Development of an AI for the Game of Chess

Studienarbeit

for the

Bachelor of Science

from the Course of Studies Applied Computer Science

at the Cooperative State University Baden-Württemberg Mannheim

by

Yiwen Tu and Luca Sailer

April 2023

Time of Project	7 Month
Student ID, Course	5390097 and 7627674, TINF20AI2
Reviewer	Prof. Dr. Karl Stroetmann

Author's declaration

Hereby I solemnly declare:

1. that this Studienarbeit, titled *Development of an AI for the Game of Chess* is entirely the product of my own scholarly work, unless otherwise indicated in the text or references, or acknowledged below;
2. We have indicated the thoughts adopted directly or indirectly from other sources at the appropriate places within the document;
3. this Studienarbeit has not been submitted either in whole or part, for a degree at this or any other university or institution;
4. We have not published this Studienarbeit in the past;
5. the printed version is equivalent to the submitted electronic one.

We are aware that a dishonest declaration will entail legal consequences.

Mannheim, April 2023

Yiwen Tu and Luca Sailer

Abstract

The present paper shows an implementation of an AI written to play the game of chess. The AI is implemented using the programming language Julia. The most important functions of Julia will be explained in a way that a python programmer will feel very familiar with the language.

In addition we use the help of the library Chess.jl to implement the search algorithms the AI will use to determine the best move. Performance is one of the key factors of this project as better performance will result in shorter calculation times which allows the engine to search deeper and therefore make a better move.

In this project the search algorithm is improved step by step. Starting with the most simple way of searching in game theory the Minimax Algorithm is implemented. Afterwards, the Alpha-Beta-Pruning algorithm is implemented which prunes big parts of the game tree to decrease the amount of paths needed to be searched. Lastly Iterative Deepening has been implemented which will enable the algorithm to presort good moves and consider these first. This will increase the number of paths pruned resulting in an even faster calculation. All algorithms will stop at a certain depth and return the best move until then.

To prevent the horizon effect a simple version of a Quiescence Search is implemented. It will prevent the engine from stopping at the latest depth if the position might have some obvious dangers which will lead to a misleading result.

Contents

Acronyms V

List of Figures VI

List of Tables XI

1. Introduction 1

2. Theory 3
- 2.1. General conditions . . . 3
- 2.2. Process Chess Engine . . . 4
- 2.3. Introducing in Julia . . . 4
- 2.4. Phases of a game of chess . . . 9
- 2.5. Chess ELO System . . . 10

3. Methods 12
- 3.1. Hypothesis . . . 12
- 3.2. Implementation . . . 12
- 3.3. Important Functions of Chess.jl . . . 12
- 3.4. Advanced Board Struct . . . 21
- 3.5. Detection of a Draw by Repetition . . . 26
- 3.6. Simple Evaluation Function . . . 27
- 3.7. Random Chess . . . 42
- 3.8. Minimax . . . 43
- 3.9. Zobrist hashing . . . 51
- 3.10. Memoization . . . 59
- 3.11. Alpha-Beta Pruning . . . 63
- 3.12. Iterative Deepening . . . 73
- 3.13. Complexity . . . 81
- 3.14. Quiescence Search . . . 84

4. Results 90

5. Discussion 92
- 5.1. Conclusion . . . 92
- 5.2. Perspective and Alternatives . . . 92

Bibliography 93

A. Appendix **95**
A.1. Weekly Chess Rating distribution 95
A.2. Play as White against ELO 1500 Bot 96
A.3. Play as Black against ELO 1500 Bot 97
A.4. Suggestion Rating Chess.com about Game with White 98
A.5. Suggestion Rating Chess.com about Game with Black 98
A.6. Engine Comparing 99
A.7. Strength comparison 114

Acronyms

PGN Portable Game Notation

List of Figures

2.1. Python and Julia: Arithmetic operators 5
2.2. Initialization Dictionary in Julia 6
2.3. Specifying Variables with explicit Types 6
2.4. Conditional Mastery: Making informed decisions with if-elseif-else Statements 7
2.5. Exploring Loops in Julia 8
2.6. Declaring functions in Julia 8
2.7. Declaring inline-functions in Julia 8
2.8. Handling with libraries in Julia 9
2.9. Initialization Lists in Julia 9
2.10. Using sort function without exclamation mark 9
2.11. Using sort function with exclamation mark 9

3.1. Import library Chess 12
3.2. Initializing the Game with the Starting Position 13
3.3. Converting Forsyth-Edwards Notation to a Board 14
3.4. Using sort function with exclamation mark 14
3.5. Using sort function with exclamation mark 15
3.6. Chess Move interpretation without exclamation mark 15
3.7. Chess Move interpretation without exclamation mark 15
3.8. Undoing Chess Moves 16
3.9. Generating a List of legal Moves for a given Chessboard 16
3.10. Checking if the Board is in a terminal state 16
3.11. Using sort function with exclamation mark 17
3.12. Determining if the Chessboard is in a Checkmate State 17
3.13. Checking for sufficient Material on the Chessboard 17
3.14. Checking for 50 move rule on the Chessboard 17
3.15. Note: Issue with the fromfen function 17
3.16. Initializing a chess game with the Game() function 18
3.17. Initializing a chess game with the SimpleGame() function 18
3.18. Type of board object returned by the board(game) function 18
3.19. Adding a move to the game using the addmove!() function 18
3.20. Moving the current position of the game to the beginning 18
3.21. Moving the current position of the game forward 18

3.22. Creating a Piece object with color "BLACK" and type "KING" 19
3.23. Retrieving the piece on the start board at position file A and rank 1 . . . 19
3.24. Retrieving the PieceType of the piece on the start board at position file A and rank 1 19
3.25. Retrieving the PieceColor of the side that moves next on the start board 19
3.26. Evaluating the material gain or loss after applying the move "c3a1" . . . 20
3.27. Setting the value of the "Date" header to "2023.01.01" for the given game 20
3.28. Creating a PGNReader object 21
3.29. Parsing the PGN file from the io-Stream 21
3.30. Parsing a Game or SimpleGame object into a PGN file using the game-topgn(g) function 21
3.31. Importing the Chess package and including the Repetition and ZobristHashing notebooks 22
3.32. Definition of the 'AdvBoard' mutable struct 22
3.33. The AdvBoard constructor function, which takes a Board as input and returns an AdvBoard object with calculated attributes 23
3.34. Function domoveAdv! implementation 24
3.35. Function undomoveAdv! implementation 24
3.36. Function clearRepCounter! implementation 24
3.37. Function evaluate_move implementation 25
3.38. Function evaluation implementation 26
3.39. Assignment of values to each chess piece 28
3.40. Julia package and module imports 28
3.41. Including notebook files 28
3.42. Implementation is_endgame function 29
3.43. Defining the global variable PIECE_VAL 30
3.44. Implementation get values function 30
3.45. Defining the global variable WHITE_PIECE_SQUARE_TABLES 32
3.46. Defining the global variable "BLACK_PIECE_SQUARE_TABLES" 34
3.47. Implementation get_square value function 35
3.48. Implementation "calc_score" function 35
3.49. Implementation "evaluate_position" function 36
3.50. Implementation "valueCapturePiece" function 37
3.51. Implementation "calcEnPassant" function 38
3.52. Implementation "calc_castle" function 38
3.53. Implementation "evaluate_move" function 40
3.54. Implementation overload "evaluate_move" function 41
3.55. Implementation "terminal_evaluation" function 42
3.56. Julia package and module imports 42

3.57. Implementation of the 'generateRandomMove' function 42
3.58. Implementation of the 'generateRandomMove' function 42
3.59. Implementation of the 'playRandomMove' function 43
3.60. Julia package and module imports 43
3.61. Including notebook 43
3.62. Defining global Cache for Minimax 43
3.63. Search Tree Minimax 44
3.64. Implementation of maxValue no incremental no Memoization 45
3.65. Implementation of minValue non incremental no Memoization 46
3.66. Implementation of maxValue no Memoization 46
3.67. Implementation of minValue no Memoization 47
3.68. Implementation of maxValue 48
3.69. Implementation of minValue 49
3.70. Implementation of MiniMax no Memoization 50
3.71. Implementation of MiniMax 51
3.72. Julia package and module imports 51
3.73. Definition of global indices for chess pieces 53
3.74. Definition of a struct for Zobrist hashing 53
3.75. Implementation of the constructor for Zobrist hashing 54
3.76. Definition of a global variable 'zobrist' using the ZobristHashing constructor 54
3.77. Implementation of the 'zobrist_hash' function for calculating the Zobrist hash of a chess board 55
3.78. Implementation of the updateCastleRightsHash function for updating the Zobrist hash after a move that affects the castling rights 56
3.79. Implementation of the updateEnPassantHash function for updating the Zobrist hash after a move that affects the en passant square 57
3.80. incrementally implementation of the 'zobrist_hash' function for calculating the Zobrist hash of a chess move 59
3.81. Julia package and module imports 60
3.82. Including notebook files 60
3.83. Implementation function initCache() 60
3.84. Implementation function evaluate() 62
3.85. Implementation function store_cache() 63
3.86. alpha_beta.png 64
3.87. Julia package and module imports2 65
3.88. Including notebook files needed for Alpha-Beta Pruning 65
3.89. Implementation of the Alpha-Beta Pruning algorithm (maximizing player) 66
3.90. Implementation of the Alpha-Beta Pruning algorithm (minimizing player) 68

3.91. Implementation of the Alpha-Beta Pruning algorithm (complete) 69
3.92. Implementation of Alpha-Beta Pruning with Memoization (maximizing player) 70
3.93. Implementation of Alpha-Beta Pruning with Memoization (minimizing player) 71
3.94. Implementation of Alpha-Beta Pruning with Memoization (complete) . 72
3.95. Julia package and module imports 73
3.96. Including notebook files 73
3.97. Implementation of the 'maxValue' function 75
3.98. Implementation of the 'minValue' function 77
3.99. Implementation of the 'value_cache' function 77
3.100Implementation of the 'pd_evaluate' function 79
3.101Implementation of the 'iterativeDeepening' function 81
3.102Julia package and module imports for the Calculation of Complexity . . 81
3.103FEN representation of the chess position between Ding Liren and Ian Nepomniachtchi in 2023 82
3.104Code snippet calculating of the number of moves in the chess position. . 82
3.105Code snippet defining the function "count_moves" to recursively calculate the number of moves in a chess position 82
3.106Result of the "count_moves" function using the minimax algorithm on the chess position between Ding Liren and Ian Nepomniachtchi in 2023 82
3.107Result of the "count_moves" function using the Alpha-Beta-Pruning algorithm on the chess position between Ding Liren and Ian Nepomniachtchi in 2023 83
3.108Result of the "count_moves" function using the Iterative Deepening algorithm on the chess position between Ding Liren and Ian Nepomniachtchi in 2023 83
3.109Julia package and module imports 84
3.110Including notebook files 84
3.111Implementation of the hasCaptureMoves function 85
3.112Implementation of the isTacticalMove function 86
3.113Implementation of the quiesceMax function 87
3.114Implementation of the quiesceMin function 88
3.115Implementation of the quiesceSee function 89

A.1. Weekly Classical Chess Rating distribution 95
A.2. Weekly Rapid Chess Rating distribution 95
A.3. Play as White against ELO 1500 Bot 96
A.4. Play as Black against ELO 1500 Bot 97

A.5. Suggestion Rating Chess.com about Game with White 98
A.6. Suggestion Rating Chess.com about Game with Black 98
A.7. Julia package and module imports1 99
A.8. Including notebook1 files to compare the engine 99
A.9. Implement a function textTime 100
A.10.Result of comparing 101
A.11.Results of Black King's Indian opening 103
A.12.Results of Middlegame Ding Liren vs. Ian Nepommiachtchi, 2023 105
A.13.Results of Middlegame Black: Fabiano Caruana - Magnus Carlsen, 2018 107
A.14.Results of Endgame White: Magnus Carlsen - Ian Nepomniachtchi, 2021 109
A.15.Results of Endgame black: Mikhail Botvinnik - Robert James Fischer, 1962 111
A.16.Julia package and module imports 114
A.17. 115
A.18.Result Comparing Strength of the Opening White: Berlin Opening . . . 115
A.19.Result Comparing Strength of the Opening Black: King's Indian Defense 116
A.20.Result Comparing Strength of the Middlegame White: Ding Liren - Ian Nepomniachtchi, 2023 117
A.21.Result Comparing Strength of the Middlegame Black: Fabiano Caruana - Magnus Carlsen, 2018 118
A.22.Result Comparing Strength of the Endgame White: Magnus Carlsen - Ian Nepomniachtchi, 2021 119
A.23.Result Comparing Strength of the Endgame Black: Mikhail Botvinnik - Robert James Fischer, 1962 120

List of Tables

4.1. Search time of all algorithms in seconds 90
4.2. Stockfish Evaluation Compared to Calculation Depth 91

1. Introduction

"Everybody uses computers to train so much now that the first nine or 10 moves of a match are made without thinking." (Hikaru Nakamura) Professional chess players rely on chess engines to analyze and learn from them to ultimately gain in strength. Chess is one of the oldest and at the same time most played board games all around the world. Many AIs have tried cracking the game of chess. Due to the vast amount of positions 10^{120} [1], also called Shannon number, the game is still very complex and far from solved. However, many artificial intelligences reach very high strengths. In 1996 the chess engine Deep Blue defeats the world champion from that time Garry Kasparov. This was the first time a chess computer has beaten a world chess champion. Now chess engines are even stronger than humans and can reliably defeat any human.

In this paper, the development of a chess AI is explained in its detail. For the implementation we use the programming language Julia and we use Jupyter notebooks to document the implementation. The goal of the chess AI is to find the best move. To achieve this, the engine will use a classical approach used in game theory. The engine will calculate future moves and evaluate the positions using search algorithms. After that, it will trace back and find the best move if both players play optimally. In this project different algorithms are implemented slowly improving the performance of the engine. Improving the performance allows the engine to go calculate more paths in the same amount of time, resulting in a higher strength. Therefore, it is crucial to optimize the performance. The simplest search algorithm is the minimax algorithm, which recursively searches the whole game tree until a certain depth and choosing the best evaluation of a position. To evaluate a position, we use a simplified evaluation heuristic, which considers the material and positional advantage of all pieces. The heuristic function estimates the strength of a position without looking into future moves. Additionally, the minimax algorithm will be expanded using a memoization function. Using this calculated values, they are stored in a cache. If a position is reached multiple times in a calculation, these positions do not have to be reevaluated multiple times. Their values can be directly read out of the cache. The performance of the cache is improved by the introduction of Zobrist hashing. After making all improvements around minimax, a first improvement to the search algorithm itself is introduced by the alpha-beta-pruning algorithm. Using this algorithm many paths can be pruned, therefore it is not necessary to compute the whole game tree. When using alpha-beta-search, a new memoization function is needed. The next improvement of the search algorithm

is the iterative deepening algorithm. Using this algorithm, the depth is slowly increased. At each depth, the moves are evaluated and sorted by their evaluation. Ordering the moves can be a big advantage. Calculating the best moves will result in earlier pruning by the alpha-beta search, drastically improving the calculation time. At last, the quality of the search algorithm is improved using quiescence search. This search is specific to certain games, especially important in the game of chess. It will stop the horizon effect and makes sure that a position is quiet before calculating its actual score.

2. Theory

In this chapter the general idea of how the chess engine is programmed is described. Additionally an explanation is given to why we chose the different approaches which we will implement in the upcoming chapters.

2.1. General conditions

This project is a university project and should deepen the understanding of the basics of artificial intelligence. On the premise of this, we are given to use the search algorithms used in classical game theory. Beginning with the simplest algorithm Minimax, then Alpha-Beta-Pruning, then Iterative Deepening. Each time improving the performance of the engine. Memoization is used to save the values of the positions. This will improve the performance because the engine calculates many transpositions of a chess game. A chess position can be reached using different move orders, which is called a transposition. Additionally, we expand the search using Quiescence Search, which is specifically used for the game of chess.

For the heuristic function, we chose to use the simplified evaluation function by Tomasz Michniewski [2]. This function returns a centipawn value using the most important factors for calculating a position without looking deeper into the future, such as material advantage and the relative positions of different pieces. Other, more complicated functions additionally consider double or isolated pawns and other factors. For simplicity and performance reasons, these will not be considered in this project, as other factors are not as important and can also be misleading.

For the development of the chess engine the programming language Julia was chosen. Many reasons lead to choosing Julia as the language for this project. The first is the similarity to the programming language Python. This allows us to get a good overview of the language without a lot of trouble. Also, the way of programming is very similar, which eases the way of thinking about how to implement the correct algorithms. The second reason for choosing Julia over Python is performance. Python is known to have a lot less performance than many other programming languages such as Java, C, C++[3]. Julia provides an additional performance boost over Python, but it will be worse than using C directly. Julia also allows us to use Jupyter Notebooks for the development of this project. This allows us to easily document and develop at the same

time. Additionally, all notebooks can be neatly formatted into this study, as seen in chapter 3.

2.2. Process Chess Engine

In this chapter, we will discuss the three key components [4] of a chess AI :

1. Move Generator: The move generator is a module that is responsible for generating the next legal move based on the current board state. In this paper, the move generator is the "nextMoves" function.

2. Evaluation Function: The evaluation function is a crucial part of any heuristic algorithm, including chess AI. It is responsible for evaluating the current state of the game. In chess, we can evaluate the position based on various factors, such as position and material. The evaluation function is used to determine the score of each position in the game tree.

