“ALEXANDRU-IOAN CUZA” UNIVERSITY OF IASI

**FACULTY OF COMPUTER SCIENCE**

****

BACHELOR THESIS

**Artificial Intelligence techniques for Chess**

Author:

Carol-Sebastian Rameder

July 2021

Scientific supervisor:

Conf. dr. Mihaela Breabăn

**Avizat,**

**Îndrumător Lucrare**

Titlul, Numele şi prenumele \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Data \_\_\_\_\_\_\_\_\_\_\_\_ Semnătura \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**DECLARAŢIE**

**privind originalitatea conţinutului lucrării de licenţă/disertaţie**

Subsemntatul **Rameder Carol Sebastian**, domiciliul în **România, jud. Iași, mun. Iași, Bulevardul Independenței, nr. 28, bl. Y4, ap.48**, născut la data de **20 iunie 1999**, identificat prin CNP **1990620226756**, absolvent al Universităţii „Alexandru Ioan Cuza” din Iaşi, **Facultatea de informatică** specializarea **informatică**, promoţia 2021, declar pe propria răspundere, cunoscând consecinţele falsului în declaraţii în sensul art. 326 din Noul Cod Penal şi dispoziţiile Legii Educaţiei Naţionale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licenţă cu titlul **Artificial Intelligence techniques for Chess** elaborată sub îndrumarea doamnei Conf. dr. Mihaela Breabăn, pe care urmează să o susţin în faţa comisiei este originală, îmi aparţine şi îmi asum conţinutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licenţă/disertaţie să fie verificată prin orice modalitate legală pentru confirmarea originalităţii, consimţind inclusiv la introducerea conţinutului său într-o bază de date în acest scop.

Am luat la cunoştinţă despre faptul că este interzisă comercializarea de lucrări ştiinţifice in vederea facilitării fasificării de către cumpărător a calităţii de autor al unei lucrări de licenţă, de diploma sau de disertaţie şi în acest sens, declar pe proprie răspundere că lucrarea de faţă nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Dată azi, ………………………… Semnătură student …………………………

**Declaratie de consimțământ**

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „Titlul complet al lucrării”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică. De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent Carol Sebastian Rameder

Iași, data Semnătura:

Table of contents

[1 Introduction 6](#_Toc75012301)

[1.1 Summary 7](#_Toc75012302)

[1.2 Motivation 7](#_Toc75012303)

[1.3 Contributions 8](#_Toc75012304)

[2 The implementation of Chess game elements 8](#_Toc75012305)

[2.1 Python and Classes 8](#_Toc75012306)

[2.2 Chess rules 9](#_Toc75012307)

[2.3 Overview Diagram 10](#_Toc75012308)

[2.4 The Engine 10](#_Toc75012309)

[2.5 Move Class 11](#_Toc75012310)

[2.6 Game state 11](#_Toc75012311)

[2.7 The implemenation of rules and constraints 12](#_Toc75012312)

[2.7.1 Piece movement 13](#_Toc75012313)

[2.7.2 Check constraint and pins 14](#_Toc75012314)

[2.7.3 Game endings 15](#_Toc75012315)

[2.8 Move making 16](#_Toc75012316)

[2.9 Possible imporvements 16](#_Toc75012317)

[2.9.1 Bitboard 16](#_Toc75012318)

[2.9.2 Numpy 17](#_Toc75012319)

[3 The Artificial Intelligence approach 18](#_Toc75012320)

[3.1 Defining the problem 18](#_Toc75012321)

[3.2 Game tree search 19](#_Toc75012322)

[3.2.1 Minimax 20](#_Toc75012323)

[3.2.2 Negamax 20](#_Toc75012324)

[3.3 Heuristic evaluation function 21](#_Toc75012325)

[3.4 Optimization 22](#_Toc75012326)

[3.4.1 Alpha – Beta pruning 23](#_Toc75012327)

[3.4.2 The Transposition Table and Zobrist Hashing 23](#_Toc75012328)

[3.4.3 Move ordering 24](#_Toc75012329)

[3.4.4 Horizon effect and Quiescence search 25](#_Toc75012330)

[3.5 General structure of the AI 25](#_Toc75012331)

[4 Game behaviour examples & stats 26](#_Toc75012332)

[4.1 Positional development 26](#_Toc75012333)

[4.2 Force mate 26](#_Toc75012334)

[4.3 Trapping a more valuable piece 26](#_Toc75012335)

[4.4 End game situations 26](#_Toc75012336)

[5 State of the art 26](#_Toc75012337)

[5.1 Monte Carlo Tree Search for chess 26](#_Toc75012338)

[5.2 AlphaZero 26](#_Toc75012339)

[6 Conclusions 26](#_Toc75012340)

[7 Bibliography 27](#_Toc75012341)

[8 Appendix with code 27](#_Toc75012342)

# Introduction

My bachelor thesis consists of an overview of Artificial Intelligence techniques used in Chess programming and the presentation of the results of a self-developed program, capable of showing optimal and intelligent behaviour in the environment of the chess game. Moreover, comparing its performance with the theoretical analysis of the latest trends in the field and state-of-the-art chess engines led to the conclusion drawn from the study. Therefore, the approach of the given subject contains both theoretical and practical aspects.

Besides coding theoretical aspects learned from various courses during the computer science study, the general objective of this thesis is getting an in-depth view about techniques and tendencies of the Artificial Intelligence domain by facing real challenges with academic writing and selecting suitable literature for a limited subject area.

AI for Chess is one of the oldest and most studied fields in this domain and computer’s performance has grown considerably with the transition from game tree search to Deep Reinforcement Learning. Nowadays, AI algorithms have reached a superhuman level, as the world champion was firstly defeated by a computer in 1997 and are frequently used not only for training for high-level tournaments, but also for cheat detection in online games.

