and its strategic nuances. Moreover, the project can serve as a stepping stone for future advancements in artificial intelligence and machine learning, particularly in the domain of game theory. Overall, the Chess Engine in Python project provides an excellent opportunity for developers to hone their skills and contribute to the development of cutting-edge technologies.

5.2 Future Scope

The future scope of the project "Chess Engine in Python" is quite promising. Here are some potential areas for further development:

1. AI Improvements: Chess is a complex game, and creating a strong AI that can beat human players is a significant challenge. There is always scope for improving the chess engine's logic and strategies to enhance the AI's performance.
2. Multiplayer Functionality: Currently, the project supports only single-player mode. Adding a multiplayer feature would make it more engaging and entertaining for users.
3. User Interface: The user interface is an essential aspect of any application. Enhancing the UI/UX design of the project can significantly improve the user experience.
4. Integration with other platforms: Integrating the project with other platforms like mobile apps, web applications, and other gaming consoles can extend the project's reach and user base.
5. Game Variations: Chess has numerous variations like blitz chess, bullet chess, and many more. Incorporating these variations into the project can attract more users and make the application more exciting.

References

1. "Artificial Intelligence: A Modern Approach" by Stuart Russell and Peter Norvig
2. "Game Programming Patterns" by Robert Nystrom
3. "Introduction to Artificial Intelligence" by Wolfgang Ertel
4. "Mastering Python Design Patterns" by Sakis Kasampalis
5. "Deep Learning with Python" by Francois Chollet.

SAMPLE INDIVIDUAL CONTRIBUTION REPORT:

<CHESS ENGINE IN PYTHON >

<ROHIT GHOSH>
<1929038>

Abstract: The project aims to develop a sophisticated chess engine in Python using algorithms such as minimax and alpha-beta pruning, with various heuristics. The engine will analyze different chess positions, generate possible moves, and evaluate the best move based on the current board state. The engine will also include a graphical user interface for users to play against the engine and evaluate its performance against human players and other chess engines.

Individual contribution and findings: As the only contributor to this project, I take full responsibility for its development and the composition of this report

Full Signature of Supervisor:
…….

Full signature of the student:
ROHIT GHOSH

TURNITIN PLAGIARISM REPORT

(This report is mandatory for all the projects and plagiarism must be below 25%)