3. Search Algorithms: Search algorithms are used to find the best move based on nextMoves and the evaluation function. Chess is a zero-sum game, which means that the sum of the evaluation at any time during the game will be equal. If one player gains an advantage, the opposing player will have the same amount of disadvantage. The most commonly used search algorithm is the Minimax algorithm, which searches the game tree recursively to find the best move. The Alpha-Beta Pruning Algorithm is a variation of the Minimax algorithm that improves its efficiency by eliminating unnecessary branches in the game tree.

2.3. Introducing in Julia

In the following section, we explain the difference between Python and Julia. The aim is to enable a Python programmer to get a quick start into Julia. In particular, syntax and important concepts that differ in Julia from Python are discussed.

2.3.1. General

Julia is a programming language primarily used for mathematical and scientific computing. It is often compared to Matlab and R due to its focus on mathematical computations. Julia has many similarities with Python in terms of syntax, as explained in the following subsections.

However, the most notable feature of Julia is its performance and efficiency in computations. A benchmark analysis conducted by Julia developers revealed that Julia's runtime is only slightly slower than that of the C programming language. C is widely regarded as one of the most performant programming languages, as it is very system-oriented and low-level. The significant difference in runtime between Julia and Python highlights Julia's capability to perform fast computations.[5]

Another notable feature of Julia is its ability to perform just-in-time (JIT) compilation, allowing it to optimize code at runtime. This feature ensures that Julia can generate highly optimized machine code that can run much faster than traditional interpreted code. Furthermore, Julia's JIT compilation makes it possible to write code that is both efficient and easy to read, making it an excellent choice for researchers and scientists who want to focus on solving problems rather than writing complex code.

2.3.2. Computing operators

Arithmetic operators

The syntax for arithmetic operations in Julia is almost identical to the syntax of Python except for the power. The power is expressed in Python with a double **, whereas in Julia the power is calculated with the caret symbol ∧ (or roof).

```
# Python
> 2 ** 3
8

# Julia
> 2 ^ 3
8
```

Figure 2.1.: Python and Julia: Arithmetic operators

The arithmetic operations of addition, subtraction, multiplication, and division are written as in Python.

Bitwise operators

Logical operators like the bitwise AND '&', bitwise OR '|' etc. can also be used.

2.3.3. Variables

When naming variables, Julia allows the use of a subset of Unicode characters. [6] This means that international alphabets and characters, including the Greek alphabet, can be used.

The data type is automatically detected during runtime, which means that Julia is also dynamically typed.

In Julia, data structures such as arrays, lists, dictionaries, tuples and sets can also be used.

Arrays are initialized with square brackets like in Python. During runtime, a data type is assigned to the array. As long as all elements of the array belong to the same data type, the array is assigned this data type. There can also be different data types in an array. Then the array receives the general data type Any.

Other data types are initialized with the corresponding keyword. The properties of these data structures are the same as in Python. Dictionaries are initialized with the keyword 'Dict', sets with 'set', tuples with 'tuple', lists with 'list'.

```
> A = Dict("a" => 1, "b" => 2)
DictString, Int64 with 2 entries:
  "b" => 2
  "a" => 1
```

Figure 2.2.: Initialization Dictionary in Julia

2.3.4. Data types

Julia is a dynamically typed programming language. Data types are determined during runtime and do not have to be specified as in other programming languages (e.g. Java). The known data types (Int, String, Float) are represented in Julia.

A data type can be prescribed to a variable. This can be declared with '::(type)'.

```
# Julia
> x::Int = 9
9
> typeof(x)
Int
```

Figure 2.3.: Specifying Variables with explicit Types

Basically, Julia is based on primitive data types. These include Boolean, Char, Int8, Int16, . . . , UInt8, UInt16, . . . , Float16, . . . [7] These can be used directly or to create composite data types like structs.

Data types can belong to a subset of another data type. This is needed for the conversion of two data types. For example, $Number \subseteq Real \subseteq Any$. *Any* is the most general data type. This dependency can be queried with the is':>' operator.

2.3.5. Control Flow

Conditional query

In Julia, as in many programming languages, a case distinction is possible by an if-elseif-else statement. The following syntax is used for this purpose.

```
if x < 0
    println("x is negetive")
elseif x > 0
    println("x is positive")
else
    println("x is zero")
end
```

Figure 2.4.: Conditional Mastery: Making informed decisions with if-elseif-else Statements

The important differences to Python are on the one hand the missing colons (:) after the queries and on the other hand the keyword 'end', which concludes the if-query. Python uses white spacing (the indentation) to identify the end of the execution block.

Loops

The loop constructs 'for' and 'while' are used in Julia in the same way as in Python. Syntactically there are small differences.

```
> i = 1;
> while i <= 3
      println(i)
      global i += 1
  end
1
2
3
```

```
> for i = 1:3
      println(i)
  end
1
2
3
```

Figure 2.5.: Exploring Loops in Julia

After the condition or iterator, a colon is not set in Julia. Additionally, the end of the loops must be terminated with the keyword 'end'.

2.3.6. Function

Functions can be declared in Julia with the keyword 'function'. To end a function, the keyword 'end' is required. Python's indentation of functions, is not necessary in Julia, but recommended to increase readability. An example of a function in Julia is the following function 'sayHello(name)'.

```
function sayHello(name)
    println("Hi $name, it's great to see you!")
end
```

Figure 2.6.: Declaring functions in Julia

Anonymous functions, called lambda functions, can also be created in Julia. For this a minus and greater sign is needed. In combination, the two characters are represented as an arrow ('->'). Before the arrow the parameters are declared. In the following example the function 'sayHello' is stored in the varibale 'sayHello'.

```
sayHello = name -> println("Hi $name, it's great to see you!")
```

Figure 2.7.: Declaring inline-functions in Julia

2.3.7. Libraries

Julia has more than 7400 libraries officially registered[8]. In contrast, Python has more than 3.9 million officially available. [9] To use libraries in Julia, the package manager should be added once in the first step. The command 'using pkg' can be used for this. In general, the 'using' command is the equivalent of the import command to Python. If the library has not been downloaded, this can be done with the command 'Pkg.add('[library name]'). The following example shows a color palette using the library 'color'.

```
using Pkg # Importieren Paket-Manager
Pkg.add("Colors") # Download Bibliothek
using Colors # Import-Befehl
palette = distinguishable_colors(10) # Aufrufen Methode von Colors
println(palette) # Darstellen Farbpalette
```

Figure 2.8.: Handling with libraries in Julia

2.3.8. Julia specifics

The programming language Julia distinguishes between modifying and non-modifying of arguments. The exclamation mark (!) after the name of a function show that it is a modifying function. Thus a function sort! can modify the arguments and sort cannot. [10]

For clarity, a list with random numbers from 1 to 10 is created.

```
randList = [3, 5, 2, 4,  8, 6, 7,9, 1, 10]
```

Figure 2.9.: Initialization Lists in Julia

Now the sort function is called without exclamation mark. This way the original list does not change.

```
sortList = sort(randList)
println(randList) #Ausgabe 3, 5, 2, 4, 8, 6, 7, 9, 1, 10
```

Figure 2.10.: Using sort function without exclamation mark

In contrast to the previous function, if you call the sort function with an exclamation mark, the original list gets sorted.

```
sortList = sort!(randList)
println(randList) #Ausgabe 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Figure 2.11.: Using sort function with exclamation mark

2.4. Phases of a game of chess

The game of chess is generally divided into three phases: The opening, middlegame and endgame. A new game of chess begins with the opening. In this phase many moves have been calculated and documented by an opening book. Many chess engines use an opening book/database to choose the best moves in this phase. After reaching the

end of the database the game transitions into the middle game. The middle game is the most complex stage of the game as the number of possible moves are very high. Not many pieces have been traded out but in comparison to the opening the pieces are far more active and open leading to many possible moves. After many pieces have been traded out the game will enter the endgame. In this phase there are only a few pieces left on the board. Therefore the amount of possible moves has drastically decreased. The endgame is still very complex as low possible moves results to the possibility of calculating more than double the amount of moves into the future compared to the middlegame.

For the chess engine it is important to determine when the game has entered which stage. This information is used to weigh the position of the King. Optimally the king should stay as safe as possible during the opening and middlegame phase of a chess game. Having an unsafe king will result in many tactical opportunities for the enemy. During the endgame the king is one of the strongest pieces left as often in the endgame there are only pawns and kings on the board. Therefore an active, centralized king is very important. The distinction of middle and endgame are very important. The opening can be treated as the middlegame. A board is considered in the endgame phase when either "both sides have no queens or every side which has a queen has additionally no other pieces or one minor piece maximum." [2] It is important to note that for this project we further specify that if a pawn gets promoted, the position can transition from an endgame into a middlegame. And later onward back to an endgame.

2.5. Chess ELO System

The ELO system is a rating system used to measure the relative strength of players or teams in competitions such as chess. The system was developed by Arpad ELO in the 1950s while working for the US Chess Federation [11]. It was first used in chess tournaments in 1960 and has since become one of the most well-known and widely used rating systems for competitions.

The ELO system is based on the idea that a player's performance can be measured by the results of their games against other players. Each player has an ELO rating, which reflects their relative strength compared to other players. When a player wins against another player, the losing player loses ELO points, while the winner gains points.

Contrary to popular belief, the ELO rating does not represent a player's true strength. The system only calculates the probable outcome of a person's game against another player. As a rule of thumb, a player whose ELO is 100 points higher than their opponent's will win about 64% of games, which is about five out of eight games. A player

with an ELO of 200 points higher than their opponent's will probably win round about 75% or three out of four games. [12]

World champion Magnus Carlsen holds the record for the highest ELO ever achieved by a human player. In 2014, the Norwegian achieved an impressive ELO of 2882.[13]

In the weekly chess rating distribution of Lichess (A.1), a well-known platform for playing chess online, both in Blitz and Classical, it can be observed that players mostly have an ELO of 1500. Hence, we assume that the ELO of 1500 reflects the average playing level.

3. Methods

3.1. Hypothesis

The Hypothesis analyzed in this paper is the following: Will a chess computer developed with Julia using the techniques of a simplified rating function, alpha-beta search, memoization, iterative deepening and quiescence search be able to play chess games against human opponents and achieve intermediate level performance of above 1500 ELO-Rating.

3.2. Implementation

In this chapter, all notebooks that were implemented to prove the hypothesis are included. All functions and algorithms are explained and documented. The theory of the algorithms is also included in the notebooks. The source code of this project is available at GitHub: https://github.com/GeekYiwen/ChessAIJulia.git

3.3. Important Functions of Chess.jl

This notebook explains all functions we will need from the Chess.jl library to achieve our goal of implementing a Chess AI.

The library `chess.jl` can be installed and imported into the notebook by the standard package manager.

[ ]:
```
using Pkg
```

[ ]:
```
# Pkg.add("Chess")
using Chess, Chess.PGN
```

Figure 3.1.: Import library Chess

3.3.1. Documentation

The full documentation and API Reference can be found under: https://romstad.github.io/Chess.jl/dev/

3.3.2. Boards

The function `startboard()` returns a chess board in the starting position.

```
[ ]: startboard()
```

Figure 3.2.: Initializing the Game with the Starting Position

Forsyth-Edwards Notation

Forsyth-Edwards Notation or short FEN is used to represent any chess position as a single String. The String is separated into # parts separated by spaces.

The first part represents the current board position. The board is viewed from white's position, meaning that the top row will be the eighth row and the bottom row being the first row. Starting from the top row all pieces on this row from left to right will be notated followed by a slash / ending the current row and starting the new row. This is done for each of the eight rows. Pieces are notated using their initial letter of their piece name (except for the Knight using the letter `N`).

Pawn => p Knight => n Bishop => b Rook => r Queen => q King => k

White pieces are represented using a capital letter and black with lower-case letters. Blank spaces in a row will be represented by the number of consecutive empty spaces.

The second part is either the letter `w` for white or `b` for black, depending on which sides moves next.

The third part represents the castling rights. A capital `K` means that white has the option of castling King/short side. The initial position has all castling rights and therefore has `KQkq` as the third part of the FEN String. If no castling rights are available, the third part of the FEN String is represented by a `-`.

The fourth part of the string is the square, the moving side can en passant on. Whenever a pawn moves two squares forward, the square one behind that pawn will be listed in this part of the FEN string. If there is no en passant available this part contains a `-` character.

The fifth part is the halfmove clock. It represents the number of halfmoves done without capturing a piece or advancing a pawn. It is incremented after every half move and set to 0 if one of the two events occur.

The sixth part is the fullmove counter. It shows the number of fullmoves done in the whole game. Initially it is set to 1 and is incremented after black has moved.

`fromfen(fenstring)` returns a board object given a Forsyth–Edwards Notation String inputted in the argument `fenstring`.

```
[ ]: spanishOpeningBoard = fromfen("r1bqkbnr/1ppp1ppp/p1n5/1B2p3/4P3/5N2/
      ↪PPPP1PPP/RNBQK2R w KQkq - 0 4")
```

```
[ ]: four_knights = fromfen("r1bqkb1r/pppp1ppp/2n2n2/4p3/4P3/2N2N2/PPPP1PPP/
      ↪R1BQKB1R w KQkq - 4 4")
```

Figure 3.3.: Converting Forsyth-Edwards Notation to a Board

3.3.3. Move

The class `Move` represents a chess move a player can make when it is their turn. A `Move` object contains a `from` square and a `to` square. The from-square determines from where a piece is moved, likewise for the to-square. For this project we will create `Moves` by using the `movefromstring(string)` function.

movefromstring function

The `movefromstring(string)` function takes in a string and returns a `Move` object if it can parse the string into a chess move.

Possible strings are UCI (universal chess interface) move strings. (https://www.chessprogramming.org/UCI)

Regular Move: - Strings specifying the from-square and to-square coordinate (e.g. “e2e4” => From e2 to e4)

Promoting: - Strings specifying the from-square and to-square coordinate and a piece into which the pawn should promote (e.g. “a7a8q” => From a7 to a8 promoting to a queen)

If it cannot parse the string it will return `nothing`.

```
[ ]: movefromstring("e2e4")
```

Figure 3.4.: Using sort function with exclamation mark

```
[ ]: movefromstring("a7a8q")
```

Figure 3.5.: Using sort function with exclamation mark

domove function

The `domove(board, move)` function takes in a `board` object and a `move` and returns a new board where the given `move` has been made on the given `board`. Moves a specified using strings or move objects. Chess.jl will try to parse the String into a move object.

```
[ ]: b = startboard()
     move = movefromstring("e2e4")
     domove(b, move)
```

```
[ ]: b = startboard()
     domove(b, "e4")
```

Figure 3.6.: Chess Move interpretation without exclamation mark

domove! function

The `domove!(board, move)` function takes in a `board` object and a `move` and applies the given `move` on the given `board` and returns an UndoInfo object. Moves can be specified using strings or move objects. Chess.jl will try to parse the String into a move object.

```
[ ]: b = startboard()
     domove!(b, "e4")
     b
```

```
[ ]: b = startboard()
     domove!(b, "e4")
```

Figure 3.7.: Chess Move interpretation without exclamation mark

undomove! function

The `undomove!(board, UndoInfo)` takes in a board and an `UndoInfo` received from the `domove!(board, move)` function and undoes the move done in the `domove!` function and returns the previous board.

```
[ ]: b = startboard()
     u = domove!(b, "e4")
     undomove!(b, u)
     b
```

Figure 3.8.: Undoing Chess Moves

moves function

The function `moves(board)` takes in a `board` and returns a list containing all legal moves for this board.

```
[ ]: b = startboard()
     moves(b)
```

Figure 3.9.: Generating a List of legal Moves for a given Chessboard

3.3.4. Game state

The result of a game can be determined using the following methods: - `isterminal(board)` - `ischeckmate(board)` - `isstalemate(board)` - `ismaterialdraw(board)` - `isrule50draw(board)`

isterminal

The `isterminal(board)` function takes in a `board` and returns whether the board is in a terminal state or not. A terminal state is a position where the side to move cannot make another legal move. This is reached when the position is in a checkmate, stalemate or when there is insufficient material or when the 50 move rule has been reached.

```
[ ]: stafford_gambit_trap = fromfen("r2Bk2r/ppp2ppp/2p5/8/4n1b1/3P4/PPP1KbPP/
       ↪RN1Q1B1R w kq - 2 9") # position where white is checkmated
```

```
[ ]: isterminal(stafford_gambit_trap)
```

Figure 3.10.: Checking if the Board is in a terminal state

ischeckmate

The `ischeckmate(board)` function takes in a `board` and returns whether this board has ended in a checkmate or not.

```
[ ]: ischeckmate(stafford_gambit_trap)
```

Figure 3.11.: Using sort function with exclamation mark

isstalemate

The `isstalemate(board)` function takes in a `board` and returns whether this board has ended in a stalemate or not.

```
[ ]: stalemate_board = fromfen("5k2/5P2/5K2/8/8/8/8/8 b - - 0 50") #␣
     ↪position where white stalemated black
```

```
[ ]: isstalemate(stalemate_board)
```

Figure 3.12.: Determining if the Chessboard is in a Checkmate State

ismaterialdraw

The `ismaterialdraw(board)` function takes in a `board` and returns whether the board still has sufficient material to have a decisive result.

```
[ ]: insufficient_material_board = fromfen("8/8/4kb2/8/8/3K4/8/8 w - - 0 4")␣
     ↪# position where both sides have insufficient material
```

```
[ ]: ismaterialdraw(insufficient_material_board)
```

Figure 3.13.: Checking for sufficient Material on the Chessboard

The `isrule50draw(board)` function takes in a `board` and returns whether the 50 move rule has been reached.