“*Over the years, chess has proven to be a fertile ground for ideas and techniques that have spread to other areas of AI . These include database enumeration techniques, chunking, search techniques (minimax, alpha-beta, iterative deepening), and the utility of information. Considering the lack of funding for chess, it is significant that it has produced so many results. Chess has been fertile because it provides a complex reasoning problem from a simple domain with a builtin performance criteria. The simple domain permits research to progress with little initial overhead. Having a hostile opponent adds complexity to the reasoning. In many domains (natural language understanding comes to mind), progress can be hindered by lack of performance criteria - it can be hard to tell whether the latest thesis is an improvement on the current state of the art, Chess provides precise answers to performance questions.* (...)

*Research into artificial intelligence using chess as the application domain has produced several important contributions to AI :*

* *The effectiveness of brute-force search. Chess has clearly demonstrated that simple, brute-force approaches should not be quickly discarded.*
* *Iterative search. Some of the ideas developed for alpha-beta search, iterative deepening in particular, are applicable to other search domains and games such as go, shogi and tic-tac-toe.”* (1)

## Summary

The thesis is structured in four chapters which ilustrates all needed details in cronological order:

* **Chess game elements** provides information for linking the programming part to the Chess game environment. It focuses on implementation details and prepares the data for applying further algorithms
* **The Artificial Intelligence approach** explains how the AI technique is used in oreder to make the computer simmulate smart behaviour related to the game
* **Game behaviour and stats** shows the results obtained in real game situations with conclusive examples and performance measurements
* **State of the art** brings a theoretical analysis of most the latest and most performant engines and shows a performance comparison with early used techniques that prepares the conclusion

## Motivation

I choose this specific topic for my final thesis as I saw the great potential of improving both my practical programming skills and academic research and writing by approaching a merger of two passions of mine. I have been playing chess, go to training and participate in championships for a long period of time in my childhood, as well as I recently got interested in the AI field and how computers can show intelligent human-like behaviour in different situations. Moreover, getting insights into how a computer can become smarter than people in a specific field and developing a program that can respond and adapt to a chess game is a personal fulfilment. When I first started programming, I truly considered this too hard to understand, but working on this thesis gave me the necessary steps to make it possible and bring my contribution.

## Contributions

1. Implementing the game of Chess with the user interface, playable for two players, with board representantion and move validation
2. Finding and adapting an AI algorithm for the scenerio of playing against the computer and optimize it for a greater response time and strategy
3. Analizing the results and obtained performance
4. Getting overview on current, most powerfull chess engines and in-depth knowledge in how they work and what theoretical aspects and algorithms are required for best performance
5. Making observation on the evolution in this domain drawing conclusions by comparing the researched techniques

# The implementation of Chess game elements

In this section, there will be presented details about data structures and classes used to ensure the data and functions call flow. These combined toghether form the core of the program, the chess engine.

## Python and Classes

The whole structure of the game is implemented in Python 3.9 as its widely known features make it suitable for this kind of project. It is a high-level programming language without strict emphasis on syntax and background behaviour, therefore there are considerably fewer lines of code needed to perform a complex task than other major programming languages such as C/C++ and Java. It is way more efficient in Python to implement functions with multiple data types such as dictionaries strings list and arrays, considering the fact that the compiler does not need to know any kind of data type and automatically assigns them during the execution. In my case, this simplicity made the debugging process way easier and helped me to focus more on solving the problem and algorithms.

In terms of disadvantages, the slow speed in execution is the main problem of a program written in Python. The interpreter has to do extra work while executing the code because the interpreter has to dynamically interpret the instructions. That is the reason why many optimizations are required in order to reach a good level of performance and search depth, as a consequence of the constant trade-off between speed and intelligence, as we will below.

The whole program is structured using the Object-oriented approach based on classes with their specific attributes and methods. This has been done to better organize the functionality structure into small parts assembling the big puzzle, in order to make it easy to read, change and debug.

## Chess rules

|  |  |
| --- | --- |
| Fig. 1 – Initial Setup | Fig. 2 – Enpassant capturing (2) |

The pieces are always placed as in the initial setup image and White always starts. Despite basic piece movement and their implementation from 2.6.1 (including castling), there are special rules like pawn promotion, enpassant, pins, checks, checkmate and stale mate:

* [Promotion] Pawns that are advanced to the 8th line for white and 1st line for black are transformed, by player’s choice in a figure (Queen, Knight, Rook or Bishop)
* [Enpassant] When a pawn advances two squares from its original square and ends the turn adjacent to a pawn of the opponent's on the same rank, it may be captured by that pawn of the opponent's, as if it had moved only one square forward. This capture is only legal on the opponent's next move immediately following the first pawn's advance. *Fig. 2* demonstrates an instance of this.
* [Checks and Pins] A king is in check when it is under attack by at least one enemy piece. A piece unable to move because it would place its own king in check (it is pinned against its own king) may still deliver check to the opposing player. It is illegal to make a move that places or leaves one's king in check. The possible ways to get out of check are moving the King, capturing the checking piece or blocking the check by placing a piece between the king and the opponent's threatening piece.
* [Checkmate] If a player's king is placed in check and there is no legal move that player can make to escape check, then the king is said to be checkmated, the game ends, and that player loses. Unlike other pieces, capturing the opponent's king is not allowed.
* [Stalemate] The game is automatically a draw if the player to move is not in check and has no legal move. This situation is called a stalemate.

