LOCI: A CHESS ENGINE FOR GENERATING GAME DATA

BY

ADEFOKUN AHIRA JUSTICE
185576

SUPERVISOR: MR. T. OGUNTUNDE

A REPORT SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF IBADAN, IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF A BACHELOR OF SCIENCE DEGREE IN COMPUTER SCIENCE

APRIL, 2019

CERTIFICATION

I certify that this work was carried out by **ADEFOKUN AHIRA JUSTICE,** with matric number, **185576** in partial fulfilment of the requirements for the award of a B.Sc in Computer Science and under my supervision.

_____ _____

Mr. T. Oguntunde **Date**

(Supervisor)

_____ _____

Dr S.O Akinola **Date**

(Head of Department)

DEDICATION

This work is dedicated to Jehovah, my parents and two brothers, Imri and Jucal.

ACKNOWLEDGEMENT

With all sincerity and deep gratitude, I wish to use this medium to acknowledge the efforts of all parties involved in making this project a success.

My gratitude goes to my Supervisor, Mr. T. Oguntunde for his guidance, encouragement and assistance for the entire duration of his supervision.

I deeply appreciate the effort of my wonderful lecturers, all whom have provided immense knowledge for my degree. Such knowledge has been integral to the completion of this project.

I would also acknowledge the emotional and moral support given to me by my siblings and friends for all the times the stress and work load weighed me down.

To my parents, Mr. and Dr. Adefokun–who have been providing all, and more than is expected of loving parents–I am much indebted.

God bless.

ABSTRACT

Chess is a game that requires creativity and sophisticated reasoning. It was once thought of as something computers will never be able to do. Much progress has been made in the field of computational chess in the last few decades as computers have become more mainstream. Computers play chess better than the best human grand masters. Part of their strength comes from evaluation functions. Tools for the comparison of different evaluation function metrics are not readily available. These evaluation functions form a very integral part of chess engines and are largely responsible for the strength of engines. Also, collating the data for this analysis is a difficult endeavor.

Loci was designed and implemented to provide a lasting solution to the aforementioned problems. Evaluation functions can be customized by modifying configuration files. The attendant changes are then observed in the produced data set.

Prototyping approach of software development was adopted and modelling was done using the Use case model, Activity diagram and Data-Flow diagrams (Context and Level-0).

Visual Studio Code was the primary development environment. The project was implemented in Python using the Pygame package for the development of the graphics module.

The fully developed project provides computational chess researchers and enthusiasts a system by which they can make changes to chess engine evaluation functions, and collect large data for study and analysis.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER ONE

INTRODUCTION

1.1 BACKGROUND OF THE STUDY

Games have been one of the most visible areas of progress in the AI space in the last few years. Chess, Jeopardy, GO and, very recently, Poker are some of the games that have been mastered by AI systems using breakthrough technologies. From that viewpoint, the success of AI seems to be really tied to the progress on game theory (Rodriguez, 2017).

While games are, obviously, the most visible materialization of game theory, it is far from being the only space on which those concepts are applied. From that perspective, there are many other areas that can be influenced by the combination of game theory and AI. The fact is that most scenarios that involve multiple "participants", collaborating or competing to accomplish a task, can be gamified and improved using AI techniques. Even though the previous statement is a generalization, I hope it conveys the point that game theory and AI is a way to think and model software systems rather than a specific technique.

Chess is a game that requires much creativity and sophisticated reasoning that it was once thought of as something no computers will ever be able to do. (Lai, 2015). It was frequently listed alongside activities like poetry writing and painting, as examples of tasks that can only be performed by humans. While writing poetry has remained very difficult for computers to this day, humans have had much more success building chess-playing computers. (Lai, 2015)

In 1997, IBM's Deep Blue defeated the reigning World Chess Champion, Garry Kasparov, under standard tournament rules, for the first time in the history of chess (Lai, 2015). In the ensuing two decades, both computer hardware and AI research advanced the state-of-art chess-playing computers to the point where even the best humans today have no realistic chance of defeating a modern chess engine running on a smartphone. (Lai, 2015)

1.2 PROBLEM STATEMENT

Training data for deep learning chess engines is rare to come by. Some of these datasets are collated manually from tournaments played by humans and artificial intelligence (AI). Also, tools for the comparison of different evaluation function metrics are not readily available.

1.3 AIM AND OBJECTIVES

The aim of this project is to build a chess engine (**loci)** for generating game data.

The proposed system has the following objectives:

1. To design and model a chess engine that enables **AI to AI** gaming.
2. To implement a chess engine model that provides large chess game data.
3. To implement a chess engine model that allows for changes to engine evaluation functions.

1.4 METHODOLOGY

- System requirement analysis was done with review of literature on chess engines and an examination of the manual existing system.
- Design was done using use case diagram, activity diagram, and data flow diagram.
- Implementation was done with Python, a programming language, and with plain text data in portable game notation (PGN) as the data output format.
- Deployment and Testing was done on a desktop computer with the command line as its interface.

1.5 SCOPE OF THE STUDY

The proposed system aims to provide a chess engine that produces chess game data. The system is designed to implement two distinct evaluation functions to describe two artificial intelligences. The system is also designed to be extendable, with the varying of the distinct evaluation functions to investigate the effects of heuristics changes.

1.6 JUSTIFICATION OF THE STUDY

In the development of chess engines, the vast majority of engines use brute force-linear algorithms. In recent years, there has been a push to make chess engines smarter by "teaching" them to play. This is a new area of artificial intelligence research that uses neural networks and deep learning to "teach" a computer to play chess. This project implements a system that produces data that can be used to train these neural networks. Also, heuristic analysis is critical to the strength of chess engines. This system aims to provide data that can be used to determine the effects of changes made to evaluation functions.

CHAPTER TWO

LITERATURE REVIEW

2.1 PREAMBLE

Chess is one of the oldest and most popular board games in the world, played by two opponents on a checkered board with specially designed pieces of contrasting colors, which are usually white and black (Soltis, 2017). As Soltis (2017) explains, white makes the first move, after which the players swap/alternate turns in accordance with fixed rules. The aim of each player is to force the opponent's principal piece, the King, into checkmate—a position where it is unable to avoid capture. (Soltis, 2017)

Of most interest in this project is the field of computational chess, the methods by which chess engines are built, and the relevance of this system to computational chess researchers and enthusiasts. These methods will be discussed in this chapter.

2.2 DEFINITIONS AND ACRONYMS

There are a few terms that come from a specialist register used in this and subsequent chapters. Definitions are presented in this section. Subsequently, the attached acronyms will be used instead.

Artificial Intelligence (AI):

Merriam-Webster defines artificial intelligence to be

"1: a branch of computer science dealing with the simulation of intelligent behavior in computers.

2: the capability of a machine to imitate intelligent human behavior."

Portable Game Notation (PGN):

Wikipedia defines Portable Game Notation as

"a plain text computer-processible format for recording chess games (both the moves and related data), supported by many chess programs."

Integrated Development Environment (IDE):

Wikipedia defines an integrated development environment as

"An integrated development environment is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a source code editor, build automation tools, and a debugger. Most of the modern IDEs have intelligent code completion."

Minimum Viable Product (MVP):

Wikipedia defines an integrated development environment as

"A minimum viable product (MVP) is a product with just enough features to satisfy early customers, and to provide feedback for future product development."

Universal Chess Interface (UCI)

"an open communication protocol for chess engines to play games automatically, that is to communicate with other programs including Graphical User Interfaces."

2.3 COMPUTER GAMES

Man has been fascinated by the idea that a computer can play games of skill ever since the advent of the electronic computer (Levy, 1983). The word skill implies some form of intelligence, and while it is implied that only humans can exhibit the type of intelligence needed to play games of skill, such as chess, bridge and poker, it so happens that the type of person who programs computers is also the type of person that enjoys playing games of skill (Levy, 1983). This similarity is important. As it turns out, the procedural analytical assessment made by players of games of skill is exactly that which is needed by computers to solve problems. As Levy (1983) explains, this similarity also accounts for the widespread interest amongst the computing fraternity in the programming of games of skill.

During the first 30 years in the history of computers, it could be argued that since computers were so expensive, it was frivolous to spend so much time on tasks like playing chess or poker (Levy, 1983). On the flip side, the scientific aspect of chess programming for instance showed that in performing a task usually associated with human intelligence, one could be said to be building artificial intellect (Levy, 1983). Levy (1983) also espouses another point in support of those who would devote efforts to writing such programs. He explains that,

by writing successfully a program in which long-term planning is involved, one would devise methods which could be applied to other long-term planning problems, such as economic forecasting. This has proven to be true in many aspects of computing and algorithms.

2.4 CHESS

Chess is played on an 8×8 grid, a board of 64 squares arranged in eight columns called files and eight rows called ranks (Soltis, 2017). These squares alternate between two colors: one light, such as white, beige, or yellow; and the other dark, such as black or green. The board is set between the two opponents so that each player has a light-colored square at the right-hand corner. (Soltis, 2017). This is illustrated in the figure below.

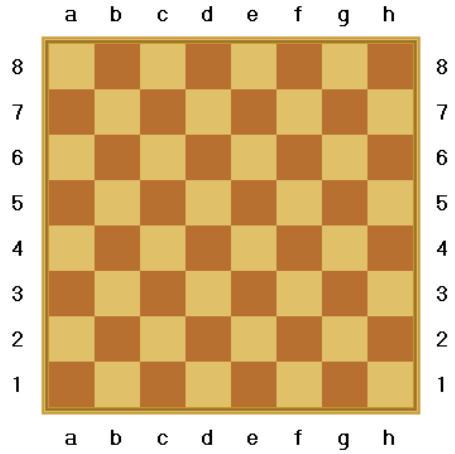

Figure 1: Chess Board

2.4.1 ALGEBRAIC NOTATION

Individual moves and entire games can be recorded using one of several forms of notation. Algebraic (or coordinate) notation, identifies each square from the point of view of the player with the light-colored pieces, also called White (Soltis, 2017). Algebraic notation is by far the most widely used form of chess notation.