```
[ ]: fifty_move_board = fromfen("8/6k1/8/8/4R3/5r2/1K6/8 b - - 100 108") #␣
     ↪position where 50 move rule has reached
```

Figure 3.14.: Checking for 50 move rule on the Chessboard

Note: fromfen is currently bugged as it does not read the 5th part of a Fen string and therefore does not read the number of half moves where no capture or pawn move has been done. Therefore it is initializing a new `Board` with 0 half moves.

```
[ ]: isrule50draw(fifty_move_board) # should be true
```

Figure 3.15.: Note: Issue with the fromfen function

3.3.5. Games

The `Game` and the `SimpleGame` class both store information about a whole chess game, such as the move sequence or any information about the game. The `Game` has more options than the `SimpleGame` such as branching of during the game. Therefore, the `SimpleGame` performs better compared to the `Game` class.

The `Game()` function returns a `Game` object in the starting position.

```
[ ]: g = Game()
```

Figure 3.16.: Initializing a chess game with the Game() function

The `SimpleGame()` function returns a `Game` object in the starting position.

```
[ ]: sg = SimpleGame()
```

Figure 3.17.: Initializing a chess game with the SimpleGame() function

The `board(game)` takes in a `game` and returns the current position of the game as a board object.

```
[ ]: typeof(board(g))
```

Figure 3.18.: Type of board object returned by the board(game) function

The function `addmove` takes in a `game` and a `move` and adds the move to the given game.

```
[ ]: addmove!(g, "e4")
     g
```

Figure 3.19.: Adding a move to the game using the addmove!() function

The functions `forward!(game)`, `back!(game)`, `tobeginning!(game)`, `toend!(game)` take in a `game` and move the currrent position of the game.

```
[ ]: tobeginning!(g)
```

Figure 3.20.: Moving the current position of the game to the beginning

```
[ ]: forward!(g)
```

Figure 3.21.: Moving the current position of the game forward

3.3.6. Pieces

The `piece` class represents an instance of any chess piece. It holds information of the type and color of the piece. For example `PIECE_WP` for white pawn, `PIECE_BN` for black knight or `EMPTY` for an empty square.

The class `PieceType` represents the type of the piece and therefore has the values `PAWN`, `KNIGHT`, `BISHOP`, `ROOK`, `QUEEN` and `KING`.

The class `PieceColor` represents the color of a piece and is either `WHITE`, `BLACK` or `COLOR_NONE`. The `COLOR_NONE` is only used for the `EMPTY` piece.

The constructor `Piece(color, type)` takes in a `PieceColor` and a `PieceType` and returns a `Piece` object with both properties.

```
[ ]: Piece(BLACK, KING)
```

Figure 3.22.: Creating a Piece object with color "BLACK" and type "KING"

```
[ ]: pieceon(startboard(), FILE_A, RANK_1)
```

Figure 3.23.: Retrieving the piece on the start board at position file A and rank 1

The function `ptype(piece)` takes a `piece` and returns the `PieceType` of the piece. The class `PieceType` contains the objects `PAWN`, `KNIGHT`, `BISHOP`, `ROOK`, `QUEEN` and `KING`.

```
[ ]: ptype(pieceon(startboard(), FILE_A, RANK_1))
```

Figure 3.24.: Retrieving the PieceType of the piece on the start board at position file A and rank 1

The color of a piece is also used for the side to move on the board object.

The function `sidetomove(board)` takes in a `board` and returns the PieceColor of the side that moves next.

```
[ ]: sidetomove(startboard())
```

Figure 3.25.: Retrieving the PieceColor of the side that moves next on the start board

3.3.7. Calculation Functions

The `see(board, move)` function takes in a board and a move and is a static exchange evaluator. It returns an Int referring to the piece value being lost or won in the near future after applying the given `move` on the given `board`. As an example if white can capture a protected queen with a bishop it will return an Int round about 7. 10 points for the queen minus 3 point for the lost bishop = 7 points. The function is only an estimate

of the material lost or won without applying any search algorithm. It includes X-ray attacks and defenses but does not consider tactical moves like pins, overloaded pieces, etc.

```
[ ]: b = fromfen("r7/4k3/8/4p3/8/2BK4/8/q7 w - - 0 1")
     see(b, movefromstring("c3a1"))
```

Figure 3.26.: Evaluating the material gain or loss after applying the move "c3a1"

3.3.8. PGN files

PGN (Portable game notation) can be used to save played games. It stores the moves that happened in the game, as well as some additional information about whom the players were, the date of the game or the location of the game.

The PGN file contains seven tags which are required. These are the `Event`, `Site`, `Date`, `Round`, `White`, `Black` and `result` tag. Additionally there are optional tags. The optional tags are `Annotator`, `PlyCount`, `TimeControl`, `Time`, `Termination`, `Mode` and `FEN` Tag.

Using Chess.jl PGN files can be created from the `Game` or `SimpleGame` object. This way games can be exported and saved in a PGN file. These can be imported later and transformed back into a `Game` object.

setheadervalue! function

The function `setheadervalue!(game, name, value)` takes in a `Game` or a `Simplegame`, the `name` of the header and a `value` for the header and sets these for the given `Game`. It will create a header if it has not been created or overwrites the value if it has been set.

Possible headers specify the Event, Site, Date, Round of the game as well as who played the game.

```
[ ]: g = Game()
     setheadervalue!(g, "Date", "2023.01.01")
     dateplayed(g)
```

Figure 3.27.: Setting the value of the "Date" header to "2023.01.01" for the given game

PGN Reader

The constructor `PGNReader(io)` takes in an io-Stream and returns a PGNReader which will be used to interact with PGN files.

```
[ ]: io = open("./../Games/RandomChess.pgn", "r")
     reader = PGNReader(io)
```

Figure 3.28.: Creating a PGNReader object

readgame function

The `readgame(PGNReader)` function takes in a `PGNReader` and tries to parse the file from the io-Stream of the PGNReader into a `Game` or `SimpleGame` object. For a full implementation of this see the `PGN_Import.ipynb` notebook.

```
[ ]: readgame(reader)
```

Figure 3.29.: Parsing the PGN file from the io-Stream

gametopgn function

The `gametopgn(g)` function takes in a `Game` or `Simplegame` and parses the game into a pgn file. Note: That this has no labels. For a complete implementation see the `PGN_Export.ipynb` notebook.

```
[ ]: g = Game()
     gametopgn(g)
```

Figure 3.30.: Parsing a Game or SimpleGame object into a PGN file using the gametopgn(g) function

3.4. Advanced Board Struct

This notebook implements the struct `AdvBoard` which combines additional attributes to the `Board` class of the chess.jl Library. These attributes relate to the `Board` directly meaning that a `Board` always has the same value for this attribute during any part of the calculation. Additionally, functions containing the Advanced Board are implemented in this notebook.

Added Attributes:

1. `score::Int64` is the static evaluation of the board in centipawns
2. `hash::UInt64` is the hash of the `Board`
3. `repCounter::Dict{UInt64, Int8}` is a dictionary counting how often a position has been reached. This is needed to check whether a 3-fold-repetition has occurred, resulting in a draw.

All of these attributes are not implemented by the Chess.jl Library. The `score` and `hash` are implemented using our choice of algorithms and therefore are not accounted for. The `repCounter` dictionary was added because the `Board` object only contains the information about the current position and does not have any information about the game. The important information is whether a threefold repetition has occurred, which leads to an immediate draw.

```
[ ]: using Chess
     using NBInclude
```

```
[ ]: @nbinclude("Repetition.ipynb")
```

```
[ ]: @nbinclude("ZobristHashing.ipynb")
```

```
[ ]: @nbinclude("EvaluatePosition.ipynb")
```

Figure 3.31.: Importing the Chess package and including the Repetition and ZobristHashing notebooks

This notebook contains the `Advanced Board` (short: `AdvBoard`) struct, which bundles additional information about the board to the `Board`. As the `::Board` Class of Chess.jl does not contain all the information we need to the board, this will be expanded to contain the following additional contents:

1. `score::Int64` is the static evaluation of the board in centipawn
2. `hash::UInt64` is the hash of the board
3. `repCounter::Dict{UInt64, Int8}` is a Dictionary that counts how often a position as occurred

The choice for a `mutual struct` was made after checking the source code of the Chess.jl implementation of the regular `Board`.

```
[ ]: mutable struct AdvBoard
         state::Board
         score::Int64
         hash::UInt64
         repCounter::Dict{UInt64, Int8}
     end
```

Figure 3.32.: Definition of the 'AdvBoard' mutable struct

Constructor for AdvBoard

The Constructor `AdvBoard` creates an `AdvBoard` from a `Board` and initializes all attributes automatically.

Arguments:

1. `board::Board` is the board that the AdvBoard should be representing

Returns an `AdvBoard` with calculated attributes.

[ ]:
```
function AdvBoard(board::Board)::AdvBoard
    score = evaluate_position(board)
    hash  = zobrist_hash(board)
    repCounter = Dict{UInt64, Int8}()
    incrementHash!(repCounter, hash)
    return AdvBoard(board, score, hash, repCounter)
end
```

Figure 3.33.: The AdvBoard constructor function, which takes a Board as input and returns an AdvBoard object with calculated attributes

domoveAdv! function

The function `domoveAdv!` has the same functionality for the `AdvBoard` as the `domove!` function for a `Board`. It applies the given move on the AdvBoard and changes all attributes according to the new position.

Arguments:

1. `aBoard::AdvBoard` is the chess board where the move should be applied
2. `move::Move` is the move that should be done

Returns a three-tuple including information needed to undo the move. These values should not be mutated.

[ ]:
```
function domoveAdv!(aBoard::AdvBoard, move::Move)::Tuple{Int64,␣
↪UInt64, UndoInfo}
    oldscore::Int64 = aBoard.score
    oldhash::UInt64 = aBoard.hash
    aBoard.score = evaluate_move(aBoard, move)
    aBoard.hash = zobrist_hash(aBoard.state, aBoard.hash, move)
    undoinfo = domove!(aBoard.state, move)
    incrementHash!(aBoard.repCounter, aBoard.hash)
```

```
    return (oldscore, oldhash, undoinfo)
end
```

Figure 3.34.: Function domoveAdv! implementation

undomoveAdv! function

The function `undomoveAdv!` has the same functionality for the `AdvBoard` as the `undomove!` function for a `Board`. It undoes the last move done on the AdvBoard and changes all attributes according to the new position.

Arguments:

1. `aBoard::AdvBoard` is the chess board where the move should be applied
2. `undoinfo` is the three-tuple returned from the `domoveAdv!` function

```
[ ]: function undomoveAdv!(ABoard::AdvBoard, undoinfo::Tuple{Int64, UInt64,␣
     ↪UndoInfo})
        decrementHash!(ABoard.repCounter, ABoard.hash)
        undomove!(ABoard.state, undoinfo[3])
        ABoard.score = undoinfo[1]
        ABoard.hash  = undoinfo[2]
     end
```

Figure 3.35.: Function undomoveAdv! implementation

```
[ ]: function clearRepCounter!(aBoard::AdvBoard)
        aBoard.repCounter = Dict{UInt64, Int8}()
     end
```

Figure 3.36.: Function clearRepCounter! implementation

evaluate_move function

In order to implement repetition control in the isterminal function, it is necessary to call the function with the Advanced Board. For this reason, we have overloaded the evaluate_move function to take an Advanced Board as input. The `evaluate_move` function takes in an advanced chess board `aBoard` and a `move` and returns the score of the board after performing the move. The function uses an incremental implementation, which means it only considers and calculates the differences between the old and new position.

The implementation of the evaluate_move function begins by performing the move on the board using the domove! function and storing the UndoInfo object returned by the function. The function then checks if the board is in a terminal state by calling the isterminal function with the Advanced Board as input. If the board is in a terminal state, the function returns the score of the board as evaluated by the terminal_evaluation function. Otherwise, the function calls the evaluate_move function with the board and the move as input parameters.

[ ]:
```
function evaluate_move(aBoard::AdvBoard, move::Move)::Int64
    nextHash = zobrist_hash(aBoard.state, aBoard.hash, move)
    if get(aBoard.repCounter, hash, 0) == 2
        return 0
    end
    undoinfo::UndoInfo = domove!(aBoard.state, move)

    if isterminal(aBoard.state)
        score::Int64 = terminal_evaluation(aBoard)
        undomove!(aBoard.state, undoinfo)
        return score
    end
    undomove!(aBoard.state, undoinfo)
    return evaluate_move(aBoard.state, move, aBoard.score)
end
```

Figure 3.37.: Function evaluate_move implementation

terminal_evaluation function

The terminal_evaluation function in Julia takes a Board object as an argument and returns an integer score representing the evaluation of the board in a given terminal state. The function checks if the board is in checkmate, stalemate, material draw, or rule 50 draw, and returns a score accordingly.

The function first checks if the board is in a checkmate position. If the board is in checkmate, the function returns a score of -100000 or 100000, depending on which player is in checkmate. A repetition position occurs when the same position occurs three times in a game, and the same player has the same possible moves. If the board is a draw, it returns a score of 0.

[ ]:
```
function terminal_evaluation(aBoard::AdvBoard)::Int64
    if ischeckmate(aBoard.state)
        return sidetomove(aBoard.state) == WHITE ? -100000 : 100000
    elseif isstalemate(aBoard.state) || ismaterialdraw(aBoard.state) ||␣
 ↪isrule50draw(aBoard.state) || isRepetition(aBoard)
        return 0
    end
end
```

Figure 3.38.: Function evaluation implementation

[ ]:
```
HTML(read(open("style.css"), String))
```

3.5. Detection of a Draw by Repetition

This notebook is used to detect a draw by repetition as this is function is not available in the `Chess.jl` library.

In chess a draw occurs if the same position is reached three times in the same game.

To detect a draw by repetition we associate a Dictionary repCounter with a `Board`. The dictionary pairs boards represented by their hash with the number of times the board has been reached in the game. Therefore the type of the Dictionary is `Dict{UInt64, Int8}()`. Each time a move is made the new board gets stored into the Dictionary with the value of one. If the board has already been stored the counter of this hash will increase by 1. If at any point a hash has the value of 3 a threefold repetition has occured and the game automatically results in a draw.

The function isRepetition returns a boolean whether the repCounter has a position that has occurred at least 3 times.

Arguments:

1. `repCounter::Dict{UInt64, Int8}()` repCounter Dictionary

Returns True if a threefold repetition has occurred. Otherwise returns False.

[ ]:
```
function isRepetition(repCounter::Dict{UInt64, Int8})
    return 3 in values(repCounter)
end
```

The function `incrementHash!` increments the counter of the given `hash` by one in the given `repCounter`. If the hash is not in the `repCounter` it will create a new entry with the hash and set the value to 1.

Arguments:

1. `repCounter::Dict{UInt64, Int8}()` repCounter Dictionary
2. `hash::UInt64` hash of the board

[ ]:
```
function incrementHash!(repCounter::Dict{UInt64, Int8}, hash::UInt64)
    !haskey(repCounter, hash) ? repCounter[hash] = 1 : repCounter[hash]␣
↪+= 1
end
```

The function `decrementHash` decrements the counter of the given `hash` by one in the given `repCounter`.

Arguments:

1. `repCounter::Dict{UInt64, Int8}()` repCounter Dictionary
2. `hash::UInt64` hash of the board

[ ]:
```
function decrementHash!(repCounter::Dict{UInt64, Int8}, hash::UInt64)
    if repCounter[hash] == 1
        delete!(repCounter, hash)
    else
        repCounter[hash] -= 1
    end
end
```

3.6. Simple Evaluation Function

An important component of a chess computer is the evaluation of a chess position, which is analyzed with an evaluation function. An evaluation function allows the chess computer to search through many possible moves and select those with the highest evaluations to improve its own position or deteriorate that of the opponent.

A Simplified Evaluation Function is an approach to evaluating the current chess position by calculating the values of each piece based on their position on the chessboard and summing them up. The sum is then compared to that of the opponent, using the values assigned to each piece as follows [2]:

```
P = 100
N = 320
B = 330
R = 500
Q = 900
K = 20000
```

Figure 3.39.: Assignment of values to each chess piece

A better position can help a player win the game by putting them in a more advantageous position, giving them more options for their next moves, and putting more pressure on their opponent. The evaluation function is used by chess engines to determine the best possible move based on the current position of the pieces, and to help determine which moves will lead to a better position.