## Overview Diagram

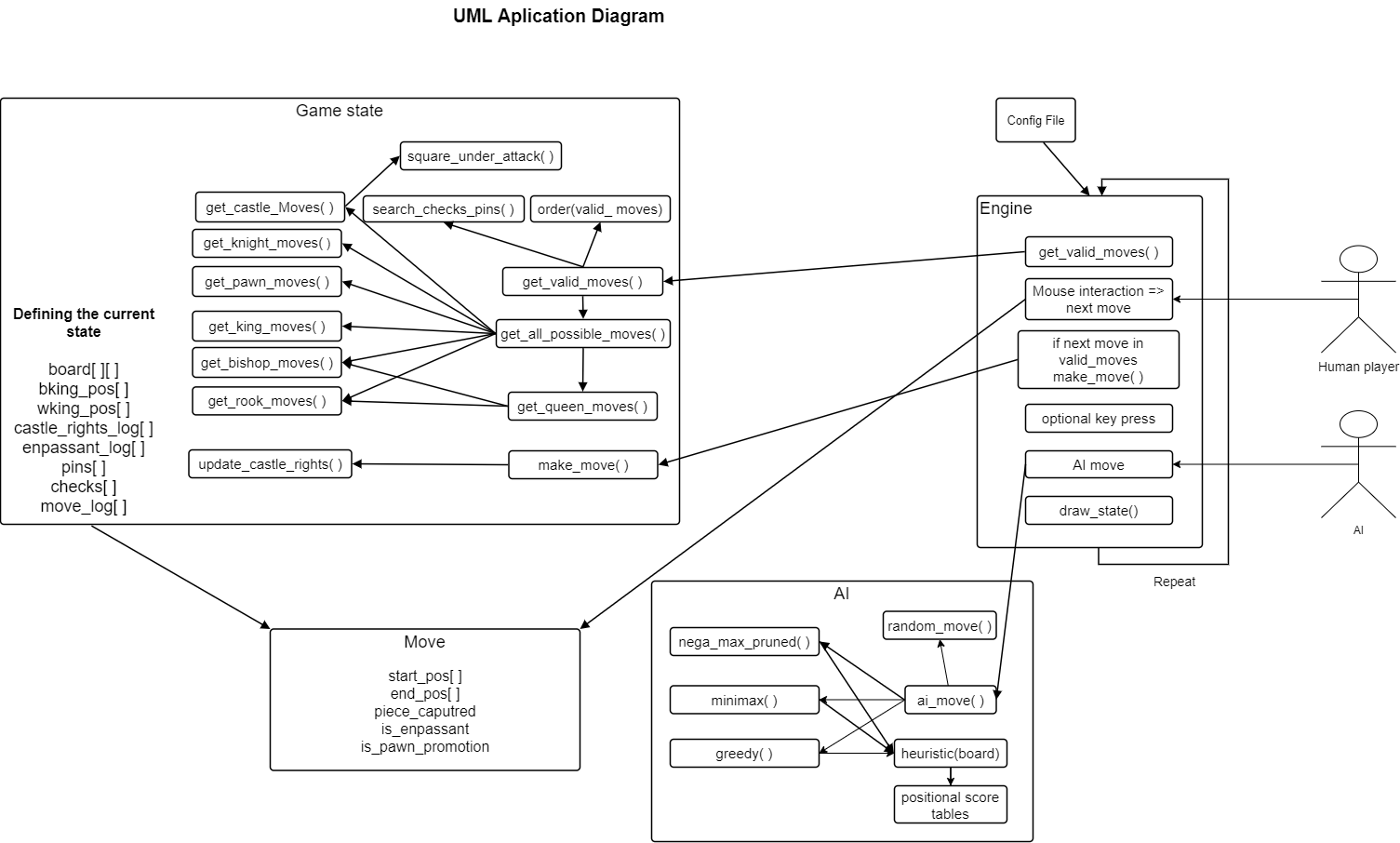


Fig. 3 – Overview Diagram

## The Engine

The Engine Class is the main core of the program which synchronize the human interaction and User Interface with the Chess Game State representation and the AI logic.

First of all, player type is taken from the configuration file for both White and Black, it can either be ‘human’ or ‘ai’. Then, the depth of the search tree and the AI technique used is set on the next two lines. The algorithm can use pruned and unpruned exhaustive search functions and the depth directly impacts the speed and performance, because it represents the number of forward moves computed for both players, in order to make the best decision (move) on each current step.

Another functionality of the Engine is to create the User Interface. It simply loads all the prepared images from the folder for the board and each type of piece. After each move made by the human player or the computer, the state is updated and displayed accordingly. By doing so, the execution is significantly lowered, because large files as photos are loaded in an array only once in the beginning.

It further listens to mouse events made on the board window and transforms them into moves with two coordinates for the start and the end. There are also keyboard options, by pressing the ‘Z’ key the user can undo the last move and, by pressing ‘R’, the board is reset to the initial setup. (*Fig.1*)

The Engine calls the specific function in the Game State Class which provides all valid moves from the current board and if the move received from the player is in that set, the “make\_move” function is called and the state of the game changes accordingly. Otherwise, if the move is not valid, the engine listens to another move from the player.

Finally, the Engine ensures that the player type (human or ai) is assigned to its pieces and creates the synchronization between White moves and Black moves. The game loop is simulated with White always starting and repeated until a final game state is reached, either Stalemate or Checkmate.

## Move Class

The Move class represents a key functionality of the project and is used for representing a chess action in a raw form, without any further verification or validation. It is only formed from start and end postion of the move and it’s type, whether it is a standard, capturing, a castle, promotion or enpassant move.

It is created by the Game State and returned in the list of valid moves. It is further analyzed by the AI logic or compared with the actions given by the player.

## Game state

In the program, there is a class especially created for storing the information of the current game state and all necesary methods for further use in the engine and AI move finder. It ensures that all chess rules are optimally verified and the restrictions regarding piece movement, castling, checks, enpassant capturing, pawn promotions and game ending situations are not violated by both the player and the AI.

The properties of game\_state class, initiated with the built-in \_init\_ function are:

* board[ ][ ] – two dimensional array with 8 columns and 8 rows, simmulating a chess table, which stores “--“ for an empty square or the piece shortened name such as “wp” for a white pawn or “bk” for a black rook for example. In the beginning, the Chess initial setup is represented.
* white and black king position – two tuples with the coordinates of each king such as (7,6). It is used for decideing whch moves are legal by computing checks and pins, castle rights and king moves.
* castle\_rights\_log[ ] – it is updated after the move of the rook or king which changes the castle rights of a player. Contains boolean values for king and queen side for both white and black
* enpassant\_log[ ] – stores and updates after each move of a pawn by checking the condition stated in chess rules above
* checks[ ] and pins[ ] – contains the checks and the pins on a given position. A pin is considered a piece that is not alowed to be moved from the direction of the attacker, as it leaves the king in check. The array contains the direction of the pin and the position of the pinned piece or the direction of the check and the attacking piece: (piece\_row, piece\_column, d[0], d[1]). Both arrays are used for move validation.
* move\_log[ ] – was previosly used for searching valid moves by simulating the capture of the king. In the last version it is used for undoing the last move by pressing the ‘R’ key

## The implemenation of rules and constraints

The chess rules and piece movement restrictions are verified in the Game State Class, as the valid moves from a given position can be obtained easily in the Engine or AI by calling the get\_valid\_moves method that expands and make calls to the other functions. It basically brings Chess logic to the program not allowing neither human nor the ai to make mistakes or violating the rules.