The eight ranks are numbered 1 through 8 beginning with the rank closest to White. The files are labeled a through h beginning with the file at White's left hand. The name of each square is a combination of its rank and file coordinate, such as h7 or g8. Additional concepts worthy of note are queenside and kingside. The portion of the board containing files a - d are

referred to as the queenside, while files e through h is referred to as the kingside. (Soltis, 2017). This can be seen in the image above.

2.4.2 PIECES

KING

Each player has only one king. Soltis (2017) describes a King in his treatise. He says; "White's king begins the game on e1. Black's king begins the game at e8 and is opposite to White's king". Each king can move one square in any direction; e.g., From its starting position, White's king can move from to d1, d2, e2, f2, or f1.

ROOK

Each player has two rooks. Soltis (2017) defines a Rook and its functions. He says; "rooks begin the game on the corner squares a1 and h1, and a8 and h8 for White and Black respectively". A rook can move vertically or horizontally to any unobstructed square along the file or rank on which it is placed.

BISHOP

Each player has two bishops. As Soltis (2017) explains, Bishops begin the game at c1 and f1 for White, c8 and f8 for Black. A bishop can move to any unobstructed square on the diagonal on which it is placed. This results in a peculiar trait, each player has one bishop that travels only on light-colored squares and one bishop that travels only on dark-colored squares.

QUEEN

Each player has one queen. The queen combines the powers of the rook and bishop—i.e., vertical, horizontal and diagonal moves to unobstructed squares. This makes it the most powerful and movable piece. The White queen begins at d1 and the Black queen at d8.

KNIGHT

Each player has two knights. They start out the game on the squares between their rooks and bishops—i.e., at b1 and g1 for White and b8 and g8 for Black. The movement of the knight is unique. Soltis (2017) explains it thus; "the knight moves in an L-shape of two steps: first one square like a rook, then one square like a bishop, but always in a direction away from the starting square". This means that, a knight at e4 could make a move to any of f2, g3, g5, f6, d6,

c5, c3, or d2. The knight has the unique ability to jump over any other piece to a new square. The knight always moves to a square of a different color (Soltis, 2017).

PAWN

Each player has eight pawns. They all begin the game on the second rank closest to each player. White's pawns start at a2, b2, c2, through to h2, while Black's pawns start at a7, b7, c7, through to h7. Pawns are unique in several ways.

One way is that a pawn can move only forward; it can never retreat (Soltis, 2017). Also, a pawn moves differently than it captures. A pawn moves to the square directly ahead of it but captures on the squares diagonally in front of it. For example, a White pawn at f5 can move to f6 but can capture only on g6 or e6.

Another attribute of pawns is that an unmoved pawn has the option of moving one or two squares forward. Soltis (2017) says this is the reason for another unique property of pawns, called en passant—that is French for in passing. En passant is available to a pawn when an enemy pawn on an adjoining file advances two squares on its initial move and could have been captured had it moved only one square (Soltis, 2017). The first pawn can take the advancing pawn, as if it had moved only one square. An en passant capture must be made on the first move or not at all (Soltis, 2017). Only pawns can capture or be captured en passant.

Lastly, any pawn reaching the end of its file is to be promoted—that is, exchanged for—a queen, rook, bishop, or knight (Soltis, 2017).

2.4.3 MOVES

The board represents a battlefield of sorts, in which two armies fight to capture each other's king (Soltis, 2017). In this context, a player's army consists of 16 pieces that begin battle on the two ranks closest to that player, that is on a total of 16 squares (Soltis, 2017). The six different types of pieces: king, rook, bishop, queen, knight, and pawn; as defined above are distinguished by appearance and by how they move. The players alternate moves, White going first (Soltis, 2017).

CAPTURING

The king, rook, bishop, queen, and knight capture enemy pieces in the same manner that they move, the only exception is the pawn. For example, a White queen on d3 can capture

a Black rook at h7 by moving to h7 and removing the enemy piece from the board. Pieces can capture only enemy pieces (Soltis, 2017).

CASTLING

The only exception to the rule that a player may move only one piece at a time is a compound move of king and rook called castling (Soltis, 2017). Soltis (2017) explains that a player castles by shifting the king two squares in the direction of a rook, which is then placed on the square the king has crossed. For example, White can castle kingside by moving the king from e1 to g1 and the rook from h1 to f1.

Castling is permitted only once in a game and is prohibited if the king or rook has previously moved or if any of the squares between them is occupied.

Also, castling is not legal if the square the king starts on, crosses, or finishes on is attacked by an enemy piece.

2.4.4 RELATIVE PIECE VALUES

Assigning the pawn a value of 1, the values of the other pieces are approximately as follows: knight 3, bishop 3, rook 5, and queen 9 (Soltis, 2017). The King is given an arbitrarily high value, say 100.

The relative values of knights and bishops vary with different pawn structures. Additionally, tactical considerations may temporarily override the pieces' usual relative values (Soltis, 2017). Material concerns are secondary to winning.

2.4.5 OBJECT OF THE GAME

When a player moves a piece to a square on which it attacks the enemy king—that is, a square from which it could capture the king if the king is not shielded or moved—the king is said to be in check. The game is won when one king is in check and cannot avoid capture or be shielded on the next move; this is called checkmate. A game also can end when a player resigns or acknowledges defeat, believing the situation to be hopeless.

There are three possible results in chess: win, lose, or draw (Soltis, 2017). These are assigned a value of 1, 0 and ½ respectively in PGN (Edwards, 1993).

Soltis (2017) describes the six ways a draw can come about; a draw can be come about (1) by mutual consent, (2) when neither player has enough pieces left to deliver checkmate, (3) when one player can check the enemy king in perpetuity, (4) when a player who is not in check has no legal move (stalemate), (5) when an identical position occurs three times with the same player having the right to move, and (6) when no piece has been captured and no pawn has been moved within a period of 50 moves.

In competitive events, a victory is scored as one point, a draw as half a point, and a loss as no points.

2.4.6 GAME NOTATION

Moves can be recorded by designating the initial of the piece moved and the square to which it moves (Edwards, 1993). For example, Be5 means a bishop has moved to e5. There are two exceptions: a knight is identified by N, and no initials are used for pawn moves. In the case of the pawn, 1 e4 means White's first move is a two-square advance of a pawn on the e-file, and for the knight, 1 … Nf6 means Black's response is to bring a knight from g8 to f6.

Castling kingside is indicated by 0-0, for both White and Black, while castling queenside is notated by 0-0-0 (Soltis, 2017).

Captures are indicated by inserting an x or: between the piece moving and the square it moves to. For pawn moves, this means dxe5 indicates a White pawn on d4 captures a piece on e5. En passant captures are designated by e.p. (Soltis, 2017).

Checks are indicated by adding ch or + at the end of the move, and checkmate is often indicated by adding # or ++ at the end of the move. Notation is used to record games as they are played and to analyze them in print afterward (Soltis, 2017).

When annotating (commenting) on a game, Soltis (2017) says;

> *"an appended exclamation mark means a very good move, two exclamation marks are occasionally used to indicate an extremely good move, a question mark indicates a bad move, two question marks indicate a blunder, and the combination of an exclamation mark and a question mark on the same move indicates a double-edged or somewhat dubious move."*

2.5 CHESS AS A COMPUTER GAME

Machines capable of playing chess have fascinated people since the latter half of the 18th century, when the Turk, the first of the pseudo-automatons, began a triumphal exhibition tour of Europe (Soltis, 2017). Like its 19th-century successor Ajeeb, the Turk was a cleverly constructed cabinet that concealed a human master (Soltis, 2017). The mystery of the Turk was the subject of more than a dozen books and a widely discussed article written by Edgar Allan Poe in 1836 (Eschner, 2017). Several world-class players were employed to operate the pseudo-automatons, including Harry Nelson Pillsbury, who was Ajeeb during part of the 1890s, and Isidor Gunsberg and Jean Taubenhaus, who operated, by remote control, Mephisto, the last of the pseudo-automatons, before it was dismantled following World War I (Soltis, 2017).

2.5.1 COMPUTER CHESS

As Soltis (2017) explains, computers began to compete against humans in the late 1960s. Richard Greenblatt, an MIT undergraduate wrote MacHack VI in February of 1967 (Greenblatt, Eastlake III, & Crocker, 1969). This program drew one game and lost four in a U.S. Chess Federation (USCF) tournament. Its results improved markedly, from a performance equivalent to a USCF rating of 1243 to reach 1640 by April 1967, about the average for a USCF member (Greenblatt, Eastlake III, & Crocker, 1969). The first American computer championship was held in New York City in 1970 and was won by Chess 3.0, a program devised by a team of Northwestern University researchers that dominated computer chess in the 1970s (Soltis, 2017).