```
[ ]: using Pkg
     #Pkg.add("Chess")
     using Chess
     using BenchmarkTools
     using NBInclude
```

Figure 3.40.: Julia package and module imports

```
[ ]: @nbinclude("Repetition.ipynb")
```

```
[ ]: @nbinclude("AdvancedBoard.ipynb")
```

Figure 3.41.: Including notebook files

3.6.1. Detecting the Endgame

The is_endgame function takes in one argument, board, which is the current state of the chess board.The function returns a boolean value of true if the game is in the endgame and false if it is not.

The function uses a few checks to determine whether the game has reached the endgame stage or not [2].

1. Both sides have no queens
2. Every side which has a queen has additionally no other pieces or one minor piece maximum.

The is_endgame function also uses some inline functions to calculate the number of minor pieces left on the board. The num_minor_pieces function takes in one argument, side, which is the color of the pieces being counted. It calculates the number of knights and bishops on the board for that side and returns the total count.

Finally, the is_in_endgame function takes in one argument, color, which is the color of the pieces being checked. It checks whether that color is in the endgame or not, based on the two checks described earlier. If the color is in the endgame, the function returns true, otherwise it returns false.

[ ]:
```
function is_endgame(board::Board)::Bool
    # Inline Functions to calculate the number of minor pieces
    num_minor_pieces(side) = squarecount(knights(board, side)) +␣
 ↪squarecount(bishops(board, side))
    # Check if Player White or Black is in endgame
    is_in_endgame(color) = isempty(queens(board, color)) ||␣
 ↪(num_minor_pieces(color) <= 1 && isempty(rooks(board, color)))
    return is_in_endgame(WHITE) && is_in_endgame(BLACK)
end
```

Figure 3.42.: Implementation is_endgame function

3.6.2. Determining the Value of Piece on a Square:

In the game of chess, each piece has a relative value based on their ability to move and control squares on the board. Understanding the values of the pieces is essential in evaluating the position of the game and determining which pieces to exchange and which to keep. In this research paper, we will explore how the values of the pieces are determined and how they help in evaluating a chess position.

Piecevalue:

In the game of chess, the values of the pieces are determined based on its importance and strength on the board. For example, the pawn is the weakest piece on the board, and its value is set at 100 points. In contrast, the queen is the most powerful piece, and its value is set at 900 points. The values of the other pieces fall between these two extremes, with the knight and bishop having a value of 320 and 330 points, respectively, and the rook having a value of 500 points.

In the following code, the global `Piece_Values` is set using a dictionary data structure. The dictionary maps the PieceType to its corresponding value in points.

```
[ ]: if !@isdefined(PIECE_VALUES)
        const PIECE_VALUES::Dict{PieceType, Int} = Dict([(PAWN,␣
     ↪100),(KNIGHT, 320),(BISHOP, 330),(ROOK, 500),(QUEEN, 900),(KING,␣
     ↪20000)])
     end
```

Figure 3.43.: Defining the global variable PIECE_VAL

The `get_value` function takes in 1 argument, `piece` which is a chess piece. It returns the value of the piece from global variable **PIECE_VALUES**.

```
[ ]: function get_value(piece::Piece)::Int64
        return get(PIECE_VALUES, ptype(piece), 0)
     end
```

Figure 3.44.: Implementation get values function

Square Values:

The position of the pieces on the board can impact their value, making it essential to consider the square values when evaluating the position of a piece. To achieve this, two tables called `White_Pawn_Square_Table` and `Black_Pawn_Square_Table` are used. These tables contain the values for each type of piece and square on the board, based on the position.

https://www.chessprogramming.org/Simplified_Evaluation_Function

```
[ ]: if !@isdefined(WHITE_PIECE_SQUARE_TABLES)
        const WHITE_PIECE_SQUARE_TABLES::Dict{PieceType,␣
     ↪Vector{Vector{Int8}}} = Dict(
            PAWN => [
            [  0,  0,  0,  0,  0,  0,  0,  0],
            [ 50, 50, 50, 50, 50, 50, 50, 50],
            [ 10, 10, 20, 30, 30, 20, 10, 10],
            [  5,  5, 10, 25, 25, 10,  5,  5],
            [  0,  0,  0, 20, 20,  0,  0,  0],
            [  5, -5,-10,  0,  0,-10, -5,  5],
            [  5, 10, 10,-20,-20, 10, 10,  5],
            [  0,  0,  0,  0,  0,  0,  0,  0]
        ],
            KNIGHT => [
```

```
        [-50,-40,-30,-30,-30,-30,-40,-50],
        [-40,-20,  0,  0,  0,  0,-20,-40],
        [-30,  0, 10, 15, 15, 10,  0,-30],
        [-30,  5, 15, 20, 20, 15,  5,-30],
        [-30,  0, 15, 20, 20, 15,  0,-30],
        [-30,  5, 10, 15, 15, 10,  5,-30],
        [-40,-20,  0,  5,  5,  0,-20,-40],
        [-50,-40,-30,-30,-30,-30,-40,-50]
    ],
        BISHOP => [
        [-20,-10,-10,-10,-10,-10,-10,-20],
        [-10,  0,  0,  0,  0,  0,  0,-10],
        [-10,  0,  5, 10, 10,  5,  0,-10],
        [-10,  5,  5, 10, 10,  5,  5,-10],
        [-10,  0, 10, 10, 10, 10,  0,-10],
        [-10, 10, 10, 10, 10, 10, 10,-10],
        [-10,  5,  0,  0,  0,  0,  5,-10],
        [-20,-10,-10,-10,-10,-10,-10,-20]
    ],
        ROOK => [
        [ 0,  0,  0,  0,  0,  0,  0,  0],
        [ 5, 10, 10, 10, 10, 10, 10,  5],
        [-5,  0,  0,  0,  0,  0,  0, -5],
        [-5,  0,  0,  0,  0,  0,  0, -5],
        [-5,  0,  0,  0,  0,  0,  0, -5],
        [-5,  0,  0,  0,  0,  0,  0, -5],
        [-5,  0,  0,  0,  0,  0,  0, -5],
        [ 0,  0,  0,  5,  5,  0,  0,  0]
    ],
        QUEEN => [
        [-20,-10,-10, -5, -5,-10,-10,-20],
        [-10,  0,  0,  0,  0,  0,  0,-10],
        [-10,  0,  5,  5,  5,  5,  0,-10],
        [ -5,  0,  5,  5,  5,  5,  0, -5],
        [  0,  0,  5,  5,  5,  5,  0, -5],
        [-10,  5,  5,  5,  5,  5,  0,-10],
        [-10,  0,  5,  0,  0,  0,  0,-10],
        [-20,-10,-10, -5, -5,-10,-10,-20]
```

```
    ],
        KING => [
        [-30,-40,-40,-50,-50,-40,-40,-30],
        [-30,-40,-40,-50,-50,-40,-40,-30],
        [-30,-40,-40,-50,-50,-40,-40,-30],
        [-30,-40,-40,-50,-50,-40,-40,-30],
        [-20,-30,-30,-40,-40,-30,-30,-20],
        [-10,-20,-20,-20,-20,-20,-20,-10],
        [ 20, 20,  0,  0,  0,  0, 20, 20],
        [ 20, 30, 10,  0,  0, 10, 30, 20]
    ]
    )
    const white_king_square_end_game_table::Array{Array{Int8, 1}, 1} = [
        [-50,-40,-30,-20,-20,-30,-40,-50],
        [-30,-20,-10,  0,  0,-10,-20,-30],
        [-30,-10, 20, 30, 30, 20,-10,-30],
        [-30,-10, 30, 40, 40, 30,-10,-30],
        [-30,-10, 30, 40, 40, 30,-10,-30],
        [-30,-10, 20, 30, 30, 20,-10,-30],
        [-30,-30,  0,  0,  0,  0,-30,-30],
        [-50,-30,-30,-30,-30,-30,-30,-50]
    ]
end
```

Figure 3.45.: Defining the global variable WHITE_PIECE_SQUARE_TABLES

```
[ ]: if !@isdefined(BLACK_PIECE_SQUARE_TABLES)
    const BLACK_PIECE_SQUARE_TABLES::Dict{PieceType,␣
 ↪Vector{Vector{Int8}}} = Dict(
        PAWN => [
        [  0,  0,  0,  0,  0,  0,  0,  0],
        [  5, 10, 10,-20,-20, 10, 10,  5],
        [  5, -5,-10,  0,  0,-10, -5,  5],
        [  0,  0,  0, 20, 20,  0,  0,  0],
        [  5,  5, 10, 25, 25, 10,  5,  5],
        [ 10, 10, 20, 30, 30, 20, 10, 10],
        [ 50, 50, 50, 50, 50, 50, 50, 50],
        [  0,  0,  0,  0,  0,  0,  0,  0]
    ],
```

```
        KNIGHT => [
        [-50,-40,-30,-30,-30,-30,-40,-50],
        [-40,-20,  0,  5,  5,  0,-20,-40],
        [-30,  5, 10, 15, 15, 10,  5,-30],
        [-30,  0, 15, 20, 20, 15,  0,-30],
        [-30,  5, 15, 20, 20, 15,  5,-30],
        [-30,  0, 10, 15, 15, 10,  0,-30],
        [-40,-20,  0,  0,  0,  0,-20,-40],
        [-50,-40,-30,-30,-30,-30,-40,-50]
    ],
        BISHOP => [
        [-20,-10,-10,-10,-10,-10,-10,-20],
        [-10,  5,  0,  0,  0,  0,  5,-10],
        [-10, 10, 10, 10, 10, 10, 10,-10],
        [-10,  0, 10, 10, 10, 10,  0,-10],
        [-10,  5,  5, 10, 10,  5,  5,-10],
        [-10,  0,  5, 10, 10,  5,  0,-10],
        [-10,  0,  0,  0,  0,  0,  0,-10],
        [-20,-10,-10,-10,-10,-10,-10,-20]
    ],
        ROOK => [
        [  0,  0,  0,  5,  5,  0,  0,  0],
        [ -5,  0,  0,  0,  0,  0,  0, -5],
        [ -5,  0,  0,  0,  0,  0,  0, -5],
        [ -5,  0,  0,  0,  0,  0,  0, -5],
        [ -5,  0,  0,  0,  0,  0,  0, -5],
        [ -5,  0,  0,  0,  0,  0,  0, -5],
        [  5, 10, 10, 10, 10, 10, 10,  5],
        [  0,  0,  0,  0,  0,  0,  0,  0]
    ],
        QUEEN => [
        [-20,-10,-10, -5, -5,-10,-10,-20],
        [-10,  0,  5,  0,  0,  0,  0,-10],
        [-10,  5,  5,  5,  5,  5,  0,-10],
        [  0,  0,  5,  5,  5,  5,  0, -5],
        [ -5,  0,  5,  5,  5,  5,  0, -5],
        [-10,  0,  5,  5,  5,  5,  0,-10],
        [-10,  0,  0,  0,  0,  0,  0,-10],
```

```
        [-20,-10,-10, -5, -5,-10,-10,-20]
    ],
        KING => [
        [ 20, 30, 10,  0,  0, 10, 30, 20],
        [ 20, 20,  0,  0,  0,  0, 20, 20],
        [-10,-20,-20,-20,-20,-20,-20,-10],
        [-20,-30,-30,-40,-40,-30,-30,-20],
        [-30,-40,-40,-50,-50,-40,-40,-30],
        [-30,-40,-40,-50,-50,-40,-40,-30],
        [-30,-40,-40,-50,-50,-40,-40,-30],
        [-30,-40,-40,-50,-50,-40,-40,-30]
    ]
    )
    const black_king_square_end_game_table::Array{Array{Int8, 1}, 1} = [
        [-50,-30,-30,-30,-30,-30,-30,-50],
        [-30,-30,  0,  0,  0,  0,-30,-30],
        [-30,-10, 20, 30, 30, 20,-10,-30],
        [-30,-10, 30, 40, 40, 30,-10,-30],
        [-30,-10, 30, 40, 40, 30,-10,-30],
        [-30,-10, 20, 30, 30, 20,-10,-30],
        [-30,-20,-10,  0,  0,-10,-20,-30],
        [-50,-40,-30,-20,-20,-30,-40,-50]
    ]
end
```

Figure 3.46.: Defining the global variable "BLACK_PIECE_SQUARE_TABLES"

The function `get_square_value` is used to evaluate the position of a chess piece on a board. As input parameters it takes a board (board::Board), a chess piece (piece::Piece) and a playing field (square::Square) on which the piece is located. The function calculates a numeric centipawn value that describes the strength or weakness of the current position of the piece.

The King uses two tables. The `table_opt` can be set to “end” to enforce the usage of the endgame table. This feature overrides the actual position of the board.

[ ]:
```
function get_square_value(board::Board, piece::Piece, square::Square,␣
↪table_opt::String="")::Int
    piece_type::PieceType = ptype(piece)
    if table_opt == "" && is_endgame(board)
```

```
        table_opt = "end"
    end

    if Chess.pcolor(piece) == WHITE
        square_table = WHITE_PIECE_SQUARE_TABLES[piece_type]
        if piece_type == KING && table_opt ==  "end"
            square_table = white_king_square_end_game_table
        end
    else
        square_table = BLACK_PIECE_SQUARE_TABLES[piece_type]
        if piece_type == KING && table_opt ==  "end"
            square_table = black_king_square_end_game_table
        end
    end

    squareString::String = tostring(square)
    x::Int8 = Int(squareString[1]) - Int('a')+1
    y::Int8 = parse(Int, squareString[2])
    return square_table[9 - y][x]
end
```

Figure 3.47.: Implementation get_square value function

Using the Piece Values and Square Values to Evaluate the Position of Pieces:

The `calc_score` function takes in 3 arguments, 1. `board` which is a chess board 1. `piece` which is a chess piece 1. `square` which is a chess square

The calc_score function is calculated by summing the value of the piece (get_value(piece)) and the square value (get_square_value(board, piece, square)). The calc_score function is used to evaluate the position of the chess pieces on the board.

[ ]:
```
function calc_score(board::Board, piece::Piece, square::Square)::Int64
    return get_value(piece) + get_square_value(board, piece, square)
end
```

Figure 3.48.: Implementation "calc_score" function

3.6.3. Evaluation of a Position

Non-incremental Evaluation of a Position

The function `evaluate_position` takes in one argument, `board` which is a chess board. It returns the value of the board. This function not incrementally evaluate one move.

If the board argument is a terminal state, the terminal_evaluation function is called, which returns the final score of the game.

The evaluate_position function iterates through each square on the board and calculates the score of each piece on that square. It uses the calc_score function to determine the value of the piece on the square. The score of the piece is then multiplied by 1 if it is a white piece or -1 if it is a black piece. Finally, the scores of all the pieces on the board are summed up to give the overall score of the board.

[ ]:
```
function evaluate_position(board::Board)::Int64
    if isterminal(board)
        return terminal_evaluation(board)
    end
    score = 0
    for x in 1:8,y in 1:8
        square = Square(SquareFile(x),SquareRank(y))
        piece = pieceon(board, square)
        if piece != EMPTY
            score += (Chess.pcolor(piece) == WHITE ? 1 : -1) *␣
 ↪calc_score(board, piece, square)
        end
    end
    return score
end
```

Figure 3.49.: Implementation "evaluate_position" function

Incrementle Evaluation of a Move

The idea behind incremental evaluation is to evaluate the effect of a move on the board's value by only considering the changes made by the move rather than evaluating the entire board from scratch.

It reduces the computational cost of evaluating a move. Since only the changes made by the move need to be evaluated, the search tree can be pruned more effectively, resulting in faster and deeper searches.

The function `valueCapturePiece` is used to update the value of a chessboard when the opponent captures a piece. It takes two input parameters:

1. `State::Board`: a chess board
2. `square::Square` the square where the capture occurred

If the square is empty, the function returns 0, as no piece was captured. Otherwise, it uses the calc_score function to determine the value of the captured piece, which is then subtracted from the current value of the chess board. The function returns the value of the captured piece as an integer.

[ ]:
```
function valueCapturePiece(board::Board, square::Square)::Int64
    capturePiece = pieceon(board, square)
    return capturePiece == EMPTY ? 0 : calc_score(board, capturePiece,␣
 ↪square)
end
```

Figure 3.50.: Implementation "valueCapturePiece" function

En passant is a unique chess move where a pawn can capture an opponent's pawn that has advanced two squares and landed on a square adjacent to it, as if the opposing pawn had only moved forward one square. https://www.chess.com/de/terms/en-passant-schachregeln

The function `calcEnPassant` is used to calculate capture with en passant. As input parameters, it takes a chess board `board` and a chess square `toSquare`. The toSquare parameter represents the destination square of the pawn that just moved.

Inside the function, the epsquare function is used to retrieve the en passant square on the board. If there is no en passant square, the function returns zero. Otherwise, it checks if the toSquare parameter matches the en passant square. Otherwise, it checks if the toSquare parameter matches the en passant square. If they match, the function returns the value of the captured pawn, otherwise it returns zero.

[ ]:
```
function calcEnPassant(board::Board, toSquare::Square)::Int64
    enpassantSquare = epsquare(board)
    if(enpassantSquare != SQ_NONE && enpassantSquare == toSquare)
        lastToMove = to(lastmove(board))
```