Firstly, get\_valid\_moves takes all possible moves when considering chess piece movement and capturing rules, without considering checks and pins that eliminates some possible moves. It iterates through all squares on the board, and call the specific function of every piece when found, as it follows.

### Piece movement

* get\_pawn\_moves() – A pawn moves straight forward one square if that square is vacant. If it has not yet moved, a pawn also has the option of moving two squares straight forward, provided both squares are vacant. Pawns cannot move backwards. A pawn, unlike other pieces, captures differently from how it moves. A pawn can capture[[1]](#footnote-1) on either of the two squares diagonally in front of the pawn (but cannot move to those squares if they are vacant). The pawn is also involved in the two special moves en passant and promotion. (2.2) The function returns all these moves.
* get\_knight\_moves() – The Knight can move and capture in an “L” pattern (two squares horizontally then one square vertically, or moving one square horizontally then two squares vertically) and can not be blocked, as it “jumps”.
* get\_bishop\_moves() – The Bishop can move on all the 4 diagonals and can capture only the first enemy piece met in any of those directions
* get\_king\_moves() – The king can move and capture exactly one square horizontally, vertically, or diagonally. A special move with the king known as castling is allowed only once per player, per game. In that case, get\_castle\_moves() is called. You are not allowed to place your king near the other king.
* get\_castle\_moves() – Castling consists of moving the king two squares towards a rook, then placing the rook on the other side of the king, adjacent to it. Castling is only permissible if all of the following conditions hold:
* The king and rook involved in castling must not have previously moved, so it can only be queenside or kingside
* There must be no pieces between the king and the rook;
* The king may not currently be in check, nor may the king pass through or end up in a square that is under attack by an enemy piece (though the rook is permitted to be under attack and to pass over an attacked square);
* get\_rook\_moves() – The Rook can move vertically and horizontally and can capture only the first enemy piece met in any of those directions
* get\_queen\_moves() – The Queen can move vertically, horizontally and diagonally and can capture only the first enemy piece met in any of those directions.

### Check constraint and pins

There are two methods possible to use for validating the moves generated as above. The first one is probably more intuitive and logic, but also considerably more expensive in terms of memory used and especially execution time, which turned to be one of the most important aspect of the algorithm.

In the first method:

* all possible moves are generated
* each move is made (simulated in the background and then reversed)
* generate all possible moves of the opponent
* for each one of them, if it “captures” the king, it means that the player left or moved the king in check last time and made an invalid move

The second method and the one actualy used in the program:

* On the current board, start iterating in all nine directions from the position of the king
* if the position of an ally piece is reached

- if it is the first one it becomes a possible pin

- if is the second one, the search in that direction stops

* if the position of an enemy piece is reached
* we check using the rules for every type of piece if it attacks
* if it does not attack the king the loop ends for that direction
* if it attacks the king the possible pin is added to the array of pins or if we do not have a possible pin, the king is in check

Comparing these two methods, in terms of complexity, the first one simulates two moves ahead on every move made in reality, only for deciding if it was legal or not. That means 900 positions in average, for 30 possible moves on the current board. In contrast, the second method only iterates through 28 squares in the worst case, when the king is located in the centre.

### Game endings

The checkmate si decided by counting the possible moves and verifying if the King is in check. If there are no possible moves and the King is attacked by an enemy piece, then the game ends and the attacker wins. Otherwise, if there are no possible moves returned by get\_valid\_moves(), but the king is not attacked, that is a Stalemate and the game ends with a draw.

In both cases, no other moves are allowed to be made and a message is displayed in the console.

|  |  |
| --- | --- |
| Fig. 4 – Stalemate (2) | Fig. 5 - Checkmate (2) |

## Move making

|  |  |
| --- | --- |
| Fig. 6 - UI board before the move | Fig. 7 - UI board after the move |

|  |  |
| --- | --- |
| Fig. 8 - Board state before the move | Fig. 9 - Board state after the move |

As it can be seen in the example, the make\_move() method from the Game State Class makes the former position of the moved pieced empty (“--” value) and updates the value on landing spot, from captured pieced “wb” to “bq”.

## Possible imporvements + sstronger heuristcs, end game table , openiing database

The logic and implemention of the program can considerably impact the speed of the chess engine and further the performance of the AI. Both ideas presented bring a low-level approach for the representation and code.

### Bitboard

*“Bitwise methods for programming games centre around the concept of the bitboard. This is a data structure designed for efficiently encoding game boards as sets of bits, first used for computer chess in the 1950s (Frey, 1977). Rather than allocating an integer for each board cell to store the value of any piece there, each cell is assigned a bit indicating the presence or absence of a piece (or pattern) there, requiring only a fraction of the memory. For example, the cells of an 8×8 chess board conveniently pack into a single 64-bit long integer.*

*Bitboards allow common game-related operations to be performed using fast bitwise manipulations. Pepicelli (2005) lists three main advantages of using bitboards.*

* *Memory Usage: Bitboards encode the board state more efficiently than integer-per-cell encodings.*
* *Efficient Operation: Read and write operations can be performed efficiently using bitwise operations.*
* *Bitwise-Parallel Operation: Bitwise operations can be applied to all board cells simultaneously.*

*Efficient memory usage can be beneficial if it allows more operations to be performed from the (much faster) registers or cache. However, the potential for bitwise-parallel operation can also yield significant performance improvements. This means that game-specific calculations such as movement or win tests need only be applied once over the entire board in a bitwise-parallel manner, rather than individually for each cell or piece. Such operations are typically stateless as they operate equally over all cells with no prior knowledge about the board state, but can sometimes be optimised with the inclusion of state information.”* (3)

### Numpy

To begin with, standard Python is an interpreted language with high-level syntax, that it is, in general, slower than C-like programming languages, which are compiled and much faster.