Soltis (2017) further explains that, technical advances accelerated progress in computer chess during the 1970s and ’80s. The sharp increases in computing power recorded in that period enabled computers to “see” much further. Computers of the 1960s could evaluate positions no more than two moves ahead, but authorities estimated that each additional half-move of search would increase a program’s performance level by 250 rating points. This was borne out by a steady improvement by the best programs until Deep Thought played above the 2700 level in 1988 (Soltis, 2017). When Deep Blue, its successor, was introduced in 1996, it saw as far as six moves ahead. (Gary Kasparov said he normally looks only three to five moves ahead, adding that for humans more are not needed.) (Soltis, 2017)

Soltis (2017) continues, another factor helping computer progress was the availability of microprocessors in the late 1970s. This allowed programmers unattached to universities to

develop commercial microcomputers that by the 1990s were nearly as strong as programs running on mainframes (Soltis, 2017). The strongest machines were capable of beating more than 90 percent of the world's serious players by the late 1980s. In 1988 a computer, HiTech, developed at Carnegie Mellon University, defeated a grandmaster, Arnold Denker, in a short match (Soltis, 2017). In the same year another Carnegie Mellon program, Deep Thought, defeated a top-notch grandmaster, Bent Larsen, in a tournament game. (Soltis, 2017)

At faster speeds even personal computers were able to defeat the world's best humans by 1994 (Soltis, 2017). In that year a Fritz 3 program, examining 100,000 positions per second, tied for first place with Kasparov, ahead of 16 other grandmasters, at a five-minute tournament in Munich, Germany (Soltis, 2017). Later in the year Kasparov was eliminated from a game/25 tournament in London after losing a two-game match against Genius running on a Pentium personal computer (Soltis, 2017).

In 1991 Deep Thought's team said the program, renamed Deep Blue, would soon be playing at the equivalent of a 3000 rating (compared with Kasparov's 2800), but this proved excessively optimistic (Soltis, 2017). The main improvement was in the computer running the chess program. IBM developed a sophisticated new multiprocessing system (later used at the 1996 Olympic Games in Atlanta, Georgia, U.S., to predict the weather) that employed 32 microprocessors, each with six programmable chips designed specifically for chess. Soltis (2017) explains that, Deep Thought had only one microprocessor and no extra chips, by comparison. The new hardware enabled Deep Blue to consider as many as 50 billion positions in three minutes, a rate that was about a thousand times faster than Deep Thought's. (Soltis, 2017)

Deep Blue made its debut in a six-game match with PCA champion Kasparov in February 1996 (Soltis, 2017). The $500,000 prize fund and IBM's live game coverage at their World Wide Web site attracted worldwide media attention (Soltis, 2017). The Kasparov–Deep Blue match in Philadelphia was the first time a world champion had played a program at a slow (40 moves in two hours) time format. Deep Blue won the first game, but Kasparov modified his style and turned the later games into strategic, rather than tactical, battles in which evaluation was more important than calculation (Soltis, 2017). He won three and drew two of the remaining games to win the match 4–2.

In a six-game rematch held May 3–11, 1997, in New York City, an upgraded Deep Blue was able to consider an average of 200 million positions per second, twice its previous speed. Its algorithm for considering positions was also improved with advice from human grandmasters. (Soltis, 2017)

Soltis (2017) says, “By adopting a new set of conservative openings, Kasparov forced Deep Blue out of much of its prematch preparation.” After resigning the second game, in a position later found to be drawable, Kasparov said he “never recovered” psychologically. With the match tied at one win, one loss, and three draws, Deep Blue won the decisive final game in 19 moves. (Soltis, 2017)

2.5.2 COMPUTER EXTENSION OF CHESS THEORY

Computers have played a huge role in the extending of chess knowledge. In 1986 Kenneth Thompson of AT&T Bell Laboratories reported a series of discoveries in basic endgames (Soltis, 2017). By working backward from positions of checkmate, Thompson was able to build up an enormous number of variations showing every possible way of reaching the final ones (Soltis, 2017). Soltis (2017) explains that this has only been possible with the most elementary endgames, with no more than five pieces on the board. Thompson’s research proved that certain conclusions that had remained unchallenged in endgame books for decades were untrue (Soltis, 2017). For example, with best play on both sides, a king and queen can defeat a king and two bishops in 92.1 percent of the initial starting positions; this endgame had been regarded as a hopeless drawn situation (Soltis, 2017). Also, a king and two bishops can defeat a king and lone knight in 91.8 percent of situations—despite human analysis that concluded the position was drawn. Thompson’s research of some five-piece endgames required considering more than 121 million positions (Soltis, 2017).

Because of their ability to store information, computers had become invaluable to professional players by the 1990s, particularly in the analysis of adjourned games (Soltis, 2017). However, computers have some severe limits. In the 1995 PCA championship, Kasparov won the 10th game with a heavily analyzed opening based on the sacrifice of a rook (Soltis, 2017). According to his aides, the prepared idea was tested on a computer beforehand, and the program evaluated the variation as being in the opponent’s favor until it had reached the end of Kasparov’s lengthy analysis (Soltis, 2017).

All the above is not an exhaustive discussion of chess and its many forms, variants or history. Suffice it to say that it is a concise description of computer chess and its implications for how chess theory has been developed.

2.6 SIMILAR SYSTEMS

Below is a consideration of systems similar to this project. These systems show similarity either in form, or in function.

2.6.1 SUNFISH

Sunfish is a simple, but strong chess engine, written in Python, mostly for teaching purposes. Without tables and its simple interface, it takes up just 111 lines of code. (Ahle, 2017)

Because Sunfish is small and strives to be simple, the code provides a great platform for experimenting. People have used it for testing parallel search algorithms, experimenting with evaluation functions, and developing deep learning chess programs. Fork it today and see what you can do. (Ahle, 2017)

2.6.2 DEEP PINK

Deep Pink is a chess AI that learns to play chess using deep learning. Here (http://erikbern.com/2014/11/29/deep-learning-for-chess/) is a blog post providing some details about how it works. (Bernhardsson, 2017)

There is a pre-trained model in the repo, but if you want to train your own model you need to download pgn files and run parse_game.py. After that, you need to run train.py, preferably on a GPU machine since it will be 10-100x faster. This might take several days for a big model. (Bernhardsson, 2017)

2.7 THE THEORY FOR CHESS ENGINES

Next, the theory that is relevant to building a basic chess AI is examined. Although there is variation between engines, almost all chess engines today implement the same algorithms (Lai, 2015). They are all based on the idea of the fixed-depth minimax algorithm developed by John von Neumann in 1928 (Neumann, 1928). It is contentious what it is that fundamentally sets these engines apart, though it is conjectured that the evaluation function is responsible for the major differences between the top chess engines (Levy, 1983).

Neumann's minimax algorithm was adapted to the problem of chess by Claude E Shannon in 1950 (Shannon, 1950). In Shannon's solution, he describes the minimax algorithm alongside α-β Pruning.

MINIMAX

The minimax algorithm is a simple recursive algorithm to score a position based on the assumption that the opponent thinks like humans do, and also wants to win the game (Lai, 2015).

In its simplest form, minimax can be expressed as

```
function minimax(position) {
      if position is won for side to move:
            return 1
      else if position is won for the opponent:
            return -1
      else if position is drawn:
            return 0

      bestScore = -∞

      for each possible move mv:
            subScore = -minimax (position.apply(mv))
            if subScore > bestScore:
                  bestScore = subScore

      return bestScore
}
```

This algorithm works in theory, and also in practice for simpler games like tic-tac-toe (game tree size of at most 9! or 362880) (Lai, 2015). Tic-tac-toe is a solved game. This means that for any given game, the end can always be computed. Chess is not solved, and we might never be able to. The search tree size of chess is estimated to be about 10^{123} (Allis, 1994) which is more than one hundred orders of magnitudes higher than what is computationally feasible using modern computers (Lai, 2015). For comparison, the estimated total number of atoms in the known universe is 10^{78} to 10^{82}. Therefore, a chess engine must decide which parts of the game tree to explore (Lai, 2015).

The most common approach is a fixed-depth search (Lai, 2015), arbitrarily limiting how far down the game tree the engine will look, and when we are at the end of the sub-tree we want to search, a call to a static evaluation function that assigns a score to the position by

analyzing it statically is made (Lai, 2015). A very simple evaluation function is to simply add up pieces of both sides, each multiplied by a constant (for instance, the relative piece values defined above. Q = 9, R = 5, B = 3, N = 3, P = 1). If these steps are applied, then a new form of the algorithm can be described.

```
function minimax(position, depth) {
      if position is won for the moving side:
            return 1
      else if position is won for the non-moving side:
            return -1
      else if position is drawn:
            return 0

      if depth == 0:
            return evaluate(position)

      bestScore = -∞

      for each possible move mv:
            subScore = -minimax(position.apply(mv), depth-1)
            if subScore > bestScore:
                  bestScore = subScore

      return bestScore
}
```

Three changes have been made

- An extra parameter, depth, has been added to minimax(). It represents how deep we want to search from the current position.
- If the depth is 0, evaluate() is called and its result returned.
- If depth > 0, minimax is called recursively, depth-1 is passed as the depth for the sub-tree.

Given reasonable depth, the search now terminates in reasonable time. This creates a new problem though – the horizon effect (Lai, 2015).

The horizon effect is dangerous because we are cutting off all branches at the same distance (called the horizon) (Lai, 2015). For instance, if the last move in one branch is QxP (queen captures pawn), we may score that position as a good one, since we just won a pawn. If however, that captured pawn is defended by another pawn, the opponent will take our queen on the next move (Lai, 2015).

This problem cannot simply be solved by increasing the depth limit, because no matter how deep we search, there always will be a horizon. The solution is quiescent search. The idea is that once *depth* == 0, we do not call evaluate() and return, rather we enter a special search

mode that only expands certain types of moves, only calling evaluate() when we get to a "quiet" and relatively stable position (Lai, 2015).

There is some debate amongst engine authors about the types of moves to include in q-search (Lai, 2015). All engine authors agree that high value captures should be included. Many believe queen promotions should be included. Some believe check and check evasions are also important, and so on (Lai, 2015).