```
        return valueCapturePiece(board, lastToMove)
    end
    return 0
end
```

Figure 3.51.: Implementation "calcEnPassant" function

In the game of chess, castling is a special move that involves the king and one of the rooks. This move allows the king to move two squares towards the rook, and the rook moves to the square over which the king passed. chess.com/lessons/playing-the-game/castling

The function `calc_castle` is used to calculate the #rook# value of a castle move. As input parameters, it takes a chess board `board` and a chess move `move`. Castling kingside does not affect the value of the rook, since the value aren't changed after the move (f1/f8). Castling queenside adds +5 to the value of the position.The function checks if the move is a queensidecastle move. If it is not, the function returns zero. Otherwise, it returns +5 Rook Square https://www.chessprogramming.org/Simplified_Evaluation_Function.

[ ]:
```
function calc_castle(board::Board, piece::Piece, toSquare::Square)::
 ↪Int64
    if(ptype(piece) == KING && cancastlequeenside(board, Chess.
 ↪pcolor(piece)) && file(toSquare) == FILE_C)
        return  5
    end
    return 0
end
```

Figure 3.52.: Implementation "calc_castle" function

The `evaluate_move` function takes in a board with its current static centipawn `score` and a `move`, and it returns the static centipawn score after performing the move. The function uses an incremental implementation, which means it only considers and calculates the differences between the old and new position.

The function first does the move to obtain an undo information structure, which is used to restore the board state after the move has been evaluated. The function then determines if the move results in an endgame position and adjusts the score accordingly.

The function evaluates the piece on the starting square and determines the type of piece and whether it is black or white. The score is then inverted if the piece is black, to imitate the effect of black being white.

The algorithm then subtracts the value of the piece on the old square and adds the value of the piece on the new square. The score of a captured piece is also subtracted from the score.

The function also considers special events in chess, such as castling, promotion, en passant, and entry into endgame. If the piece is a pawn, the algorithm checks for en passant moves and adjusts the score accordingly. If the move results in a promotion, the piece is updated to the promoted piece. If the piece is a king, the algorithm checks for castling moves and adjusts the score accordingly. Castling kingside does not affect the value of the rook, while castling queenside adds +5 to the value of the position.

The algorithm then calculates the score of the piece on the new square using the calc_score function, which takes into account the position of the piece on the board.

If the move results in an endgame position, the algorithm adjusts the score based on the value of the piece on the board.

The score is then inverted back if the piece is black.

[ ]:
```
function evaluate_move(board::Board, move::Move, score::Int64)::Int64
    undoinfo::UndoInfo = domove!(board, move)
    is_after_move_endgame::Bool = is_endgame(board)
    undomove!(board, undoinfo)

    toMove::Square = to(move)
    fromMove::Square = from(move)
    piece::Piece = pieceon(board, from(move))
    pieceType::PieceType = ptype(piece)
    isBlack::Bool = Chess.pcolor(piece) == BLACK

    # invert score if piece is black (imitate black is white)
    score = isBlack ? -score : score

    # Sub Points for old Position from Piece
    score -= valueCapturePiece(board, fromMove)
    score += valueCapturePiece(board, toMove)

    # Handle en passant and promotion
    if pieceType == PAWN
        score += calcEnPassant(board, toMove)
        if ispromotion(move)
```

```
            piece = Piece(sidetomove(board), promotion(move))
        end
    end

    # Handle castling
    if pieceType == KING
        score += calc_castle(board, piece, toMove)
    end

    score += calc_score(board, piece, toMove)

    # invert back
    score = isBlack ? -score : score

    # entry into endgame
    if is_endgame(board) != is_after_move_endgame
        for kingPos in kings(board)
            k = pieceon(board, kingPos)
            score = Chess.pcolor(k) == BLACK ? -score : score
            if is_endgame(board)
                score -= get_square_value(board, k, kingPos, "middle")
                score += get_square_value(board, k, kingPos, "end")
            else
                score -= get_square_value(board, k, kingPos, "end")
                score += get_square_value(board, k, kingPos, "middle")
            end
            score = Chess.pcolor(k) == BLACK ? -score : score
        end
    end
    return score
end
```

Figure 3.53.: Implementation "evaluate_move" function

In order to implement repetition control in the isterminal function, it is necessary to call the function with the Advanced Board. For this reason, we have overloaded the evaluate_move function to take an Advanced Board as input. The `evaluate_move` function takes in an advanced chess board `aBoard` and a `move` and returns the score of the board after performing the move. The function uses an incremental implementation,

which means it only considers and calculates the differences between the old and new position.

The implementation of the evaluate_move function begins by performing the move on the board using the domove! function and storing the UndoInfo object returned by the function. The function then checks if the board is in a terminal state by calling the isterminal function with the Advanced Board as input. If the board is in a terminal state, the function returns the score of the board as evaluated by the terminal_evaluation function. Otherwise, the function calls the evaluate_move function with the board and the move as input parameters.

[ ]:
```
function evaluate_move(aBoard::AdvBoard, move::Move)::Int64
    # display(aBoard.state)
    # print(move)
    undoinfo::UndoInfo = domove!(aBoard.state, move)
    if isterminal(aBoard.state)
        score = terminal_evaluation(aBoard)
        undomove!(aBoard.state, undoinfo)
        return score
    end
    undomove!(aBoard.state, undoinfo)
    return evaluate_move(aBoard.state, move, aBoard.score)
end
```

Figure 3.54.: Implementation overload "evaluate_move" function

terminal_evaluation Function

The terminal_evaluation function in Julia takes a Board object as an argument and returns an integer score representing the evaluation of the board in a given terminal state. The function checks if the board is in checkmate, stalemate, material draw, or rule 50 draws, and returns a score accordingly.

The function first checks if the board is in a checkmate position. If the board is in checkmate, the function returns a score of -100000 or 100000, depending on which player is in checkmate. A repetition position occurs when the same position occurs three times in a game, and the same player has the same possible moves. If the board is a draw, it returns a score of 0.

[ ]:
```
function terminal_evaluation(aBoard::AdvBoard)::Int64
    if ischeckmate(aBoard.state)
```

```
        return sidetomove(aBoard.state) == WHITE ? -100000 : 100000
    elseif isRepetition(aBoard.repCounter) || isstalemate(aBoard.state)␣
 ↪||
           ismaterialdraw(aBoard.state)    || isrule50draw(aBoard.state)
        return 0
    end
end
```

Figure 3.55.: Implementation "terminal_evaluation" function

3.7. Random Chess

This notebook implements the simplest chess engine there is. It generates all legal moves and takes one of those at random and plays it.

[ ]:
```
using Pkg
# Pkg.add("Chess")
using Chess
using Random
```

Figure 3.56.: Julia package and module imports

The function `generateRandomMove(game)` takes in a `game` and returns a random move out of all legal moves in the current position.

[ ]:
```
function generateRandomMove(game::Game)
    return generateRandomMove(board(game))
end
```

Figure 3.57.: Implementation of the 'generateRandomMove' function

The function `generateRandomMove(game)` takes in a `board` and returns a random move out of all legal moves in the current position.

[ ]:
```
function generateRandomMove(board::Board)
    return rand(moves(board))
end
```

Figure 3.58.: Implementation of the 'generateRandomMove' function

The function `playRandomMove(game)` takes in a game and applies a random move generated by the `generateRandomMove(game)` function above to the inputted game.

[ ]:
```
function playRandomMove(game::Game)
    domove!(game, generateRandomMove(game))
end
```

Figure 3.59.: Implementation of the 'playRandomMove' function

3.8. Minimax

[ ]:
```
using Pkg
# Pkg.add("Chess")
using Chess
using Random

# Pkg.add("NBInclude")
using NBInclude
```

Figure 3.60.: Julia package and module imports

[ ]:
```
@nbinclude("AdvancedBoard.ipynb")
```

Figure 3.61.: Including notebook

[ ]:
```
gCache = Dict()
```

Figure 3.62.: Defining global Cache for Minimax

This notebook implements an engine which evaluates a chess position using the Minimax algorithm. The Minimax algorithm calculates all the different possibilities of the current position and chooses the best move. It assumes that all players try to make the best possible moves. From white's perspective, white chooses the best move of black's best moves. We have to limit the number of moves that Minimax will search for in the future due to the huge number of possibilities that chess has. Therefore, we stop the minimax algorithm at a certain depth.

The minimax algorithm can be described using these functions:

Note: $finished(s) = isterminal(s) | depth == 0$

1. $finished(s) \longrightarrow minValue(s) = evaluate\_position(s)$
2. $\neg finished(s) \longrightarrow minValue(s) = min(\{maxValue(n) | n \in nextStates(s)\})$
3. $finished(s) \longrightarrow maxValue(s) = evaluate\_position(s)$
4. $\neg finished(s) \longrightarrow maxValue(s) = min(\{minValue(n) | n \in nextStates(s)\})$

To explain the concept of the Minimax algorithm, let's consider a diagram with an example and a search depth of 4. Each node in the tree represents a game state, and the edges indicate the moves that can be made from that state. The algorithm works recursively by traversing the tree from bottom to top and left to right. When the algorithm reaches the search depth or reaches a terminal state of the game, a value is assigned to that state by the heuristic function. This is further explained in the EvaluatePosition.ipynb notebook. For the white player, the goal is to maximize this value to win, while the black player aims to minimize the value.

In the example, values (5, 6, 5, 9,...) are computed for the search depth. These values are propagated along the tree and represented as letters. After the algorithm calculates all the leaf nodes of state D, the maximum value is selected, and the value 6 is assigned to that state. Similarly, the value 9 is assigned to state E.

By applying this process by alternating between searching for the maximum (max) and minimum (min) values throughout the entire tree, the best move for the root node A can be determined. In this example, the best move for A is the move which leads to node B and has a value of 6.

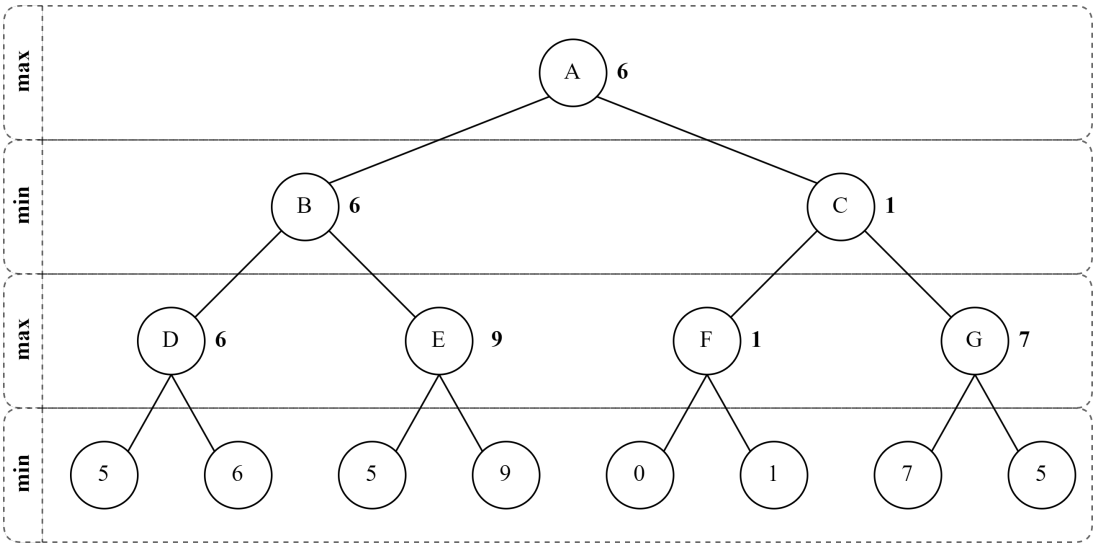

Figure 3.63.: Search Tree Minimax

3.8.1. Minimax-function non incremental no memoization

MaxValue

This function looks for the maximum value that white can achieve from a position. Meanwhile black tries to minimize the value of the position. Therefore white will take the maximal value of the minimum value of the next move that can happen.

Note: The depth is subtracted/added to give a better/worse rating to evaluate short mating paths as a petter path. (if there is a Mate in 3 (M3) and M1 Path then choose the M1 Path)

The function `maxValue_noIncNoMem` is the **non-incremental** implementation of the maxValue function.

Arguments:

1. `State::Board` is the current state
2. `depth::Int64` is the maximal plys minimax should calculate

Returns the maximal score white can achieve with perfect play.

[ ]:
```
function maxValue_noIncNoMem(State::Board, depth::Int64)::Int64
    if isterminal(State) || depth == 0
        return evaluate_position(State) - depth
    end
    return maximum([ minValue_noIncNoMem(domove(State, ns), depth-1)␣
 ↪for ns in moves(State) ])
end
```

Figure 3.64.: Implementation of maxValue no incremental no Memoization

MinValue

MinValue does the opposite of the MaxValue function. Therefore it looks for the minimum value that black can achieve from a position. Meanwhile white tries to maximize the value of the position. Therefore black will take the minimal value of the maximum value of the next move that can happen.

The function `minValue_noIncNoMem` is the **non-incremental** implementation of the minValue function.

Arguments:

1. `State::Board` is the current state
2. `depth::Int64` is the maximal plys minimax should calculate

Returns the minimal score black can achieve with perfect play.

[ ]:
```
function minValue_noIncnoMem(State::Board, depth::Int64)::Int64
    if isterminal(State) || depth == 0
      return evaluate_position(State) + depth
```

```
    end
    return minimum([ maxValue_noIncNoMem(domove(State, ns), depth-1)␣
 ↪for ns in moves(State) ])
end
```

Figure 3.65.: Implementation of minValue non incremental no Memoization

3.8.2. Minimax incremental but no Memoization

The function `maxValue_NoMem` is the **incremental** implementation of the maxValue function.

Arguments:

1. `State::Board` is the current state
2. `score::Int64` current evaluation of the board
3. `depth::Int64` is the maximal plys minimax should calculate

Returns the minimal score white can achieve with perfect play.

[ ]:
```
function maxValue_noMem(State::Board, score::Int64, depth::Int64)::
 ↪Int64
    if isterminal(State)
        return terminal_evaluation(State) - depth
    end
    if depth == 0
        return score
    end
    value = -Inf
    for move in moves(State)
        nextEval = evaluate_move(State, move, score)
        undoinfo = domove!(State, move)
        value = max(minValue_noMem(State, nextEval, depth-1), value)
        undomove!(State, undoinfo)
    end
    return value
end
```

Figure 3.66.: Implementation of maxValue no Memoization

The function `minValue_NoMem` is the **incremental** implementation of the minValue function.

Arguments:

1. `State::Board` is the current state
2. `score::Int64` current evaluation of the board
3. `depth::Int64` is the maximal plys minimax should calculate

Returns the minimal score black can achieve with perfect play.

```
[ ]: function minValue_noMem(State::Board, score::Int64, depth::Int64)::
      ↪Int64
        if isterminal(State)
            return terminal_evaluation(State) + depth
        end
        if depth == 0
            return score
        end
        value = Inf
        for move in moves(State)
            nextEval = evaluate_move(State, move, score)
            undoinfo = domove!(State, move)
            value = min(maxValue_noMem(State, nextEval, depth-1), value)
            undomove!(State, undoinfo)
        end
        return value
    end
```

Figure 3.67.: Implementation of minValue no Memoization

3.8.3. Minimax incremental and with Memoization

The function `maxValue` is the **incremental** implementation of the maxValue function using a cache.

Arguments:

1. `State::Board` is the current state
2. `depth::Int64` is the maximal plys minimax should calculate
3. `cache` is the cache that is used to save the entries

Returns the minimal score white can achieve with perfect play.

[ ]:
```
function maxValue(aBoard::AdvBoard, depth::Int64, cache)::Int64
    # Memoization
    entry = ("maxValue", aBoard.hash, depth)
    if entry in keys(cache)
        return cache[entry]
    end
    if isterminal(aBoard.state)
        result = terminal_evaluation(aBoard) - depth
    elseif depth == 0
        result = aBoard.score
    else
        result = -200000
        for move in moves(aBoard.state)
            undoinfo = domoveAdv!(aBoard, move)
            result = max(minValue(aBoard, depth-1, cache), result)
            undomoveAdv!(aBoard, undoinfo)
        end
    end
    # Save in Cache
    merge!(cache, Dict(entry => result))
    return result
end
```

Figure 3.68.: Implementation of maxValue

The function `minValue` is the **incremental** implementation of the minValue function that uses a cache.

Arguments:

1. `State::Board` is the current state
2. `depth::Int64` is the maximal plys minimax should calculate
3. `cache` is the cache that is used to save the entries

Returns the minimal score black can achieve with perfect play.

[ ]:
```
function minValue(aBoard::AdvBoard, depth::Int64, cache)::Int64
    # Memoization
    entry = ("minValue", aBoard.hash, depth)
    if entry in keys(cache)
        return cache[entry]
```

```
    end
    if isterminal(aBoard.state)
        result = terminal_evaluation(aBoard) + depth
    elseif depth == 0
        result = aBoard.score
    else
        result = 200000
        for move in moves(aBoard.state)
            undoinfo = domoveAdv!(aBoard, move)
            result = min(maxValue(aBoard, depth-1, cache), result)
            undomoveAdv!(aBoard, undoinfo)
        end
    end
    # Save in Cache
    merge!(cache, Dict(entry => result))
    return result
end
```

Figure 3.69.: Implementation of minValue

3.8.4. Minimax-function

non-incremental implementation

`minimax` is a function that implements the minimax algorithm to determine the best move for the current player at the current game state. This is the non-incremental implementation and does not have AdvBoard implemented into it.

Arguments:

1. `State::Board` An advanced chess board in the current state.
2. `score::Int64` current evaluation of the board
3. `depth::Int64`: The maximum depth to search.

Returns a `Tuple{Int64, Move}` containing the best score and the corresponding best move

[ ]:
```
function minimax_noMem(State::Board, score::Int64, depth::Int64)::
↪Tuple{Int64, Move}
    next_moves = moves(State)
    if sidetomove(State) == WHITE
```

```
        bestVal = maxValue_noMem(State, score, depth)
        BestMoves = [move for move in next_moves
                     if minValue_noMem(domove(State, move),␣
↪evaluate_move(State, move, score), depth-1) == bestVal]
    elseif sidetomove(State) == BLACK
        bestVal = minValue_noMem(State, score, depth)
        BestMoves = [move for move in next_moves
                     if maxValue_noMem(domove(State, move),␣
↪evaluate_move(State, move, score), depth-1) == bestVal]
    end
    BestMove = rand(BestMoves)
    return bestVal, BestMove
end
```

Figure 3.70.: Implementation of MiniMax no Memoization

incremental implementation

`minimax` is a function that implements the minimax algorithm to determine the best move for the current player at the current game state.

Arguments:

1. `aBoard::AdvBoard`: An advanced chess board in the current state.
2. `depth::Int64`: The maximum depth to search.
3. `cache::Dict{}`: a dictionary that stores previously calculated values

Returns a `Tuple{Int64, Move}` containing the best score and the corresponding best move

[ ]:
```
function minimax(aBoard::AdvBoard, depth::Int64, cache=Dict())::
↪Tuple{Int64, Move}
    BestMoves = []
    side = sidetomove(aBoard.state)
    bestVal = (side == WHITE) ? -200000 : 200000
    if side == WHITE
        bestVal = maxValue(aBoard, depth, cache)
        for move in moves(aBoard.state)
            undoinfo = domoveAdv!(aBoard, move)
            if minValue(aBoard, depth-1, cache) == bestVal
                append!(BestMoves, [move])
```

```
            end
            undomoveAdv!(aBoard, undoinfo)
        end
    elseif side == BLACK
        bestVal = minValue(aBoard, depth, cache)
        for move in moves(aBoard.state)
            undoinfo = domoveAdv!(aBoard, move)
            if maxValue(aBoard, depth-1, cache) == bestVal
                append!(BestMoves, [move])
            end
            undomoveAdv!(aBoard, undoinfo)
        end
    end
    BestMove = rand(BestMoves)
    return bestVal, BestMove
end
```

Figure 3.71.: Implementation of MiniMax

3.9. Zobrist hashing

[ ]:
```
# Pkg.add("Chess")
using Chess
using Random
using NBInclude
```

Figure 3.72.: Julia package and module imports

This notebook allows the user to hash the `Board` object into a bit string. This is used to save the `Board` in the cache efficiently.

Literature to Zobrist Hashing: https://www.chessprogramming.org/Zobrist_Hashing

Zobrist hashing is a hashing method which converts a chess board into an integer. This allows us to speed up operations with the cache. This is due to the bytesize a board needs. Using the hash, we can reduce the bytesize of a board by 228 bytes per board entry in the cache.