Numpy is a library designed for Python that brings functionality for arrays and algebraic

operations. It performs its complex computation using a well-optimized C code, therefore the core of Numpy makes it possible to incorporate the flexibility of Python with the speed of compiled code.

Using Numpy arrays for the game representation and engine will significantly improve the speed of the program on execution or will facilitate the exploration of a deeper search tree in the same amount of time.

# The Artificial Intelligence approach

Chess is one of the most complex and strategic games and the Artificial Intelligence Agent needs to forward plan as many states as possible and efficient to reach a favourable game-ending state, such as a checkmate from a favourable material position or stalemate from a position with far more material than the opponent. The game is completely deterministic because there is no randomness involved in winning the game and the search space is very large, as there are approximately 10120 possible games of chess[[2]](#footnote-2), and is not be computational in a reasonable time.

The purpose of the program is to show human-like intelligence and analyze the chess environment and apply its strategy to shrink search space in its favour. The method used for achieving this goal is finding the best move based on chess generally known pre-sets, by dynamically iterating through the game tree search as it follows.

## Defining the problem

The program basicaly gives a solution to a problem which can be organized with the standard AI structure:

* describe a state : 2.6
* identify special states and the problem space : 2.7.3
* describe the transition and validate them : 2.7.1 and 2.7.2
* specify a search strategy : 3.2

## Game tree search

|  |  |
| --- | --- |
| Fig. 10 – Game tree (4) | The Initial Setup represents the root node and the possible position are added consequently as a new node after a possible move of a player. Each player turn is located on a new level, on a higher depth as the game advances. Therefore, the branching factor[[3]](#footnote-3) of the tree is represented by the number of possible moves of a player at a certain moment, with a value between 25 and 40 in most cases. The maximum depth is given by the length of the chess game. |

This representation is natural and optimal designed for searching in such a big space as it successfully simulates the rules and logic of the chess game. The game tree can further integrate move comparison and optimizations that can enlarge the amount of relevant position analyzed.

In order to determine the best decision, the depth-first search is used in the tree, because the outcomes from a branch are obtained only in leaf nodes and must be further compared to the other possibilities.

The advantage of this method is that it knows exactly what move is the most optimal, with the given criteria and methods to evaluate each possible branch of the game, because the search is exhaustive.

The disadvantage is that the costs are high and grow exponentially as the depth of the search is increased, thus making it limited. Therefore, further variants of the game could not be explored and the evaluation of the leaf nodes is considered valid and representative for the whole branch until the end of the game. In most cases, the short term outcome computed over the actual path to win can be losing or deceptive in chess.

### Minimax

|  |  |
| --- | --- |
| Fig. 11 - Minimax (5) | We consider the fact that each leaf node has an arbitrary value which represents how favourable the position for both players is, in the sense that it is lower if the reached state is in the advantage of the first player and higher otherwise. (3.3) |

The Minimax is an exhaustive search algorithm mainly used in artificial intelligence decision making in adversarial turn-based games like chess, that uses backtracking and depth-first search. It simulates the succession of the moves that each chess player has to take in order to maximise his chances to win. The algorithm is build on the fundamental idea that each player makes everytime the best move available, which is not humnaly achievable. The agents are placed alternative on the levels in the game tree, starting with who is on turn in the root node and ending if a game final state or maximum declared depth is reached. Each possible move adds a new node as a reachable chess position in an analyzed branch, on the next level in the tree.

The evaluation takes place only on leaf nodes and then backpropagate, up to the initial call (root) as follows. The two players have directly opposing strategies because one of them always chooses and takes the maximum available value from child nodes, while the other one minimizes the score. When the backpropagation of all values from leaf nodes to the root ends, is the point where the decision made by the computer takes place, indicating artificial intelligence.

### Negamax

Negamax with Alpha-Beta Pruning is the algorithm used in the most efficient version of the implemented program. Negamax is a simplified version of the Minimax Search Algorithm that is more suitable for practical use, as it uses all previously presented principles, offering the same results in a more facile way for coding, understanding and debugging. It also can be slightly faster, because it involves fewer comparisons on each step.

The core idea behind this is that minimizing the chances of the opponent to win has exactly the same results as the direct pursuit of self-development and organizing the pieces. In practice, no matter the current level in the recursion, the node will backpropagate the highest available value in child nodes, multiplied by -1. All summed up:

max(a, b) == -min(-a, -b)

Pseudocode: (6)

**function** negamax(node, depth, α, β, color) **is**

**if** depth = 0 **or** node is a terminal node **then**

**return** color × the heuristic value of node

childNodes := generateMoves(node)

childNodes := orderMoves(childNodes)

value := −∞

**foreach** child in childNodes **do**

value := max(value, −negamax(child, depth − 1, −β, −α, −color))

α := max(α, value)

**if** α ≥ β **then**

**break** *(\* cut-off \*)*

**return** value

## Heuristic evaluation function

In reality, there is no certain method to calculate exactly how good or useful is to reach a position during the game, and only an approximation can be appropriate in making a winning decision. Therefore, in a complex game like chess, a heuristic function is used by the computer to evaluate a position during the search and further make a move that will directly lead to that advantageous game state. By doing this, there is a trade between finding the exact optimal move after a complete and precise search and doing this task in the short thinking time assigned to a chess move.

The players have adversarial strategies as they both will try to adjust the value of the heuristic in their favour trying to win the game. The convention is that both players have equal values for the same strategy or piece on the table, but with different signs. Thus, white will try to maximize the value of the heuristic and black will try to minimize it.