There is a tradeoff to be made here – if too many moves are included, q-searches become too large. If too few moves are included, we may suffer some reduced forms of the horizon effect (Lai, 2015).

α-β PRUNING

Without any loss of information or introducing any heuristics, our search algorithm can be optimized by introducing a "window" for every call to minimax() (Lai, 2015). This optimization is called α-β pruning (Newell & Simon, 1976), where α and β are the lower and upper bounds. Since α and β are horrible choices of variable names, let us refer to them as lowerbound and upperbound in the rest of this report (Lai, 2015).

As Lai (2015) explains, the basic idea is, if the true score is below the lowerbound, that means the calling function already has a better move, and therefore it shouldn't care about the exact value of this node (only the fact that it is lower than the lowerbound). On the other hand, if the true score is higher than the upperbound, that also means the calling function shouldn't care about the exact value, just that it is higher than the upperbound (Lai, 2015). It may seem counter-intuitive at first to have an upperbound, but the reason is that chess is a zero-sum game and upperbound and lowerbound switch places for each ply as we search deeper.

A detailed analysis of α-β pruning is omitted here for brevity. There is one very important result that needs highlighting – that the order in which moves are searched is extremely important (Lai, 2015).

When we examined standard minimax, move ordering (the order in which nodes of the game tree are explored) was irrelevant as all nodes are to be visited exactly once (Lai, 2015). With α-β, we explore the nodes that have the potential to be "useful", and we stop searching a node once we prove that the result will be outside our defined window (Lai, 2015). If we could by some means expand only the best nodes first, we will not have to examine as many nodes as with less optimal ordering (Lai, 2015).

We examine the worst case (opposite of optimal ordering). In this scenario, α-β degenerates into standard minimax (Lai, 2015). On the other hand, it can be shown that in the best case, α-β allows the search to go twice as far in the same amount of time (Russell & Norvig, 2019).

There is no way to ensure optimal ordering all the time. If there was, there would no need to search at all (Lai, 2015). However, we can ensure that move ordering is near optimal and thereby make the search more efficient by making heuristics changes (Lai, 2015).

Depth-limited minimax() with α-β pruning and q-search form the backbone of virtually all existing chess engines (Lai, 2015).

EVALUATION

As mentioned above, the evaluation function is a very important part of a chess engine, and almost all improvements in playing strength among the top engines nowadays come from improvements in their respective evaluation functions (Lai, 2015). We delve into a discussion of evaluation functions and how their construction influences chess engine behavior.

The job of the evaluation function is to statically assign scores or values to positions (i.e. not looking ahead or down the game tree) (Lai, 2015). Evaluation functions contain most of the domain-specific knowledge designed into chess engines.

In this project, we will develop an evaluation function based on a configuration file approach. Domain-specific knowledge will be put into configuration files to be read by the engine. These files contain information about relative piece values based on their current board position. However, before we do that, let us take a look at the evaluation function of Stockfish (Home - Stockfish - Open source chess engine, 2015), an open source chess engine that is currently [one of] the strongest chess engines in the world (Lai, 2015). Examining this state-of-art evaluation function can help us define an effective evaluation function model for this project.

Stockfish's evaluation function consists of 9 parts

- **Material**: Each piece on the board gets a score for its existence. Synergetic effects are also taken into account (for example, having both bishops gives a bonus higher than two times the material value of a bishop), and polynomial regression is used to model more complex interactions.

- **Piece-Square Tables**: Each piece gets a bonus or penalty depending on where they are, independent of where other pieces are. This evaluation term encourages the engine to advance pawns and develop knights and bishops for example.
- **Pawn Structure**: The position is scanned for passed, isolated, opposed, backward, unsupported, and lever pawns. Bonuses and penalties are assigned to each feature. These features are all local, involving 2-3 adjacent pawns.
- **Piece-specific Evaluation**: Each piece is evaluated individually, using piece-type-specific features. For example, bishops and knights get a bonus for being in a "pawn outpost", rooks get a bonus for being on an open file, semi-open file, or the same rank as enemy pawns.
- **Mobility**: Pieces get bonuses for how many possible moves they have. In Stockfish's implementation, squares controlled by opponent pieces of lesser value are not counted. For example, for bishop mobility, squares controlled by enemy pawns are not included, and for queen mobility, squares controlled by bishops, knights, or rooks are not included. Each piece type has a different set of mobility bonuses.
- **King Safety**: Bonuses and penalties are given depending on the number and proximity of attackers, completeness of pawn shelter, and castling rights.
- **Threat**: Undefended pieces are given penalties, defended pieces are given bonuses depending on piece type, and defender type.
- **Space**: Bonuses are given for having "safe" empty squares on a player's side.
- **Draw-ish-ness**: Certain material combinations often result in draws, so in these cases, the evaluation is scaled toward 0.

It is quite a complicated function with a lot of hand-coded knowledge. Most engines don't have evaluation functions that are nearly as extensive, because it is difficult to tune such a high number of parameters by hand (Lai, 2015).

LOCI

The three parts of conventional chess engine build outlined above in this section are the fundamentals that go into building the chess engine for this project, Loci. Although there is some variation from this outlined standard, Loci maintains this theoretical framework. Some of this variation is the focus on two AIs, thereby requiring the definition of two evaluation

function configuration files. Also, the storage of game data necessitates the development of a "PGN engine" of sorts.

The next chapter details the methodology and design required to build such a system, per software engineering standards.

CHAPTER THREE

METHODOLOGY

3.1 OVERVIEW

In developing this system, the **prototyping paradigm** of software engineering was used. Sommerville (2011) describes software engineering to be the production of software from the early stages of system specification through to maintenance of the system after it has been deployed. For this project, the development of the software system includes the following activities:

- System requirement analysis
- Design
- Implementation
- Deployment and testing

3.2 EXISTING SYSTEMS

The existing systems, by which a developer of deep learning chess engines could collate data for the training of its neural network is largely a mammoth task of obtaining precomputed data of chess games from online repositories and normalizing or parsing such data into usable formats either by manual means or automated scripting or both.

Also, there are no systems of note, which permit evaluation function changes to the degree of studying effects of heuristics changes.

The system requirements are obtained both from an examination of the existing systems and by considerations of how it can be extended upon and made better.

3.3 PROPOSED SYSTEM

The proposed system generates game data dynamically, as games played in-the-moment by two distinct “artificial intelligences” and stored as portable game notation (PGN).

The proposed system is very processor intensive and requires a considerable amount of computational power to run efficiently. This is the most crucial non-functional requirement of

the system. Since the system does not deal with the management of a database, security as a non-functional requirement is not a primary concern.

The system is designed to receive input from the user on the number of games to compute. It is also designed to receive configuration input specifying the value of individual game pieces. These configuration files prescribe the behavior of the AIs that will play against each other. After the above-specified input has been supplied by the user, computation proceeds for a given period of time, hours or days even. The time spent in computation is a factor of the number of games played by the engine and the computational power of the deployment environment.

The proposed system has a **client-server architecture**, with a command-line interface as the client frontend, and the chess engine as the server backend.

This system is a niche application and thus requires some expertise with computational chess to be used effectively.

3.4 SYSTEM REQUIREMENTS SPECIFICATION

Software system requirements can be classified as functional requirements or non-functional requirements (Sommerville, 2011). Functional requirements describe the services that a system provides, how it reacts to certain input, and how it behaves in given situations (Sommerville, 2011). Non-functional requirements are the constraints on the services the system offers. This includes timing constraints, development constraints, and constraints imposed by standards for the software system (Sommerville, 2011)

3.4.1 NON-FUNCTIONAL REQUIREMENTS

For this system, the primary non-functional requirement is **performance**. The system requires a considerable amount of computational power to produce results within reasonable time limits, as it is very process bound and computationally complex system. This requires a computer with a very fast processor and huge amounts of memory.

Correctness is also a key non-functional requirement of the system. The correctness of the game data stored in portable game notation (PGN) depends on the correctness of each stored move. Any errors produced by the system would affect the integrity of the produced data.

Requirements like security, while considered, are not important requirements for this system.

Hardware Requirements

Recommended hardware specifications

Processor Intel® Core™ i7-7700HQ processor Quad-core 2.80 GHz

Memory 16 GB DDR4 RAM Memory

Software Requirements

Windows 7, 8, 8.1, 10

Python 3.x

Python Chess

Pygame

Tools

Visual Studio Code

Python

3.4.2 FUNCTIONAL REQUIREMENTS

The functional requirements of a system describe what that system should do, along with its primary functions (Sommerville, 2011).

The functional requirements of this system are:

- A user shall be able to tell the system how many games they want to be computed.
- A user shall be able to extend the system and implement their own evaluation functions.
- A user shall be able to access the data of computed games in portable game notation (PGN) format.
- The system shall be able to receive input specifying the number of games to be played, and validate the correctness of this input.
- The system shall be able to receive configuration input describing the evaluation functions for the AI that will play the specified number of games, and validate the correctness of this input.
- The system will play the specified amount of games and produce the data of those games in portable game notation (PGN) format.

3.5 SYSTEM MODELING

In designing this system, some system models are used to describe the various aspects of the system's operation. For the interaction model, a use case diagram is produced. An activity diagram along with a development view of the system architecture is also produced.

3.5.1 USE CASE DIAGRAM

The use case model is used to describe a simple scenario of what a user expects from a system (Sommerville, 2011).

The use case diagram presented below makes a simplification of the above stated functional requirements and depicts it in terms of how the user interacts with the system.