```
$ Base.summarysize(startboard()) # = 236
```

```
$ Base.summarysize(zobrist_hash(startboard())) # = 8
```

To calculate the integer a hashing table is created containing the following information:

1. an int for each piece on each square (12 unique pieces * 64 squares in total)
2. an int for each castling right for each side (4 in total)
3. an int for the file on which an en passant can occur (8 in total)
4. an int to indicate that it is black's turn (1 in total)

For each piece of information (as specified above) the corresponding int gets XOR'ed ($\oplus$)to the board's hash. The board's hash starts at a value of 0. For example: A white rook is located on the `A1` square. This means that the int for a white rook on the `A1` square is taken from the hashing table and XOR'ed with the hash value of 0. This is done for each piece, castling right, en passant file and turn color always modifying the value gotten from the previous calculations. This will always deliver a unique integer for each board, which is reproducible and reversible. The XOR operator is choosing to be able to reverse any operations done easily as applying the XOR operator twice will always return the same value.

$((a \oplus b) \oplus b) = b$

A hash can be easily calculated iteratively, as only each change needs to be XOR'ed with the previous hash. Given a hash `h` and moving a white rook from `A1` to `A2` means to XOR the value of a white rook on `A1` and XORing the value of a white rook on `A2` with the previous hash h.

The uniqueness of a hash is limited to the length of the integer. Collisions can happen after calculating round about 65.000 boards when using a 32-bit Hash and about 4 billion (4.000.000.000) boards when a using 64-bit hash.

3.9.1. Creating the hashing table

Each unique chess `piece` is assigned an integer. This will later be used in the Zobrist hashing table initialized below.

[ ]:
```
if !@isdefined(indices)
    const indices = Dict(
                PIECE_WP  => 1,
                PIECE_WR  => 2,
                PIECE_WN  => 3,
                PIECE_WB  => 4,
                PIECE_WQ  => 5,
                PIECE_WK  => 6,
```

```
                PIECE_BP  => 7,
                PIECE_BR  => 8,
                PIECE_BN  => 9,
                PIECE_BB  => 10,
                PIECE_BQ  => 11,
                PIECE_BK  => 12)
end
```

Figure 3.73.: Definition of global indices for chess pieces

The `ZobristHashing` struct contains 4 attributes:

1. The `pieces` attribute has a two-dimensional Array of 64-bit Integers. The size of this array will be 64x12. Each Square on a chess board is assigned 12 Integers representing the 12 chess pieces.
2. The `castling_rights` attribute consists of an integer array of length 4. It contains an Integer for each castling right. (White queen and King side and black queen and king side)
3. The `en_passant` attribute consists of an integer array of length 8. It contains an Integer for each file on the chess board. The file on which an en_passant square is available will be notated here.
4. The `turn` attribute consists of a single integer containing whether black has to move.

[ ]:
```
struct ZobristHashing
    pieces::Array{UInt64, 2}
    castling_rights::Array{UInt64, 1}
    en_passant::Array{UInt64, 1}
    turn::UInt64
end
```

Figure 3.74.: Definition of a struct for Zobrist hashing

Constructor for Zobrist hash table

The `ZobristHashing` function is the basic constructor of a `ZobristHashing` table and initializes the table after the constraints above.

Returns the `zobristHashing` table with pseudo-random values.

```
[ ]: function ZobristHashing()::ZobristHashing
         pieces = [rand(UInt64) for _ in 1:12, _ in 1:64]
         castling_rights = [rand(UInt64) for _ in 1:4]
         en_passant = [rand(UInt64) for _ in 1:8]
         turn = rand(UInt64)
         return ZobristHashing(pieces, castling_rights, en_passant, turn)
     end
```

Figure 3.75.: Implementation of the constructor for Zobrist hashing

Initialization of zobrist hashing table

```
[ ]: if !@isdefined(zobrist)
         const zobrist = ZobristHashing()
     end
```

Figure 3.76.: Definition of a global variable 'zobrist' using the ZobristHashing constructor

3.9.2. Non Incremental Zobrist Hashing

zobrist_hash function(non incremental)

The `zobrist_hash` function takes in a `board` and a `zobrist` hashing table and hashes the `board` into an integer using the hashing table. To convert the `board` into an int the function begins with 0 as the hash. It then iterates over each piece on the board and applies the xor operation to the value for the piece on its square gotten from the table with the hash. After that the values for the castling rights, en passant and move turn are also xor-ed to the hash. This creates a unique integer for each board. At the end this hash is returned and can be stored in a cache.

Arguments:

1. `board::Board` The board that needs to be hashed

Returns the `hash::UInt64` of the given board.

```
[ ]: function zobrist_hash(board::Board)::UInt64
         hash = 0
         # hash pieces
         for x in 1:8,y in 1:8
             square = Square(SquareFile(x), SquareRank(y))
             piece = pieceon(board, square)
             if piece != EMPTY
```

```
            hash = xor(hash, zobrist.pieces[indices[piece], square.val])
        end
    end

    # hash castling rights
    if cancastlekingside(board, WHITE)
        hash = xor(hash, zobrist.castling_rights[1])
    end
    if cancastlequeenside(board, WHITE)
        hash = xor(hash, zobrist.castling_rights[2])
    end
    if cancastlekingside(board, BLACK)
        hash = xor(hash, zobrist.castling_rights[3])
    end
    if cancastlequeenside(board, BLACK)
        hash = xor(hash, zobrist.castling_rights[4])
    end

    # hash en passant
    if epsquare(board) != SQ_NONE
        epfile = file(epsquare(board))
        hash = xor(hash, zobrist.
↪en_passant[Int(tochar(epfile))-Int('a') + 1])
    end

    # hash turn color
    if sidetomove(board) == BLACK
        hash = xor(hash, zobrist.turn)
    end
    return hash
end
```

Figure 3.77.: Implementation of the 'zobrist_hash' function for calculating the Zobrist hash of a chess board

3.9.3. Incremental Zobrist Hashing

updateCastleRightsHash function

The function `updateCastleRightsHash` is an auxiliary for the `zobrist_hash` function. It checks whether the move done will change any castling rights and modifies the hash accordingly. If no castling rights are modified by the move the hash will not be changed.

Arguments:

1. `board::Board` The current board
2. `hash::UInt64` A hash value after some calculations of the `zobrist_hash` function
3. `move::Move` The move for which the hash needs to be calculated

Returns the `hash::UInt64` of the board after applying the castling rights to the hash.

```
[ ]: function updateCastleRightsHash(board::Board, hash::UInt64, move::Move)
     ↪::UInt64
         for (color, offset) in [(WHITE, 0), (BLACK, 2)]
             isKingSideCastle::Bool = cancastlekingside(board, color)
             isQueenSideCastle::Bool = cancastlequeenside(board, color)
             if isKingSideCastle || isQueenSideCastle
                 undoinfo = domove!(board, move)
                 if isKingSideCastle && !cancastlekingside(board, color)
                     hash = xor(hash, zobrist.castling_rights[1 + offset])
                 end
                 if isQueenSideCastle && !cancastlequeenside(board, color)
                     hash = xor(hash, zobrist.castling_rights[2 + offset])
                 end
                 undomove!(board, undoinfo)
             end
         end
         return hash
     end
```

Figure 3.78.: Implementation of the updateCastleRightsHash function for updating the Zobrist hash after a move that affects the castling rights

updateEnPassantHash function

The function `updateEnPassantHash` is an auxiliary for the `zobrist_hash` function. It checks whether the move done will create an en passant square and modifies the hash accordingly. If no en passant square is made by the move the hash will not be modified.

Arguments:

1. `board::Board` The current board
2. `hash::UInt64` A hash value after some calculations of the `zobrist_hash` function
3. `move::Move` The move for which the hash needs to be calculated

Returns the `hash::UInt64` of the board after applying the en passant square to the hash.

```
[ ]: function updateEnPassantHash(board::Board, hash::UInt64, move::Move)::
     ↪UInt64
         undoinfo = domove!(board, move)
         if epsquare(board) != SQ_NONE
             epfile = file(epsquare(board))
             hash = xor(hash, zobrist.
     ↪en_passant[Int(tochar(epfile))-Int('a') + 1])
         end
         undomove!(board, undoinfo)
         return hash
     end
```

Figure 3.79.: Implementation of the updateEnPassantHash function for updating the Zobrist hash after a move that affects the en passant square

zobrist_hash function

The function `zobrist_hash` returns the zobrist hash of the given board after doing the given move. It only calculates the differences made via the given move and does not iterate over the whole board. Therefore, the performance of this function compared to the `zobrist_hash(board)` function is much better.

Arguments:

1. `board::Board` The current board
2. `hash::UInt64` The hash of the current board
3. `move::Move` The move for which the hash needs to be calculated

Returns the `hash::UInt64` of the board after doing the given move on the current board.

```
[ ]: function zobrist_hash(board::Board, hash::UInt64, move::Move)::UInt64
         toMove = to(move)
         fromMove = from(move)
         piece = pieceon(board, from(move))
         pieceType = ptype(piece)
```

```
    color = Chess.pcolor(piece)
    hash = xor(hash, zobrist.pieces[indices[piece], fromMove.val])
    hash = xor(hash, zobrist.pieces[indices[piece], toMove.val])
    #Capturing pieces
    capturePiece = pieceon(board, toMove)
    if capturePiece != EMPTY
        hash = xor(hash, zobrist.pieces[indices[capturePiece], toMove.
↪val])
    end

    #Castling
    if pieceType == KING
        if cancastlekingside(board, WHITE) && toMove == SQ_G1
            hash = xor(hash, zobrist.pieces[indices[PIECE_WR], SQ_H1.
↪val])
            hash = xor(hash, zobrist.pieces[indices[PIECE_WR], SQ_F1.
↪val])
        elseif cancastlequeenside(board, WHITE) && toMove == SQ_C1
            hash = xor(hash, zobrist.pieces[indices[PIECE_WR], SQ_A1.
↪val])
            hash = xor(hash, zobrist.pieces[indices[PIECE_WR], SQ_D1.
↪val])
        elseif cancastlekingside(board, BLACK) && toMove == SQ_G8
            hash = xor(hash, zobrist.pieces[indices[PIECE_BR], SQ_H8.
↪val])
            hash = xor(hash, zobrist.pieces[indices[PIECE_BR], SQ_F8.
↪val])
        elseif cancastlequeenside(board, BLACK) && toMove == SQ_C8
            hash = xor(hash, zobrist.pieces[indices[PIECE_BR], SQ_A8.
↪val])
            hash = xor(hash, zobrist.pieces[indices[PIECE_BR], SQ_D8.
↪val])
        end
    end

    hash = updateCastleRightsHash(board,hash, move)

    #En passant
```

```
    enpassantSquare = epsquare(board)
    if enpassantSquare != SQ_NONE
        if enpassantSquare == toMove
            lastToMove = to(lastmove(board))
            if color ==WHITE
                hash = xor(hash, zobrist.pieces[indices[PIECE_WP],␣
↪lastToMove.val])
            else
                hash = xor(hash, zobrist.pieces[indices[PIECE_BP],␣
↪lastToMove.val])
            end
        end
        epfile = file(epsquare(board))
        hash = xor(hash, zobrist.
↪en_passant[Int(tochar(epfile))-Int('a') + 1])
    end

    hash = updateEnPassantHash(board,hash, move)
    #Promotion
    if ispromotion(move)
        hash = xor(hash, zobrist.pieces[indices[piece], toMove.val])
        promoPiece = Piece(color,promotion(move))
        hash = xor(hash, zobrist.pieces[indices[promoPiece], toMove.
↪val])
    end

    #Turn
    hash = xor(hash, zobrist.turn)
end
```

Figure 3.80.: incrementally implementation of the 'zobrist_hash' function for calculating the Zobrist hash of a chess move

3.10. Memoization

This notebook contains all functions needed for memoization function calls that we have already calculated. This will greatly reduce the runtime of the program.

The general idea is implemented in this function:

1. If the value has been calculated: Look up the value in the global Cache using the arguments of the function as the key and return that value.
2. If the value has not been calculated: Calculate the value as usual and before returning the value add the value to the global Cache for later lookup.

```
function memoize(f)
    global gCache
    function f_memoized(b, s, d)
        if (b, s, d) in keys(gCache)
            return gCache[args]
        end
        result = f(b, s, d)
        merge!(gCache, Dict((b, s, d) => result))
        return result
    end
    return f_memoized
end
```

[ ]:
```
using Chess
using NBInclude
```

Figure 3.81.: Julia package and module imports

[ ]:
```
@nbinclude("ZobristHashing.ipynb")
```

Figure 3.82.: Including notebook files

The function `initCache()` returns an empty Dictionary of type `Dict{UInt64, Tuple{String, Int64, Int64}}` which will be used for the global Cache variable. As the key we use the hash of a board. The value consists of a tuple consisting of 3 arguments [14]:

1. `flag::String` is "=", "<=" or ">=" representing if the value stored is the exact value or a upper or lower
2. `value::Int64` is the value of the position
3. `depth::Int64` is the maximum depth at which the value has been calculated at. boundary

[ ]:
```
function initCache()
    return Dict{UInt64, Tuple{String, Int64, Int64}}()
end
```

Figure 3.83.: Implementation function initCache()

The `evaluate` function adds memoization to any function using the alpha-beta-pruning algorithm. This function is used to evaluate the given board state using the provided evaluation function (f) and cache. It performs alpha-beta pruning to reduce the number of nodes that need to be evaluated, and checks the cache to see if the given state has already been evaluated before.

Arguments:

1. `aBoard::AdvBoard` is the current state
2. `f::Function` takes in either the function that needs to be calculated and memorized
3. `depth::Int64` is the number of half moves the engine should analyze before terminating
4. `cache::Dict{UInt64, Tuple{String, Int64, Int64}}` is the cache used
5. `alpha::Int64` is a minimal value that has been calculated during the recursive process
6. `beta::Int64` is a maximal value that has been calculated during the recursive process
7. `flagQuiesce::Bool` is an optional flag specifies whether the quiesce-search should be used.

The function `evaluate` evaluates the same result as the function `f`. Additionally it saves calculated results in the `cache`. And uses any entries in the `cache` if the same function has already been called.

[ ]:
```
function evaluate(aBoard::AdvBoard, f::Function, depth::Int64, cache::
↪Dict{UInt64, Tuple{String, Int64, Int64}},
                  alpha::Int64, beta::Int64, flagQuiesce::Bool=false)::
↪Int64
    if get(aBoard.repCounter, aBoard.hash, 0) >= 3
        return 0
    end
    tuple::Tuple{String, Int64, Int64} = get(cache, aBoard.hash, ("",␣
↪0, 0))
    if tuple != ("", 0, 0)
        flag::String, v::Int64, d::Int64 = tuple
        if d >= depth
            if ((flag == "=") || (flag == "<=" && v <= alpha) || (flag␣
↪== ">=" && beta <= v))
                return v
```