The heuristic implemented for the AI Engine represents the core of the logical thinking responsible for finding the most suitable move at each step. It stores all the information available for the computer and directly impacts its strategy through the preset data with chess insights. In the list below, there is the information given to the computer for evaluating positions linked with the consequence in a chess game and visual diagram explanation: (7)

* the material is counted accordingly to the general chess indications with some strategic little changes. A pawn is worth 100 points, a knight = 320, a bishop = 330, a rook = 500, a queen = 900. This encourages the AI not only to keep the material balance, which is crucial in most cases, and try to capture more valuable pieces but also to follow the chess more advanced indications like keeping the bishop pair, not accepting rook and pawn for two minor pieces (R+P < 2N)
* the value for the Knight, Queen or Bishop is multiplied by a higher coefficient the more central it is placed in order to control as many squares as possible. The Bishop has higher values for squares located on the two principal diagonals. (Fig. 12)
* the value of a Pawn is lower on the default squares for those in the centre, as it is strongly indicated to develop those pawns in the opening and control the centre. Moreover, the value of a pawn increases as it is closer to promotion, one of the most powerful moves and winning chess strategies (Fig. 14)
* the King is encouraged to castle or move to a corner, because it is safer there and the possibilities to be in a pin or a check are significantly lower there. Castling is also benefic for rook development and is valued accordingly. In the end of the game, the King has another positional map score that values a more central location like any other attacking piece (Fig. 15)
* the Rook has a higher positional coefficient for the 7th line where it can put pressure on the opponent’s king and undeveloped pawns on their initial square (Fig. 13)
* for the coefficient of black’s pieces, there is a similar inverted map, that will discourage the computer to let, for example, human pawns on their 7th line

|  |  |  |  |
| --- | --- | --- | --- |
| Fig. 12 - Knight Positional Score | Fig. 13 - Rook Positional Score | Fig. 14 - Pawn Positional Score | Fig. 15 - King Positional Score |

## Optimization

The simple Minimax search has many issues regarding the speed and performance of the moves because the number of the analyzed position is too high to be computed and tested in a game for a depth bigger than 3. Therefore, simple and logical moves from a human perspective were not acknowledged by the computer, who could not analyze even two complete moves in advance. There comes the need for optimization of the game tree search logic and structure, which will save time and computation power wasted on irrelevant or repetitive searches and turn it in its favour.

### Alpha – Beta pruning

|  |  |
| --- | --- |
| Fig. 16 - Alpha Beta Pruning (8) | The purpose of this optimization applied to the Minimax algorithm is to l ower the number of irrelevant positions evaluated during the search in each iteration, without affecting the performance of the program by any means. It does all that by “pruning” the move branches that do not impact the final decision made by the engine but slows down the process. |

The addition of this procedure over the naive Minimax is to add an upper and a lower bound (α and β) for the search, that are updated accordingly and define the interval of relevant values in the sub-tree nodes, which should be looked over during the iteration. More precisely, if the minimizing player (black) has a better position with the value β earlier in the tree, on another branch, it does not have to look through all the child nodes of a maximizing node (white) that found by itself a position with the value α > β, and will further only try to maximize that. The last process is irrelevant for the minimizing node, and for the final decision made by the engine. This process also happens with exchanged roles for black and white.

To sum up, alpha-beta pruning is a powerful and mandatory optimization method when working with the Minimax algorithm and, in practice, they work hand in hand.

(..)table pos analyzed comparison

### The Transposition Table and Zobrist Hashing

During the dynamic search in the game tree, there is a large number of positions that are analyzed multiple times on different branches of moves, because you can get to a certain chess state by making a distinct set of moves. A beneficial tool for speeding up the search process is creating a log of positions already evaluated by the heuristic function with the objective to not repeat the call on the heuristic evaluation function and compute the value obtained previously. Here comes the concept of the Transposition Table, which is generated along with the chess game and consists of a Python dictionary with a unique code for each position as the key and the value returned by the heuristic. (Appendx 1)

The Zobrist Hashing method is used for generating a unique 64-bit key for each chess position analyzed by the engine. It uses the XOR binary operation between all occupied squares on the table, differentiating the type and colour of the piece locating there. Thus, it is safe and optimal to use, as it will cause collisions only after evaluating 232 (4 billion) positions. (9)

### Move ordering

As the Minimax search algorithm uses depth-first search, it will evaluate every time the leaf nodes from left to right. Therefore, if the most optimal nodes (positions) are located on the right side of the tree, the alpha-beta pruning will have a minor impact in optimizing the search. Thus, we need to determine which game states should be evaluated first and have a greater chance to determine the pruning in comparison with those less relevant, by satisfying the α > β criteria.

In practice, another heuristic is used to decide the move order, since there is not a single solution to determine that in the useful time in between the moves. In this case, chess good practice is used and as a consequence, checks, pawn promotion, castling and capturing a more valuable piece will be placed ahead of losing material, moving the king and a figure backwards or marginal. Just like the alpha-beta, move order will have no impact on the decision made by the computer and can only speed up the search time, making possible the search to a higher depth with the same processing power. In the engine, the moves are added starting with the Queen (best attacking piece) and ending with the King (least rewarding) before returning the valid moves. They are sorted according to their index set on the criteria above, trying to maximize the Most Valuable Victim – Least Valuable Agressor coeficient.

|  |  |  |
| --- | --- | --- |
| Type of move | Index score for sort | In the worst-case scenario, the complexity of the pruned tree remains the same and no optimization is made, with the value of *O(bd)[[4]](#footnote-4)*.If move ordering is perfectly applied before using the search algorithm, the complexity can theoretically be reduced to *O(bd/2)*. |
| Pawn promotion | 900 |
| Piece capture, from PxQ | 800 |
| Enpassant(PxP) | 0 |
| to QxP (minimal) | -800 |
| Castle | -900 |
| Standard | -1000 |
| Marginal move | -1100 |
| King | -1200 |

### Horizon effect and Quiescence search

As for now, the programme searches for the best branch of moves possible, until the maximum depth is reached, where it uses the heuristic to evaluate the position, without any kind of information about the future of the chess game, starting from that point. The problem with this approach is that the heuristic value can change significantly after the next move from the opponent. For example, within a last ply[[5]](#footnote-5), capturing a centrally placed knight with the queen will bring a powerful improvement of the heuristic and will likely be preferred during the backpropagation in the search tree, over other moves. As the maximum depth is reached, the AI does not “see” that the queen can be captured by another piece that defends the knight, right on the next move, making that branch delusory. This problem is named the “horizon effect”.