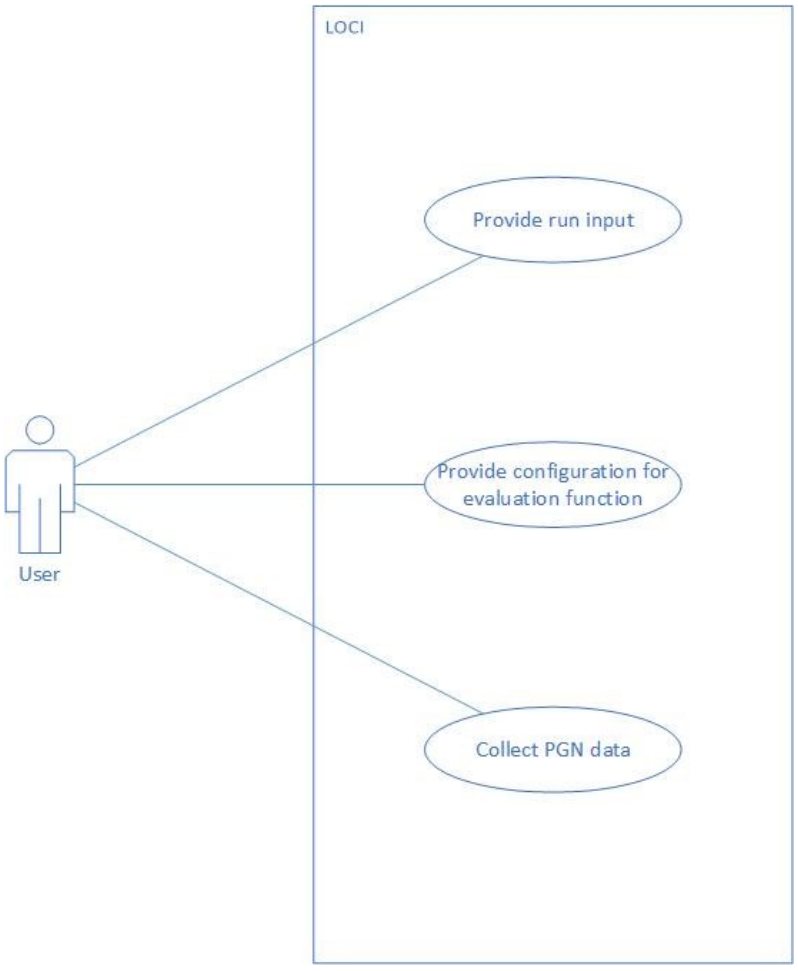

Figure 2: Use case diagram

3.5.2 ACTIVITY DIAGRAM

Activity diagrams are used to show all the activities that are included in a system process and the flow of control from one activity to another (Sommerville, 2011).

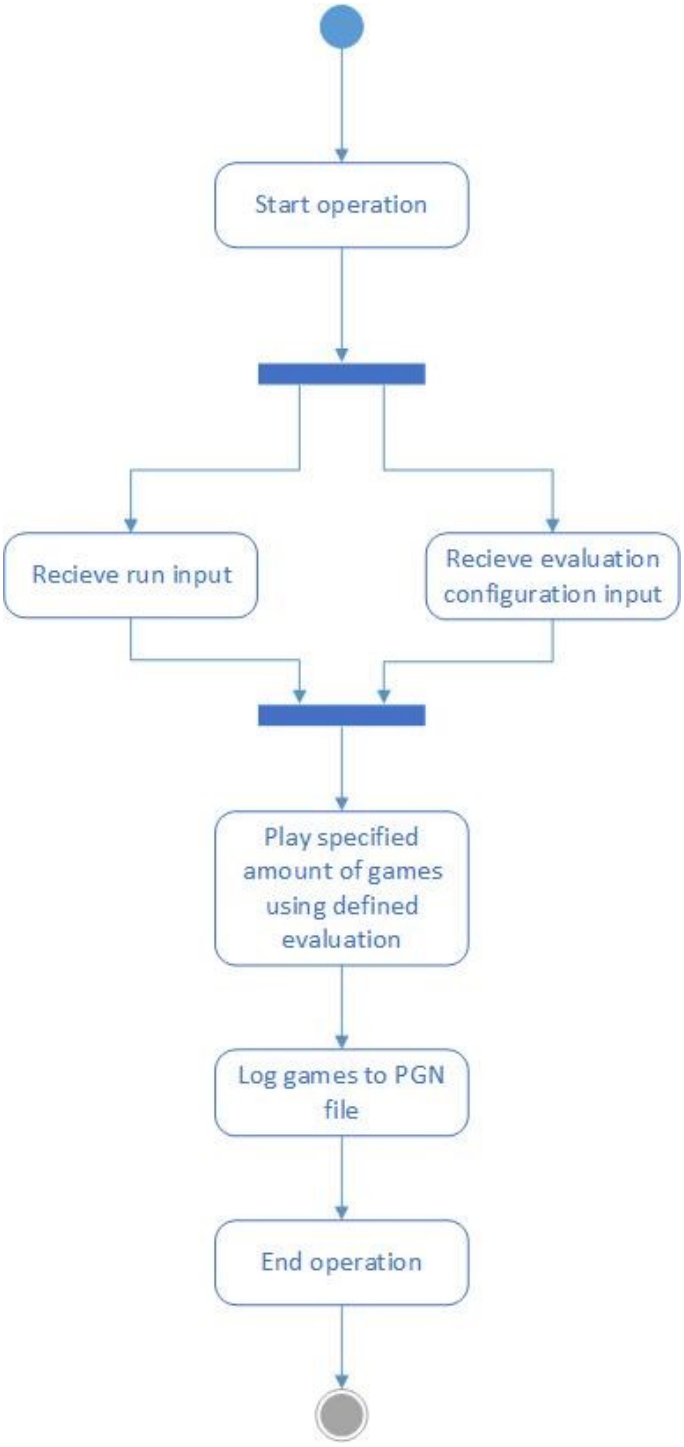

Figure 3: Activity diagram

3.5.4 DATA-FLOW DIAGRAM

Data-flow diagrams are used in the process modeling of a system. Valacich et al (2012) describe a data-flow diagram as a graphic/image that illustrates the movement of data between external entities, the processes, and the data stores within a system.

There are two levels of abstraction of the DFD that are of interest in the design of this system, the **context diagram,** and the **level-0 diagram**.

A context diagram shows the boundary or scope of the system, and the system's relationship to its environment (Valacich, George, & Hoffer, 2012). The context diagram has only one process. Below is the context diagram for this system.

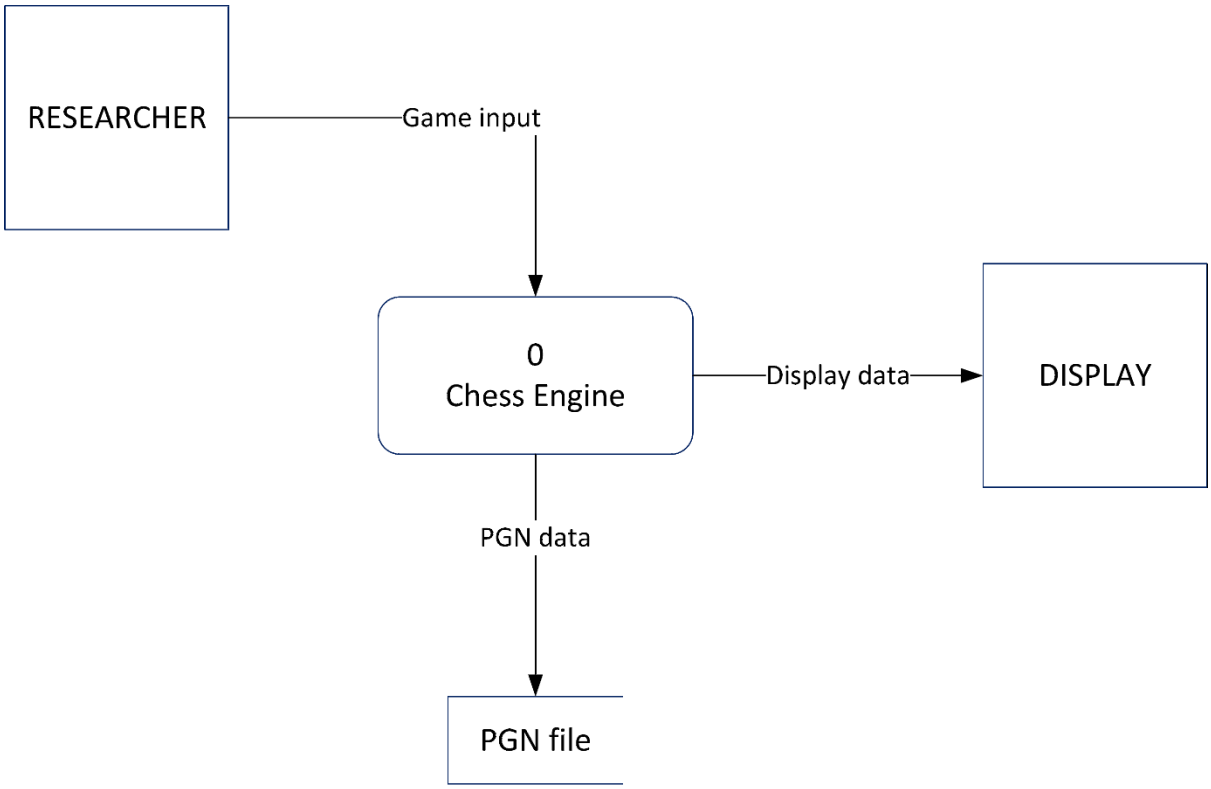

Figure 4: Context diagram

The level-0 diagram represents the basic individual processes in the system at the highest level of abstraction (Valacich, George, & Hoffer, 2012). Below is the level-0 diagram for this system. There are 4 primary processes described.