```
            elseif flag == "<="
                beta = min(beta, v)
            elseif flag == ">="
                alpha = max(alpha, v)
            end
        end
    end
    # no value stored in gCache for State or depth of stored State has␣
 ↪less depth than required
    if flagQuiesce
        v = f(aBoard, depth, cache, alpha, beta, flagQuiesce)
    else
        v = f(aBoard, depth, cache, alpha, beta)
    end
    store_cache(aBoard.hash, depth, cache, alpha, beta, v)
    return v
end
```

Figure 3.84.: Implementation function evaluate()

The `store_cache` is a helping function that stores values into the `cache`.

Arguments:

1. `hash` is the hash that needs to be stored
2. `depth` is the number of half moves the engine should analyze before terminating
3. `cache::Dict{UInt64, Tuple{String, Int64, Int64}}` is the cache used
4. `alpha` is a minimal value that has been calculated during the recursive process
5. `beta` is a maximal value that has been calculated during the recursive process
6. `v` is the value that got calculated by the `evaluate` function that needs to be stored

[ ]:
```
function store_cache(hash::UInt64, depth::Int64,
                     cache::Dict{UInt64, Tuple{String, Int64, Int64}},␣
 ↪alpha::Int64, beta::Int64, v::Int64)
    if v <= alpha
        push!(cache, hash => ("<=", v, depth))
    elseif v < beta
        push!(cache, hash => ("=", v, depth))
    else # beta <= v
        push!(cache, hash => (">=", v, depth))
```

```
    end
end
```

Figure 3.85.: Implementation function store_cache()

3.11. Alpha-Beta Pruning

Alpha-Beta pruning is an optimization of the MiniMax algorithm, which is described in the MiniMax.ipynb notebook. The Minimax algorithm considers all possible moves and their effects on the game tree to find the best move for a player. However, this algorithm can be computationally intensive as it searches the entire game tree up to a certain depth.

Alpha-Beta pruning allows for a significant reduction of the search space by cutting off certain branches of the game tree. Therefore, the same result as the MiniMax algorithm can be evaluated without having to analyze all possible moves. This is achieved by using two values: alpha and beta.

Alpha represents the best move found so far for the player trying to maximize their score. It serves as a lower boundary for the value that this player can achieve. On the other hand, Beta represents the best move found so far for the player trying to minimize the score. It serves as an upper boundary for the value that this player can achieve. During the search, the values of Alpha and Beta are updated to consider the best moves known so far for the players.

In the example, it becomes clear how Alpha-Beta pruning works. Starting with the assignment of the value 6 to state D, the upper boundary of state B is also set to 6. This means that state E must be less than 6 to be considered as the best move.

In state E, the first leaf node is evaluated with a value of 9. Since the Max player now knows that they can achieve at least 9 points, the lower boundary (Alpha) of state E is set to 9. Due to this update, the algorithm does not need to calculate any further leaf nodes from state E since a better result is already known. This result cannot be improved upon since there is already a better result for the minimizing player.

A similar case occurs with state C. By evaluating state B, the lower boundary is set to 6. State F sets the upper boundary to 1. Since the maximizing player has a better option (B), the further sub-branches are not considered.

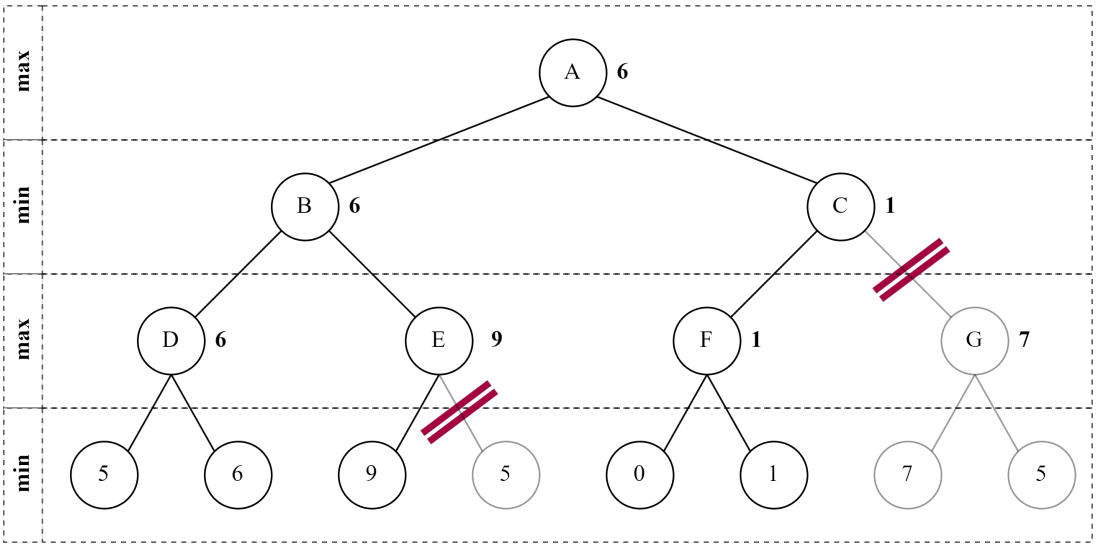

Figure 3.86.: alpha_beta.png

By using Alpha-Beta pruning, large portions of the game tree can be eliminated when they have no impact on the final decision. This leads to significant savings in computation time and allows the algorithm to search more efficiently for the best move.

The algorithm can be summarzied using the following equations.

$s \in states$

1. $\alpha \leq maxValue(s) \leq \beta \longrightarrow alphaBetaMax(s, \alpha, \beta) = maxValue(s)$
2. $maxValue(s) < \alpha \longrightarrow alphaBetaMax(s, \alpha, \beta) \leq \alpha$
3. $\beta < maxValue(s) \longrightarrow \beta \leq alphaBetaMax(s, \alpha, \beta)$
4. $\alpha \leq minValue(s) \leq \beta \longrightarrow alphaBetaMax(s, \alpha, \beta) = minValue(s)$
5. $minValue(s) < \alpha \longrightarrow alphaBetaMax(s, \alpha, \beta) \leq \alpha$
6. $\beta < minValue(s) \longrightarrow \beta \leq alphaBetaMax(s, \alpha, \beta)$

A comprehensive validation of this technique is presented in the work of Donald E. Knuth and Ronald W. Moore [15], which provides further insight into the effectiveness of this algorithm.

[ ]:
```
using Pkg
# Pkg.add("Chess")
using Chess
using Random
```

```
# Pkg.add("NBInclude")
using NBInclude
```

Figure 3.87.: Julia package and module imports2

```
[ ]: @nbinclude("EvaluatePosition.ipynb")
```

```
[ ]: @nbinclude("AdvancedBoard.ipynb")
```

```
[ ]: @nbinclude("Memoization.ipynb")
```

Figure 3.88.: Including notebook files needed for Alpha-Beta Pruning

3.11.1. AlphaBetaMax

The `alphaBetaMax_noMem` function takes in 3 arguments and 2 optional arguments.

1. `aBoard::AdvBoard` is a chess board in the current state
2. `depth` is the number of halfmoves the engine should analyze before terminating
3. `alpha` is optional and is default to -Infinity. Alpha is a minimal value that has been calculated during the recursive process
4. `beta` is optional and is default to Infinity . Beta is a maximal value that has been calculated during the recursive process

The function returns the maximal centipawn evaluation of the current position for the player playing white where both players have played the optimal moves according to the algorithm and terminating after the given depth. This function does not use Memoization.

We will discuss the requirements for the alphaBetaMax function and how it satisfies these requirements: - One of the requirements for the alphaBetaMax function is that it should compute the same result as the maxValue function when the maximum value of a node is between and . This means that:

$$\alpha \leq maxValue(s) \leq \beta \longrightarrow alphaBetaMax(s, \alpha, \beta) = maxValue(s)$$

This requirement ensures that the alphaBetaMax function does not prune any nodes that could potentially lead to the optimal solution. - Another requirement for the alphaBetaMax function is that it should return a value less than or equal to when the maximum value of a node is less than . This means that:

$$maxValue(s) < \alpha \longrightarrow alphaBetaMax(s, \alpha, \beta) \leq \alpha$$

- The last requirement for the alphaBetaMax function is that it should return a value greater than or equal to when the maximum value of a node is greater than . This means that:

$$\beta < maxValue(s) \longrightarrow \beta \leq alphaBetaMax(s, \alpha, \beta)$$

[ ]:
```
function alphaBetaMax_noMem(aBoard::AdvBoard, depth::Int64, alpha::
↪Int64=-200000, beta::Int64=200000)::Int64
    if isterminal(aBoard.state)
        return terminal_evaluation(aBoard) - depth
    end
    if depth == 0
        return aBoard.score
    end
    for move in moves(aBoard.state)
        Undo = domoveAdv!(aBoard, move)
        value = alphaBetaMin_noMem(aBoard, depth - 1, alpha, beta)
        undomoveAdv!(aBoard, Undo)
        if value >= beta
            return value
        end
        alpha = max(alpha, value)
    end
    return alpha
end
```

Figure 3.89.: Implementation of the Alpha-Beta Pruning algorithm (maximizing player)

3.11.2. AlphaBetaMin

The `alphaBetaMin_noMem` function takes in 3 arguments and 2 optional arguments.

1. `aBoard::AdvBoard` is a chess board in the current state
2. `depth` is the number of halfmoves the engine should analyze before terminating
3. `alpha` is optional and is default to -Infinity. Alpha is a minimal value that has been calculated during the recursive process

4. `beta` is optional and is default to Infinity . Beta is a maximal value that has been calculated during the recursive process

The function returns the minimal centipawn evaluation of the current position for the player playing black where both players have played the optimal moves according to the algorithm and terminating after the given depth. This function does not use Memoization.

Similar to "alphaBetaMax", there are certain requirements that must be met for "alphaBetaMin" to ensure a correct approximation: - If "maxValue(s)" is between the values of and , "alphaBetaMin" computes the same value as "minValue(s)", i.e.

$$\alpha \leq minValue(s) \leq \beta \longrightarrow alphaBetaMax(s, \alpha, \beta) = minValue(s)$$

This means that "alphaBetaMin" returns the exact minimum value if it is between and . - If "minValue(s)" is less than , the value returned by "alphaBetaMin" must be less than or equal to , i.e.

$$minValue(s) < \alpha \longrightarrow alphaBetaMax(s, \alpha, \beta) \leq \alpha$$

This means that "alphaBetaMin" returns a value that is not smaller than if the minimum value is smaller than . This optimizes the algorithm, as there is no point in examining further moves if the minimum value is already smaller than the best known value. - Similarly, if "minValue(s)" is greater than , the value returned by "alphaBetaMin" must be greater than or equal to , i.e.

$$\beta < minValue(s) \longrightarrow \beta \leq alphaBetaMax(s, \alpha, \beta)$$

This means that "alphaBetaMin" returns a value that is not larger than if the minimum value is greater than . This is another optimization of the algorithm, as there is no point in examining further moves if the minimum value is already greater than the best known value.

[ ]:
```
function alphaBetaMin_noMem(aBoard::AdvBoard, depth::Int64, alpha::
↪Int64=-200000, beta::Int64=200000)::Int64
    if isterminal(aBoard.state)
        return terminal_evaluation(aBoard) + depth
    end
    if depth == 0
```

```
        return aBoard.score
    end
    for move in moves(aBoard.state)
        Undo = domoveAdv!(aBoard, move)
        value = alphaBetaMax_noMem(aBoard, depth - 1, alpha, beta)
        undomoveAdv!(aBoard, Undo)
        if value <= alpha
            return value
        end
        beta = min(beta, value)
    end
    return beta
end
```

Figure 3.90.: Implementation of the Alpha-Beta Pruning algorithm (minimizing player)

3.11.3. Alpha-Beta-Pruning function

The `alphaBetaPruning_noMem` function takes in 3 arguments

1. `aBoard::AdvBoard` is a chess board in the current state
2. `depth` is the number of halfmoves the engine should analyze before terminating

The function returns the best value and the best move the moving player can play in the current position. It calls the alpha-beta-pruning algorithm. If multiple moves are found which result in the best evaluation a random move will be chosen. This function does not use Memoization.

The `alphaBetaPruning_noMem` function calls the alpha-beta pruning algorithm. It first generates a list of possible moves from the current state. Then it iterates over each move, evaluates the heuristic of the current state using the evaluate_move function, and calls the alphaBetaMin_noMem or alphaBetaMax_noMem function depending on which player is moving. The algorithm prunes parts of the game tree that cannot lead to a better score than the current alpha or beta value.

[ ]:
```
function alphaBetaPruning_noMem(aBoard::AdvBoard, depth::Int64)::
 ↪Tuple{Int64, Move}
    next_moves = moves(aBoard.state)
    BestMoves = []
    if sidetomove(aBoard.state) == WHITE
```

```
        bestVal = alphaBetaMax_noMem(aBoard, Int(depth))
        for move in next_moves
            Undo = domoveAdv!(aBoard, move)
            if alphaBetaMin_noMem(aBoard, depth - 1) == bestVal
                append!(BestMoves, [move])
            end
            undomoveAdv!(aBoard, Undo)
        end
    elseif sidetomove(aBoard.state) == BLACK
        bestVal = alphaBetaMin_noMem(aBoard, depth)
        for move in next_moves
            Undo = domoveAdv!(aBoard, move)
            if alphaBetaMax_noMem(aBoard, depth - 1) == bestVal
                append!(BestMoves, [move])
            end
            undomoveAdv!(aBoard, Undo)
        end
    end
    BestMove = rand(BestMoves)
    return bestVal, BestMove
end
```

Figure 3.91.: Implementation of the Alpha-Beta Pruning algorithm (complete)

3.11.4. Alpha-beta-Pruning with Memoization

Transposition tables store previously computed positions and their associated values, allowing the algorithm to avoid redundant computations and speed up the search process [16] (p. 8). In this research paper, the transposition tables are referred to as memoization, and the values are stored in a cache. The cache is implemented as a dictionary, where the key is a hash value representing a state, and the value is a tuple containing a flag, the evaluation of the state, and the remaining search depth. A detailed description of the cache can be found in the Memoization Notebook section.

[ ]:
```
@nbinclude("Memoization.ipynb")
```

AlphaBetaMax function with Memoization

The `alphaBetaMax` function takes in 4 arguments and 2 optional arguments.

1. `aBoard::AdvBoard` is a chess board in the current state
2. `depth` is the number of halfmoves the engine should analyze before terminating
3. `cache` is a dictionary which stores the calculated values
4. `alpha` is optional and is default to -Infinity. Alpha is a minimal value that has been calculated during the recursive process
5. `beta` is optional and is default to Infinity . Beta is a maximal value that has been calculated during the recursive process

The function returns the maximal centipawn evaluation of the current position for the player playing white where both players have played the optimal moves according to the algorithm and terminating after the given depth. This function does use Memoization meaning it saves and uses calculated values stored the `Cache`.

[ ]:
```
function alphaBetaMax(aBoard::AdvBoard, depth::Int64,
                      cache::Dict{UInt64, Tuple{String, Int64,␣
↪Int64}}, alpha::Int64=-200000, beta::Int64=200000)::Int64
    if isterminal(aBoard.state)
        return terminal_evaluation(aBoard) - depth
    end
    if depth == 0
        return aBoard.score
    end
    for move in moves(aBoard.state)
        Undo = domoveAdv!(aBoard, move)
        value = evaluate(aBoard, alphaBetaMin, depth - 1, cache, alpha,␣
↪beta)
        undomoveAdv!(aBoard, Undo)
        if value >= beta
            return value
        end
        alpha = max(alpha, value)
    end
    return alpha
end
```

Figure 3.92.: Implementation of Alpha-Beta Pruning with Memoization (maximizing player)

AlphaBetaMin function with memoization

The Alpha-Beta-Min function takes in 4 arguments and 2 optional arguments.

1. `aBoard::AdvBoard` is a chess board in the current state
2. `depth` is the number of halfmoves the engine should analyze before terminating
3. `cache` is a dictionary which stores the calculated values
4. `alpha` is optional and is default to -Infinity. Alpha is a minimal value that has been calculated during the recursive process
5. `beta` is optional and is default to Infinity . Beta is a maximal value that has been calculated during the recursive process

The function returns the minimal centipawn evaluation of the current position for the player playing black where both players have played the optimal moves according to the algorithm and terminating after the given depth. This function does use Memoization meaning it saves and uses calculated values stored the `Cache`.

[ ]:
```
function alphaBetaMin(aBoard::AdvBoard, depth::Int64,
                      cache::Dict{UInt64, Tuple{String, Int64,␣
↪Int64}}, alpha::Int64=-200000, beta::Int64=200000)::Int64
    if isterminal(aBoard.state)
        return terminal_evaluation(aBoard) + depth
    end
    if depth == 0
        return aBoard.score
    end
    for move in moves(aBoard.state)
        Undo = domoveAdv!(aBoard, move)
        value = evaluate(aBoard, alphaBetaMax, depth - 1, cache ,alpha,␣
↪beta)
        undomoveAdv!(aBoard, Undo)
        if value <= alpha
            return value
        end
        beta = min(beta, value)
    end
    return beta
end
```

Figure 3.93.: Implementation of Alpha-Beta Pruning with Memoization (minimizing player)

The `alphaBetaPruning` function takes in 3 arguments and 1 optional argument.

1. `aBoard::AdvBoard` is a chess board in the current state
2. `depth` is the number of halfmoves the engine should analyze before terminating

3. cache is optional and is default empty dictionary. Cache is a dictionary which stores the calculated values

The function returns the best value and the best move the moving player can play in the current position. It calls the alpha-beta-pruning algorithm. If multiple moves are found which result in the best evaluation a random move will be chosen.