To diminish this issue, the Quiescence Search needs to be performed in order to further explore the unstable node. In practice, a branch that has such nodes on the last ply needs to be extended until the game state becomes “quiet”. This is be done by recalling the already implemented Negamax algorithm in such situations. To easily determine if a final node is quiet, chess intuition and another heuristic is needed, as there is not a specific method to determine this aspect. A significant improvement is easily done by further inspecting positions where a King is in check or capturing is still possible, as this kind of moves usually impacts the chess game heavily.

(in programme..)

## General structure of the AI

As it can be seen in Fig. 3 – Overview Diagram, the Engine calls the ai\_move function, which initially calls the selected ai method used. In the configuration file, the AI technique used for the current run is mention, alongside the depth of the search, when it is needed. Besides the case of a random move, the heuristic with its positional score tables is always used. The possible options are:

* random – selects a random move from the list received as a parameter from the current state in the engine
* greedy – it takes the best decision possible at each step looking only one move ahead. By all means, greedy is not a suitable strategy for chess.
* minimax – uses the method described in 3.2.1, without any pruning or optimization. Finally, it will take the same decision after a long thinking time and can be tested only with a low search depth (2 or 3). Nevertheless, it is useful for measuring the impact that optimization actually bring into practice, in comparison with the following method.
* nega\_max\_pruned – uses the method described in 3.2.2 and represents the most advanced version of the ai engine, which includes all the optimization features and can search to a higher depth than the standard minimax version. (Appenix 2) ?!

# Game behaviour examples & stats

## Positional development

screen shots

## Force mate

## Trapping a more valuable piece

## End game situations

# State of the art engines

In 1997, the chess engine developed by IBM named Deep Blue sets a new milestone for artificial intelligence and chess programming by defeating the world champion at that time, Gary Kasparov. It was the first time ever when a computer reached a superhuman level in such a complex game as chess. As a measure, its ability was ranked with 2853 ELO[[6]](#footnote-6) points, more than Kasparov’s 2795 at that time, and it only started the modern evolution of such engines, as more and more professionals in this sport acknowledged their potential for training and practice on all levels.

Therefore, the general trend in the field of chess engines is making the transition from classic to modern methods, which made the gap between humans and chess artificial intelligence even larger. A classic engine uses, at their core, use the same technologies presented above like the Minimax search with Alpha-Beta pruning, Move Ordering, Quiescence Search, Bitboards and Endgame tables. They all heavily rely on exhaustive search and human chess knowledge, that is passed to the computer through much more complex, tested and adapted heuristic functions than those used in the prototype implementation. Until late 2017, this kind of programmes dominated the competitions, the strongest being Stockfish, which was continuously upgraded over the years and reached ~3500 ELO.

In the last period of time, the approach on the AI chess changed, as modern Machine Learning techniques like Neural Networks and Deep Reinforcement Learning brought a significant performance advance raising from under 3500 to almost 3800 this year, with 1000 ELO points over the best human player. Deepmind[[7]](#footnote-7) published a research paper that formed the background for their experiment Alpha Zero, which won in practical games against Stockfish at that time (155wins, 839 draws, 6 losses), because the last one, a classical chess engine was time-limited and used much less computational power.

These days, Stockfish and Leela Zero lead the world rank and both of them use Deep Learning. Leela Zero is an open-source engine developed on the theoretical aspects shown in the AlphaZero article, that overcome the prototype performance, while the latest variant of Stockfish has received an update for the board evaluation function, as an Efficiently Updatable Neural Network (NNUE) is currently used.

|  |  |
| --- | --- |
|  | The AlphaZero algorithm learns only from self-play and Reinforcement Learning for developing a final model that was able to beat the strongest engine at that time, Stockfish. Even though the last one was developed with multiple version updates over multiple years, AlphaZero was able to overpower it within only 4 hours of training, without any human chess knowledge offered. It was given the rules of chess and started from random play. This perfectly underlines how powerful Deep Learning is, compared to classic search techniques.  In 2020, Deepmind reveals MuZero, an algorithm that reached the chess performance of AlphaZero and further adds the ability to learn environmental dynamics. Therefore, it was able to develop itself not only without chess knowledge but also without having the rules offered. This fact expands its usability to other games and fields like robotics, industrial control and intelligent assistants. |
|  |  |

## Techniques overview

|  |
| --- |
|  |

Both AlphaZero and MuZero (10) (11) use the Markov Decision Process model as their background for the general active Reinforcement Learning algorithm, as it follows:

* The deterministic environment
* Chess possible board states si ∈ S and available valid moves ai ∈ A, that can be proceeded accordingly
* The rewards given during the training process: R(s,a) → r. In the final, they should be large for smart moves that led to winning and small otherwise
* The policy function ( 𝛑 (s) → a ) receive a state of the board and outputs a probability distribution generated over the available actions (moves), that represents the chances of every move to be the most optimal
* The value function ( v(s) → vi ) Estimates the winning chances given the current board

Monte-Carlo Tree Search algorithm is performed for training the model. It adjusts the policy and the value function by simulating real game situations and comparing them with the data given by the deep learning model, computing a game adapted cost function and backpropagation. Compared to Minimax Tree Search, the search is made no longer in-depth but in breadth with a limited number of positions analyzed and the main advantage is that this method systematically faces end game situations. Therefore, the evaluation is precise and human-made heuristics are no longer required.

|  |
| --- |
| Fig. 17 - MCTS for Chess (12) |

The MCTS steps are adapted to the chess game as it follows:

* Selection – the tree levels respect the Minimax alternative selection strategy. Each player selects the node with the best wins/traverses ratio, maximizing the Upper Confidence Bound (UCB)
* Expansion – a new position is added to the search tree for further evaluation after an unexplored leaf node was reached
* Simulation – self-play uses the current policy until an endgame board is reached and the possible outcomes are -1, 0 or 1. (white losses, draw or white wins)
* Backpropagation – the nodes representing the traversed chess positions on the way to the most recent analyzed leaf node are updated

# Conclusions

To sum up, the thesis presents the algorithms and methods used for developing the chess game, for both single-player and multiplayer situations. It manages the moves validation by applying the described algorithms for chess rules on the current board representation. This first game component was fundamental for the application, as it further makes possible the observation and analysis of the AI logic.