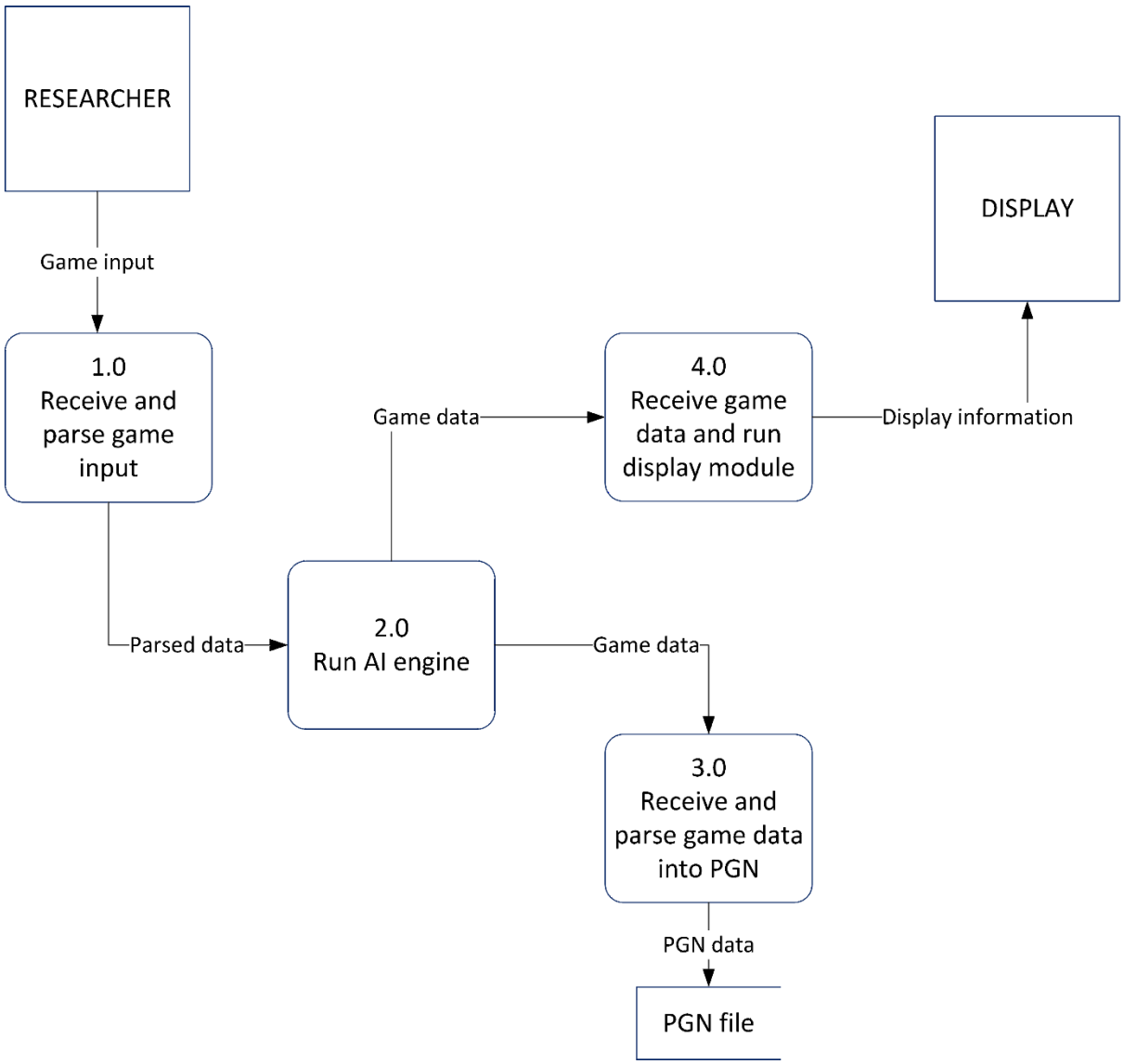

Figure 5: Level-0 diagram

3.5.4 SYSTEM ARCHITECTURE

As Somerville (2011) discusses, architectural modeling can be done by examining what views or perspectives are useful when designing and documenting a system. Somerville (2011) goes on to describe four views that are relevant to system architectures; *a logical view, a process view, a development view, and a physical view*.

For the purposes of this system, I describe a **development view** of system architecture below.

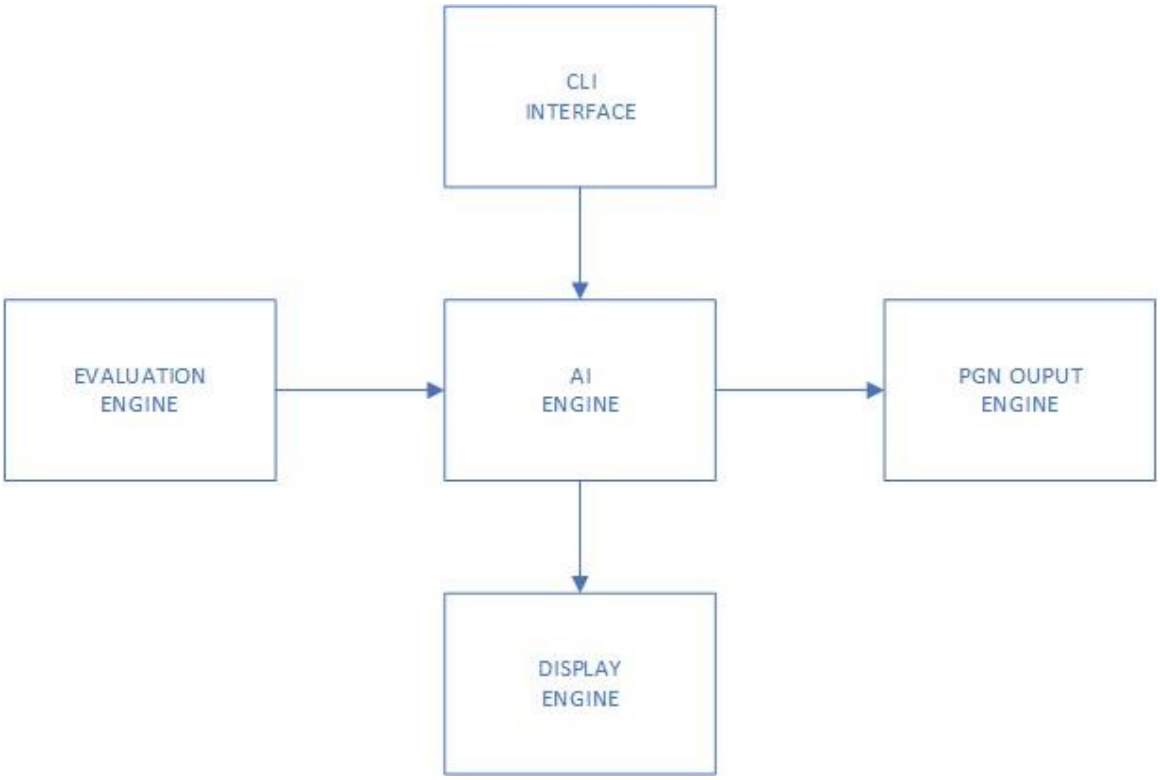

Figure 6: Architecture

3.6 IMPLEMENTATION TOOLS

The language chosen for implementation is **Python**. Python is desirable for use in this project for some reasons.

One reason is that code written in Python is easy to read as it has a very English-like syntax. This is important because one of the functional requirements is user extendibility of the system. Python is a scripting language and this makes it easier for a third-party to add functionality to the code.

Another reason is that python has suitable APIs and libraries that make the development of this system easier. One of those libraries is **python-chess**, a pure Python chess library with move generation, move validation and support for common formats (Fiekas, 2019). The version of python-chess used with this project is *python-chess 0.23.8*.

The visualization GUI will be built using **pygame**. Pygame is a Python wrapper module for the SDL multimedia library. It contains python functions and classes that will allow you to use SDL's support for playing CD-ROMs, audio and video output, and keyboard, mouse and joystick input (Shinners, Dudfield, Appen, & Pendleton, 2018).

The chosen IDE is **Visual Studio Code** and the version control is **Git**.

CHAPTER FOUR

IMPLEMENTATION

4.1 OVERVIEW

This chapter focuses on discussing the results obtained after applying the steps and methods discussed in the previous chapter. Furthermore, the project structure is outlined, the tools used in development, and the system's performance. Any limitations of the system are discussed here.

4.2 PROJECT STRUCTURE

Below is a representation of the project structure as it appears in the implementation, directories are in bold.

```
display
      fonts
            stroke_dimension.ttf
      images
            bB.png
            bK.png
            bN.png
            bP.png
            bQ.png
            bR.png
            btile.png
            wB.png
            wK.png
            wN.png
            wP.png
            wQ.png
            wR.png
            wtile.png
      __init__.py
      board.py
      display.py
      fenparser.py
      pieces.py
evaluate
      black.eval
      white.eval
mvp
      loci.py
pgns
      ...
ai.py
eval.py
inout.py
run.py
```

The project structure shows how the implementation is broken into folders and modules, grouped by common function.

The outermost project level gives an overview of the system; **display**, **evaluate**, **mvp**, **pgns**, and four major python modules. These modules each control a major part of the system's function. The **display** directory houses the display module. The **evaluate** directory contains the evaluation configuration files. The **mvp** directory houses a standalone single file MVP that plays chess against a human via UCI commands. Its interface is the command line. Lastly, the **pgns** directory stores the output of the PGN engine.

ai.py defines the generic AI engine. Minimax with fixed depth and α-β pruning is implemented within. The depth is fixed arbitrarily at 5. At the end of each search a function **evaluateBoard()** belonging to **eval.py** is called. The AI engine executes all the functions of the evaluation engine.

eval.py defines the evaluation functions of Black and White based on the piece-value tables defined in the **evaluate** directory; **black.eval** and **white.eval**, varying these files results in playing behavior change in the respective players.

inout.py is the engine's input/output system. It deals with reading the evaluation files and writing output from the PGN engine.

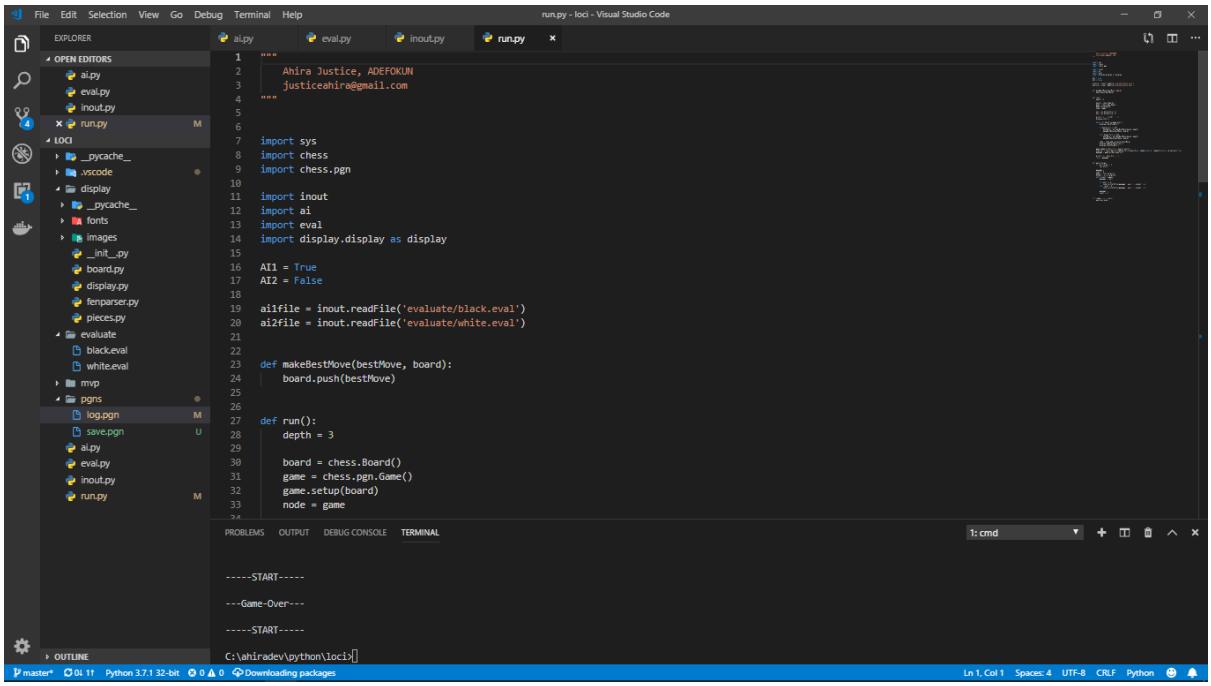

Figure 7: Development environment

run.py is the user's interface to the system. It receives user input from the command line, formats the given commands, and rejects invalid input. It is also responsible for executing the AI engine and the display module.

4.3 OUTPUT

For this system, there are two kinds of output, PGN output and display output. Examples are presented below.

4.3.1 PGN OUTPUT

```
1. g4 Nc6 2. d3 Nf6 3. f3 Nxg4 4. a4 Nxh2 5. b4 Nxf1 6. e3 Nxe3 7. Kd2 Nxd1 8. a5
Nxb4 9. Ra2 Nxa2 10. Bb2 Nxb2 11. Ke3 Nxd3 12. a6 bxa6 13. Rh2 Nab4 14. Rh1 Nxc2+
15. Kd2 Nd4 16. Kc3 Nxf3 17. Rh2 Nxh2 18. Kb3 Nf3 19. Ka3 Nxg1 20. Kb3 Nf3 21. Ka3
c5 22. Kb3 d5 23. Ka3 e5 24. Kb3 Bf5 25. Ka3 Nd4 26. Ka2 Bd6 27. Ka1 O-O 28. Ka2
Qf6 29. Ka3 Rfb8 30. Ka4 Rxb1 31. Ka3 Rb2 32. Ka4 Rab8 33. Ka3 R8b3+ 34. Ka4 h5
35. Ka5 Kh7 36. Ka4 c4 37. Ka5 Bc5 38. Ka4 e4 39. Ka5 Ne5 40. Ka4 Qg6 41. Ka5 h4
42. Ka4 h3 43. Ka5 h2 44. Ka4 h1=R 45. Ka5 Rhb1 46. Ka4 Rb4+ 47. Ka3 R4b3+ 48. Ka4
Rb4+ 49. Ka3 R4b3+ 50. Ka4 Rb4+ 51. Ka3 R4b3+ 52. Ka4 Rb4+ 53. Ka3 R4b3+ 54. Ka4
1/2-1/2
```

The PGN output presented above describes a single game.

4.3.2 DISPLAY OUTPUT

This is the display windows that lets users visualize the game's moves as they occur. It also represents the "thinking" time before each move.

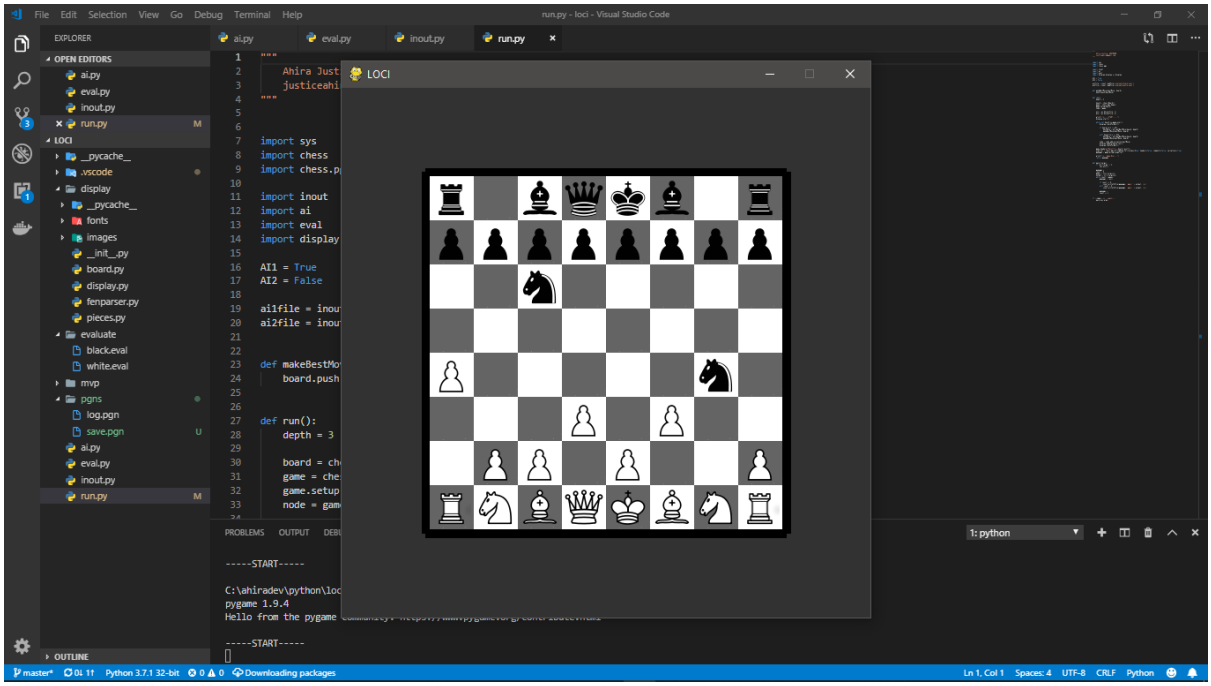

Figure 8: Working display module

4.4 TOOLS

All of the tools described in the previous chapter were used. This section details the ways in which they were used.

4.4.1 IDE – VISUAL STUDIO CODE

Visual Studio Code is a source-code editor developed by Microsoft for Windows, Linux and macOS. It includes support for debugging, embedded Git control, syntax highlighting, intelligent code completion, snippets, and code refactoring. It is also customizable, so users can change the editor's theme, keyboard shortcuts, and preferences. The source code is free and open source and released under the permissive MIT License.

4.4.2 VERSION CONTROL – GIT

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. Git is easy to learn and has a tiny footprint with lightning fast performance. It outclasses supply chain management (SCM) tools like Subversion, CVS, Perforce, and ClearCase with features like cheap local branching, convenient staging areas, and multiple workflows.

4.4.3 PYTHON

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python has a design philosophy that emphasizes code readability, notably using significant whitespace. It provides constructs that enable clear programming on both small and large scales.

Details about why python is suitable have been provided in the previous chapter.

4.4.4 PYTHON-CHESS

Python-chess is a pure Python chess library with move generation and validation, PGN parsing and writing, Polyglot opening book reading, Gaviota tablebase probing, Syzygy tablebase probing and UCI/XBoard engine communication.