In the second part, the logic behind the AI opponent module is described in detail and the trade-off between search time and performance was a constant concern. The optimizations made possible a deeper search in the same amount of time, through the efficient use of computational power and thinking time. The ability to look 3 moves ahead and have into consideration material advantage and strategical positioning clearly does not match a master level, but can cope with more than a beginner.

Finally, in the last part, the overview of the algorithms and techniques used by the most recent and intelligent engines that reasons the progress achieved by Deep Learning compared to classic methods. To put it simply the computer becomes more efficient when it learns by itself than applying human knowledge.

# Bibliography

1. *The Role of Chess in Artificial Intelligence Research.* **Robert Levinson, David E. Wilkins, T. Anthony Marsland, Jonathan Schaeffer, Feng-hsiung Hsu.** 1991. International joint conference on Artificial intelligence.

2. ***https://en.wikipedia.org/wiki/Rules\_of\_chess.* [Online]**

**3. *Bitboard methods for games.* Cameron Browne ( QUT, Brisbane, Australia ). s.l. : ICGA Journal, 2014.**

**4. *https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1.* [Online]**

**5. *https://www3.ntu.edu.sg/home/ehchua/programming/java/javagame\_tictactoe\_ai.html.* [Online]**

**6. *https://en.wikipedia.org/wiki/Negamax.* [Online]**

**7. https://www.chessprogramming.org/Simplified\_Evaluation\_Function. [Online]**

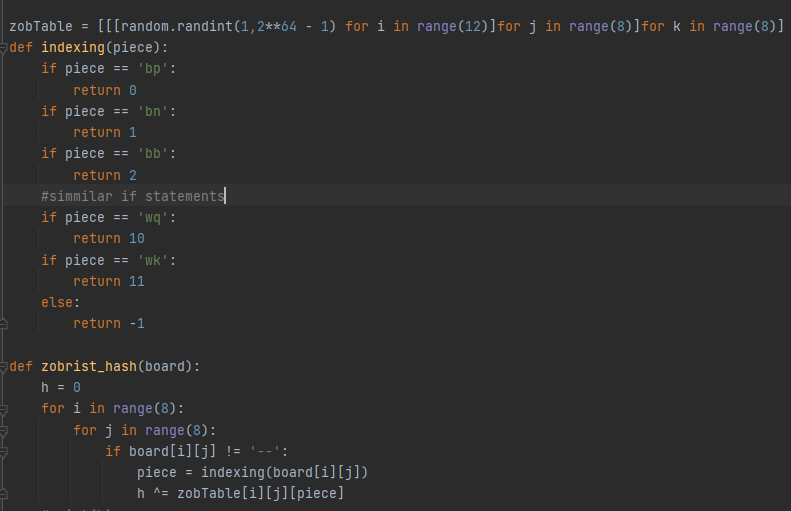
**8. Sockalingam, Kieran. *Alpha beta pruning.* s.l. : University of Oxford, 2015.**

**9. Zobriist, Albert. *A new hashing method with application for game playing (https://research.cs.wisc.edu/techreports/1970/TR88.pdf).***

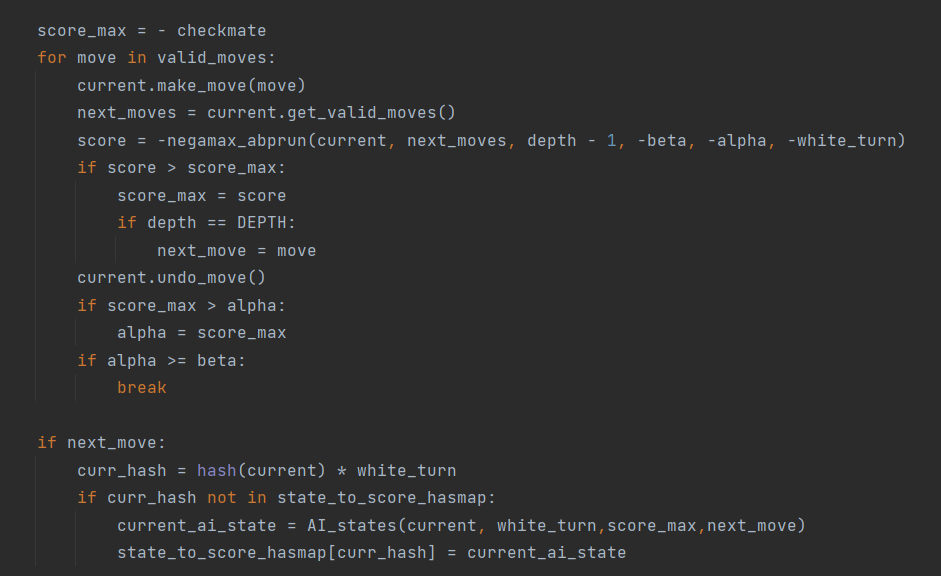
**10. *Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model (https://arxiv.org/pdf/1911.08265.pdf).* 2020.**

**11. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm ( https://arxiv.org/pdf/1911.08265.pdf ).* 2017.**

# Appendix with code



1



2

1. You can capture only enemy pieces except for the King [↑](#footnote-ref-1)
2. Shannon number (lower bound approximation) – 35 moves for each player on a move, resulting in 103 for a both sides, the factor being multplied by 40 – the average length of a game [↑](#footnote-ref-2)
3. Branching factor - the indicative average number of children for each node. In this case, it is a uniform average as it can vary from move to move [↑](#footnote-ref-3)
4. O() – defines the upper bound of a complexity function; b – branch factor; d – depth of the search [↑](#footnote-ref-4)
5. Ply – half from a complete turn, the move of one player alone. The layer of the search tree. [↑](#footnote-ref-5)
6. ELO – Official chess rating system that approximates the capabilities of a player based on previous results [↑](#footnote-ref-6)
7. Google artificial intelligence subsidiary [↑](#footnote-ref-7)