4.5 LIMITATIONS

Here we consider the few limitations of the system. There are a lot of heuristics considerations not included in the implementation of this system. This is because heuristics is domain specific knowledge and subject to dispute. The researcher is free to modify the system to include or subtract heuristics considerations from the system. Necessary changes are to be made in the **eval.py** module and **evaluate** directory.

CHAPTER FIVE

CONCLUSION

5.1 SUMMARY

In this project, we explored a new approach to chess engines and how they can be customized by researchers to further research into computational chess and AI. The deduced considerations have been informative and instructive. Standards of computational chess have been applied to this system and the result is a system that allows hands on learning about these principles.

5.2 RECOMMENDATIONS

I make a few recommendations for future work. Firstly, the study of this system could inform a better, more comprehensive system that incorporates better search and evaluation, something researchers would find useful for more in-depth analysis of the evaluation function and strength of engines. Also, the system could be expanded to include human players and the results are then studied, on a compare and contrast basis. Lastly, the system could be expanded to allow UCI chess play. This would allow interface between more accomplished chess engines and a study of the results of changes made to them.

5.3 CONCLUSION

Either as a hobbyist or professional, I hope all who examine this project find their horizons a little broadened and the better for it. Investigating chess computationally has always paved the way for astounding achievements in the field of Artificial Intelligence. Computational chess has grown in tandem with the most powerful computers, pushing the bounds of what humanity ever imagined possible. May this spirit stay alive.

APPENDIX

These two modules define the algorithmic core of the project, the basis for chess AI.

ai.py

```
import sys
import chess
import eval



class AI:
    def __init__(self, evalfile, state):
        self.state = state
        self.eval = eval.AIEval(evalfile, state)



    def getLegalMoves(self, board):
        assert type(board) is chess.Board, 'Incorrect object type: %s' %
(board.__class__)

        moves = []
        for move in board.legal_moves:
            moves.append(move)

        return moves



    def minimax(self, depth, board, alpha, beta, isMaximisingPlayer):
        assert type(board) is chess.Board, 'Incorrect object type: %s' %
(board.__class__)

        if depth == 0:
            return -self.eval.evaluateBoard(board)

        newGameMoves = self.getLegalMoves(board)
```

```
        if isMaximisingPlayer is True:
            bestMove = -9999
            for newGameMove in newGameMoves:
                board.push(newGameMove)
                bestMove = max(bestMove, self.minimax(depth - 1, board, alpha,
beta, not isMaximisingPlayer))
                board.pop()
                alpha = max(alpha, bestMove)

                if beta <= alpha:
                    return bestMove

            return bestMove

        elif isMaximisingPlayer is False:
            bestMove = 9999
            for newGameMove in newGameMoves:
                board.push(newGameMove)
                bestMove = min(bestMove, self.minimax(depth - 1, board, alpha,
beta, not isMaximisingPlayer))
                board.pop()
                beta = min(beta, bestMove)

                if beta <= alpha:
                    return bestMove

            return bestMove



    def minimaxRoot(self, depth, board, isMaximisingPlayer):
        assert type(board) is chess.Board, 'Incorrect object type: %s' %
(board.__class__)

        newGameMoves = self.getLegalMoves(board)
        bestMove = -9999
        bestMoveFound = 0

        for newGameMove in newGameMoves:
            board.push(newGameMove)
```

```
            value = self.minimax(depth-1, board, -10000, 10000, not
isMaximisingPlayer)
            board.pop()

            if value >= bestMove:
                bestMove = value
                bestMoveFound = newGameMove

        return bestMoveFound



    def getBestMove(self, board, depth):
        assert type(board) is chess.Board, 'Incorrect object type: %s' %
(board.__class__)

        if self.state == 1:
            bestMove = self.minimaxRoot(depth, board, True)
        elif self.state == 2:
            bestMove = self.minimaxRoot(depth, board, False)

        return bestMove
```

eval.py

```
import sys
import chess



class AIEval:
    def __init__(self, evalfile, state):
        if state == 1:
            self.pawnEvalWhite = evalfile[0]
            self.pawnEvalBlack = self.reverseList(self.pawnEvalWhite)

            self.bishopEvalWhite = evalfile[1]
            self.bishopEvalBlack = self.reverseList(self.bishopEvalWhite)

            self.rookEvalWhite = evalfile[2]
```

```
            self.rookEvalBlack = self.reverseList(self.rookEvalWhite)

            self.knightEval = evalfile[3]

            self.evalQueen = evalfile[4]

            self.kingEvalWhite = evalfile[5]
            self.kingEvalBlack = self.reverseList(self.kingEvalWhite)

        elif state == 2:
            self.pawnEvalBlack = evalfile[0]
            self.pawnEvalWhite = self.reverseList(self.pawnEvalBlack)

            self.bishopEvalBlack = evalfile[1]
            self.bishopEvalWhite = self.reverseList(self.bishopEvalBlack)

            self.rookEvalBlack = evalfile[2]
            self.rookEvalWhite = self.reverseList(self.rookEvalBlack)

            self.knightEval = evalfile[3]

            self.evalQueen = evalfile[4]

            self.kingEvalBlack = evalfile[5]
            self.kingEvalWhite = self.reverseList(self.kingEvalBlack)



    def reverseList(self, _list):
        return _list[::-1]



    def getAbsoluteValue(self, piece, piece_color, x, y):
        if (piece.piece_type == chess.PAWN):
            return 10 + (self.pawnEvalWhite[y][x] if piece_color is chess.WHITE
else self.pawnEvalBlack[y][x])
        elif (piece.piece_type == chess.ROOK):
            return 50 + (self.rookEvalWhite[y][x] if piece_color is chess.WHITE
else self.rookEvalBlack[y][x])
        elif (piece.piece_type == chess.KNIGHT):
```

```
            return 50 + self.knightEval[y][x]
        elif (piece.piece_type == chess.BISHOP):
            return 50 + (self.bishopEvalWhite[y][x] if piece_color is chess.WHITE
else self.bishopEvalBlack[y][x])
        elif (piece.piece_type == chess.QUEEN):
            return 50 + self.evalQueen[y][x]
        elif (piece.piece_type == chess.KING):
            return 50 + (self.kingEvalWhite[y][x] if piece_color is chess.WHITE
else self.kingEvalBlack[y][x])
        else:
            assert "Unknown piece type: " + piece.piece_type
            sys.exit()



    def getPieceValue(self, piece, x, y):
        if (piece is None):
            return 0

        absoluteValue = self.getAbsoluteValue(piece, piece.color, x, y)
        return absoluteValue if piece.color is chess.WHITE else -absoluteValue



    def evaluateBoard(self, board):
        assert type(board) is chess.Board, 'Incorrect object type: %s' %
(board.__class__)

        totalEvaluation = 0

        for i in list(range(8)):
            for j in list(range(8)):
                totalEvaluation +=
self.getPieceValue(board.piece_at(chess.square(j, i)), i, j)

        return totalEvaluation
```

REFERENCES

Ahle, T. D. (2017). *Sunfish*. Retrieved from GitHub: https://github.com/thomasahle/sunfish

Allis, L. V. (1994). Searching for Solutions in Games and Artificial Intelligence (1994). *Ponsen & Looijen*.

Bernhardsson, E. (2017). *Deep Pink*. Retrieved from Github: https://github.com/erikbern/deep-pink

Edwards, S. J. (1993). *Standard: Portable Game Notation Specification and Implementation Guide.* Retrieved from http://www.saremba.de/chessgml/standards/pgn/pgn-complete.htm

Eschner, K. (2017, July 20). *Debunking the Mechanical Turk Helped Set Edgar Allan Poe on the Path to Mystery Writing*. Retrieved from Smithsonian.com: https://www.smithsonianmag.com/smart-news/debunking-mechanical-turk-helped-set-edgar-allan-poe-path-mystery-writing-180964059/

Fiekas, N. (2019, January). *python-chess*. Retrieved from https://pypi.org/project/python-chess/

Greenblatt, R., Eastlake III, D., & Crocker, S. (1969). *The Greenblatt Chess Program.* Massachusetts Institute of Technology. Retrieved April 6, 2016, from https://dspace.mit.edu/bitstream/handle/1721.1/6176/AIM-174.pdf

Home - Stockfish - Open source chess engine. (2015). Retrieved from Stockfish.org: http://stockfishchess.org

Lai, M. (2015). *Giraffe: Using Deep Reinforcement Learning to Play Chess.* Imperial College London, Department of Computing, London.

Levy, D. (1983). *Computer Gamesmanship: The Complete Guide to Creating and Structuring Intelligent Games Programs.* New York, New York, United States of America: Simon & Schuster, Inc.

Neumann, J. v. (1928). Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 295-320.

Newell, A., & Simon, H. A. (1976). Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 113-126.

Rodriguez, J. (2017, February 21). *Game Theory and Artificial Intelligence.* Retrieved from Medium: https://medium.com/@jrodthoughts/game-theory-and-artificial-intelligence-ee8a6b6eff54

Russell, S., & Norvig, P. (2019). *Artificial Intelligence: A Modern Approach* (3rd ed.). Pearson.

Shannon, C. E. (1950). XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 256-275.

Shinners, P., Dudfield, R., Appen, M. v., & Pendleton, B. (2018, July). *Pygame*. Retrieved from https://pypi.org/project/Pygame/

Soltis, A. E. (2017, September 26). Chess. *Encyclopædia Britannica*. Encyclopædia Britannica, inc. Retrieved February 12, 2019, from https://www.britannica.com/topic/chess

Sommerville, I. (2011). *Software Engineering* (9th ed.). United States of America: Pearson Education, Inc.

Valacich, J. S., George, J. F., & Hoffer, J. A. (2012). *Essentials of Systems Analysis and Design.* Pearson Education, Inc.