```
[ ]: function alphaBetaPruning(aBoard::AdvBoard, depth::Int64,
                               cache::Dict{UInt64, Tuple{String, Int64,␣
     ↪Int64}} = initCache())::Tuple{Int64, Move}
        next_moves = moves(aBoard.state)
        BestMoves = []
        if sidetomove(aBoard.state) == WHITE
            bestVal = evaluate(aBoard, alphaBetaMax, depth, cache, -200000,␣
     ↪200000)
            for move in next_moves
                Undo = domoveAdv!(aBoard, move)
                if evaluate(aBoard, alphaBetaMin, depth-1, cache,-200000,␣
     ↪200000) == bestVal
                    append!(BestMoves, [move])
                end
                undomoveAdv!(aBoard, Undo)
            end
        elseif sidetomove(aBoard.state) == BLACK
            bestVal = evaluate(aBoard, alphaBetaMin, depth, cache,-200000,␣
     ↪200000)
            for move in next_moves
                Undo = domoveAdv!(aBoard, move)
                if evaluate(aBoard, alphaBetaMax, depth-1, cache,-200000,␣
     ↪200000) == bestVal
                    append!(BestMoves, [move])
                end
                undomoveAdv!(aBoard, Undo)
            end
        end
        BestMove = rand(BestMoves)
        return bestVal, BestMove
    end
```

Figure 3.94.: Implementation of Alpha-Beta Pruning with Memoization (complete)

3.12. Iterative Deepening

This notebook implements the iterative deepening algorithm.

It works by repeatedly applying alpha-beta pruning to a game tree while increasing depth limits until a certain time limit is reached or a mate is found. The iterative deepening algorithm iterates over the depth starting at a depth of one. It will search the game tree to a depth of one. Using the evaluation, it can then order the next moves after their evaluation. Adding move ordering to the alpha-beta-pruning algorithm will improve the performance of the search. Moves with a good evaluation have a high chance of being actual good moves, resulting in more paths pruned while searching the game tree. It will run until it has reached the maximum depth. When it finds a forced mate at any stage, this is also the shortest way of mating the opponent. This approach allows the algorithm to focus on the most promising branches of the game tree, while still exploring deeper levels when necessary.

[ ]:
```
using Pkg
# Pkg.add("Chess")
using Chess
using Random
# Pkg.add("DataStructures")
using DataStructures
# Pkg.add("NBInclude")
using NBInclude
using BenchmarkTools
```

Figure 3.95.: Julia package and module imports

[ ]:
```
@nbinclude("AdvancedBoard.ipynb")
```

[ ]:
```
@nbinclude("Memoization.ipynb")
```

[ ]:
```
@nbinclude("QuiescenceSearch.ipynb")
```

Figure 3.96.: Including notebook files

3.12.1. maxValue

The function `maxValue` returns the minimal centipawn evaluation of the current position for the player playing white, where both players have played the optimal moves according to the algorithm and terminating after the given depth. All possible moves

are sorted by a `PriorityQueue` using the evaluation of the position. Good moves will be prioritized, which will increase the chance of pruning paths.

Arguments:

1. `aBoard::AdvBoard` is a chess `state`
2. `depth::Int64` is the number of half moves the engine should analyze before terminating
3. `alpha::Int64` is a minimal value that has been calculated during the recursive process
4. `beta::Int64` is a maximal value that has been calculated during the recursive process
5. `flagQuiesce::Bool` is an optional flag specifies whether the quiesce-search should be used. If not, then the see-variant of the quiesce-search will be used.

Returns the value evaluation which is the best score for white if both players play the ideal moves according to the engine.

[ ]:
```
function maxValue(aBoard::AdvBoard, depth::Int64,
                  cache::Dict{UInt64, Tuple{String, Int64, Int64}},
                  alpha::Int64, beta::Int64, flagQuiesce::Bool=false)::
↪Int64
    if isterminal(aBoard.state)
        return terminal_evaluation(aBoard)
    end
    if depth == 0
        # return aBoard.score
        if flagQuiesce
            return quiesceMax(aBoard, 0, alpha, beta)
            # return evaluate(aBoard, quiesceMax, 0, cache, alpha,␣
↪beta)
        else
            return aBoard.score
        end
    end
    value = alpha
    queue = PriorityQueue{Move, Int64}()
    for move in moves(aBoard.state)
        nextHash = zobrist_hash(aBoard.state, aBoard.hash, move)
        val = value_cache(nextHash, depth-2, cache)
```

```
        if val == nothing
            val = -200000
        end
        enqueue!(queue, move, -val)
    end
    while !isempty(queue)
        move = peek(queue)[1]
        undo = domoveAdv!(aBoard, move)
        value = max(value, evaluate(aBoard, minValue,  depth - 1,␣
↪cache, value, beta, flagQuiesce))
        undomoveAdv!(aBoard, undo)
        if value >= beta
            return value
        end
        delete!(queue, move)
    end
    return value
end
```

Figure 3.97.: Implementation of the 'maxValue' function

3.12.2. minValue

The function `minValue` returns the maximal centipawn evaluation of the current position for the player playing black where both players have played the optimal moves according to the algorithm and terminating after the given depth. All possible moves are sorted by a `PriorityQueue` using the evaluation of the position. Good moves will be prioritized, which will increase the chance of pruning paths.

Arguments:

1. `aBoard::AdvBoard` is a chess `state`
2. `depth::Int64` is the number of half moves the engine should analyze before terminating
3. `alpha::Int64` is a minimal value that has been calculated during the recursive process
4. `beta::Int64` is a maximal value that has been calculated during the recursive process
5. `flagQuiesce::Bool`is an optional flag specifies whether the quiesce-search should be used. If not, then the `see`-variant of the quiesce-search will be used.

Returns the value evaluation which is the best score for black if both players play the ideal moves according to the engine.

[ ]:
```
function minValue(aBoard::AdvBoard, depth::Int64,
                  cache::Dict{UInt64, Tuple{String, Int64, Int64}},␣
↪alpha::Int64, beta::Int64, flagQuiesce::Bool=false)::Int64
    if isterminal(aBoard.state)
        return terminal_evaluation(aBoard)
    end
    if depth == 0
        # return aBoard.score
        if flagQuiesce
            return quiesceMin(aBoard,  0,  alpha, beta)
            # return evaluate(aBoard,quiesceMin , 0, cache, alpha,␣
↪beta) with Transposition
        else
            return aBoard.score
        end
    end
    value = beta
    queue = PriorityQueue{Move, Int64}()
    for move in moves(aBoard.state)
        nextHash = zobrist_hash(aBoard.state, aBoard.hash, move)
        val = value_cache(nextHash, depth-2, cache)
        if val == nothing
            val = 200000
        end
        enqueue!(queue, move, val)
    end
    while !isempty(queue)
        move = peek(queue)[1]
        Undo = domoveAdv!(aBoard, move)
        value = min(value, evaluate(aBoard, maxValue, depth - 1, cache,␣
↪alpha, value, flagQuiesce))
        undomoveAdv!(aBoard, Undo)
        if value <= alpha
            return value
        end
        delete!(queue, move)
```

```
    end
    return value
end
```

Figure 3.98.: Implementation of the 'minValue' function

The function `value_cache` is a helping function for the `minValue` and `maxValue` function.

Arguments:

1. `hash::UInt64` is a chess `state` of type `Board`
2. `depth::Int64` is the number of half moves the engine should analyze before terminating
3. `cache::Dict{UInt64, Tuple{String, Int64, Int64}}` is the cache that is searched

The function looks into the cache and returns any previously saved values for this position. This information is used to sort good moves inside the PriorityQueue.

The function returns the value for this `hash` if the `hash` is in the Cache and has a sufficient pre-calculated depth. If the cache does not have an entry for this `hash` or entry does not have sufficient depth, the function will return `nothing`.

[ ]:
```
function value_cache(hash::UInt64, depth::Int64, cache::Dict{UInt64,␣
↪Tuple{String, Int64, Int64}})
    tuple::Tuple{String, Int64, Int64} = get(cache, hash, ("", 0, 0))
    if tuple != ("", 0, 0)
        _, value, d = tuple
        if d >= depth
            return value
        end
    end
    # new move or no entry with enough depth
    return nothing
end
```

Figure 3.99.: Implementation of the 'value_cache' function

Function: pd_evaluate

This function performs an iterative deepening search on a given state of the board using the evaluation function f and caching the results using the dictionary cache.

Arguments:

1. `aBoard::AdvBoard` is a chess board in the current state
2. `f::Function` is the evaluation function that will be called
3. `depth::Int64` is the maximum depth of the search
4. `cache::Dict{UInt64, Tuple{String, Int64, Int64}}` is the dictionary to cache the results
5. `quiece:: Boolean` is Flag to use QuienceSearch
6. `increaseDepth:: Float` an number indicating the time threshold in seconds. If the search time is lower than this value and the current depth equals depth, the depth is increased by one. To deactivate this feature, you have to set this parameter to zero.
7. `showTimes:: Boolean` is Flag to print Time and SearchDepth

Returns:

1. `bestVal`: The best value found in the search.
2. `depth`: The depth at which the best value was found.

[ ]:
```
function pd_evaluate(aBoard::AdvBoard, f::Function, depth::Int64,
                     cache::Dict{UInt64, Tuple{String, Int64, Int64}},␣
↪quiece, increaseDepth, showTimes)
    bestVal = aBoard.score
    # println("Boards score ", score)
    d = 1
    alpha = -100000
    beta = 100000
    while d <= depth
        # start time
        starttime = time()
        flagQuiesce = quiece && (d == depth)
        bestVal = evaluate(aBoard, f, d, cache, alpha, beta,␣
↪flagQuiesce)
        if abs(bestVal) == 100000
            return bestVal, d
        end
        # Check if is d equals depth and the difference between␣
↪startime and currenttime increases depth
        if showTimes
```

```
            println("The best value was calculated with a depth of $d␣
↪and it took $(time() - starttime ) seconds. ")
        end
        if increaseDepth != 0 && d == depth && time() - starttime <␣
↪increaseDepth
            depth += 1
        end
        d +=1
    end
    return bestVal, depth
end
```

Figure 3.100.: Implementation of the 'pd_evaluate' function

Function: iterativeDeepening

This function performs an iterative deepening search on the given state and returns the best move found.

The algorithm calculates the best score for a player. After that, it will search for this move in the cache. This move will be in the cache with the flag of "=". If not, all moves that got pruned and could potentially carry this move will be considered and further evaluated.

Arguments:

1. `aBoard::AdvBoard` is a chess board in the current state
2. `depth::Int64` is the maximum depth to search
3. `cache::Dict{UInt64, Tuple{String, Int64, Int64}}` is the cache dictionary to store the evaluated board states. It is optional and initializes an empty Dictionary by default.
4. `quiece:: Boolean` is Flag to use QuienceSearch. It is optional and initializes to true by default.
5. `timeBoundIncreaseDepth:: Float` a time threshold to increase depth.
6. `showTimes:: Boolean` is Flag to print Time and SearchDepth. It is optional and initializes to true by default.

Returns:

1. `bestVal`: The best score found by the iterative deepening search.
2. `BestMove`: The best move found by the iterative deepening search.

[ ]:
```
function iterativeDeepening(aBoard::AdvBoard,  depth::Int64,
                            cache::Dict{UInt64, Tuple{String, Int64,␣
↪Int64}} = initCache(),
                            quiese::Bool = true, timeBoundIncreaseDepth:
↪:Float64 = 1.0, showTime::Bool = true)
   side = sidetomove(aBoard.state)
   bestVal, depth = (side == WHITE) ? pd_evaluate(aBoard, maxValue,␣
↪depth, cache, quiese, timeBoundIncreaseDepth, showTime) :
                                      pd_evaluate(aBoard, minValue,␣
↪depth, cache, quiese, timeBoundIncreaseDepth, showTime)
   next_moves = moves(aBoard.state)

   # The best move is usually contained in the cache directly. If so␣
↪it will have a sign of "=" with the value equal to bestVal
   BestMoves::Array{Move} = [move for move in next_moves if␣
↪zobrist_hash(aBoard.state, aBoard.hash, move) in keys(cache) &&
                                                                  ␣
↪cache[zobrist_hash(aBoard.state, aBoard.hash, move)][1] == "=" &&
                                                                  ␣
↪cache[zobrist_hash(aBoard.state, aBoard.hash, move)][2] == bestVal &&
                                                                  ␣
↪get(aBoard.repCounter, zobrist_hash(domove(aBoard.state, move)), 0) !
↪= 3]

   # Recalculation if not directly contained in the cache
   if length(BestMoves) == 0
       for move in next_moves
           undoinfo = domoveAdv!(aBoard, move)
           if get(aBoard.repCounter, zobrist_hash(aBoard.state), 0) ==␣
↪3
               if bestVal == 0
                   append!(BestMoves, [move])
               end
           else
               if side == WHITE
                   if evaluate(aBoard, minValue, depth-1, cache,␣
↪-100000, 100000, quiese) == bestVal
                       append!(BestMoves, [move])
```

```
                    end
                else
                    if evaluate(aBoard, maxValue, depth-1, cache,␣
 ↪-100000, 100000, quiese) == bestVal
                        append!(BestMoves, [move])
                    end
                end
            end
            undomoveAdv!(aBoard, undoinfo)
        end
    end
    print(BestMoves)
    BestMove::Move = rand(BestMoves)

    return bestVal, BestMove
end
```

Figure 3.101.: Implementation of the 'iterativeDeepening' function

3.13. Complexity

This notebook estimates the complexity of the search algorithms as described by Norvig, 2009.[17]

[3]:
```
using Chess
```

Figure 3.102.: Julia package and module imports for the Calculation of Complexity

3.13.1. Minimax

The complexity of the minimax algorithm can be estimated using the term $ O(bˆm) $, where b is the `branching factor` and m is the search depth. [17] The `branching factor` is the number of possible moves playable in a current position. This estimate can be used due to the assumption that the number of moves does not drastically change in a short period of time.

[4]:
```
# Ding Liren - Ian Nepomniachtchi 2023
b = fromfen("r1bq1rk1/ppp2ppp/2np4/8/2PPPp2/2P2N2/P1Q1BPPP/R3K2R w KQ -␣
 ↪0 11")
```

```
[4]: Board (r1bq1rk1/ppp2ppp/2np4/8/2PPPp2/2P2N2/P1Q1BPPP/R3K2R w KQ -):
 r  -  b  q  -  r  k  -
 p  p  p  -  -  p  p  p
 -  -  n  p  -  -  -  -
 -  -  -  -  -  -  -  -
 -  -  P  P  P  p  -  -
 -  -  P  -  -  N  -  -
 P  -  Q  -  B  P  P  P
 R  -  -  -  K  -  -  R
```

Figure 3.103.: FEN representation of the chess position between Ding Liren and Ian Nepomniachtchi in 2023

In this example $b^m = 42875$ and the real number of states calculated in minimax is 38055.

```
[5]: length(moves(b))^3
```

```
[5]: 42875
```

Figure 3.104.: Code snippet calculating of the number of moves in the chess position.

```
[6]: function count_moves(board, d)
         if d == 1
             return length(moves(board))
         end
         sum = 0
         for move in moves(board)
             undoinfo = domove!(board, move)
             sum += count_moves(board, d-1)
             undomove!(board, undoinfo)
         end
         sum
     end
```

```
[6]: count_moves (generic function with 1 method)
```

Figure 3.105.: Code snippet defining the function "count_moves" to recursively calculate the number of moves in a chess position

```
[7]: count_moves(b, 3)
```

```
[7]: 38055
```

Figure 3.106.: Result of the "count_moves" function using the minimax algorithm on the chess position between Ding Liren and Ian Nepomniachtchi in 2023

3.13.2. Alpha-Beta-Pruning

Using the Alpha-beta-pruning many paths are getting pruned for a constant cost of saving two additional variables α and β and managing them. The complexity of alpha-beta-pruning can be estimated using the term $O(b^{\frac{3m}{4}})$. [17] For the above example the estimated amount of boards analyzed is about 3000 which is 13 times less than simple minimax. Note: The actual value has not been calculated.

```
[11]: m = 3
      round(length(moves(b))^((3*m)/4))
```

```
[11]: 2980.0
```

Figure 3.107.: Result of the "count_moves" function using the Alpha-Beta-Pruning algorithm on the chess position between Ding Liren and Ian Nepomniachtchi in 2023

3.13.3. Iterative Deepening

The effectiveness of alpha-beta pruning is highly dependent on the order in which the states are examined. If successors are examined in a random order rather than best-first, the total number of nodes examined will be roughly $O(b^{\frac{3m}{4}})$ for moderate b. When using move ordering the complexity reduces to $O(b^{\frac{m}{2}})$. [17] This reduces the number of boards analysed by another factor of about 14.

As we saw in Chapter 3, iterative deepening on an exponential game tree adds only a constant fraction to the total search time, which can be more than made up from better move ordering. Note: the actual value has not been calculated.

```
[10]: m = 3
      round(length(moves(b))^(m/2))
```

```
[10]: 207.0
```

Figure 3.108.: Result of the "count_moves" function using the Iterative Deepening algorithm on the chess position between Ding Liren and Ian Nepomniachtchi in 2023

3.14. Quiescence Search

Quiescence Search is a search algorithm designed to avoid the horizon effect in the game of chess. It is called when the search algorithm (e.g., Minimax, alpha-beta-pruning, ...) have reached their maximum depth. Quiescence Search will then further search through the game tree until it hits a quiet position.

A position is quiet if: $no\ captures(en\ passant\ included) \vee no\ checks \vee no\ promotions$

Additionally, a position is tactical if it is not quiet. Accordingly, a tactical move is a move where a piece is captured, a check is given or a pawn is promoted.

Then and only then it will use the heuristic function to estimate the position. It is important to not stop searching when hitting the end of the depth as this last move can conclude into very good moves for the opposing side. As an example, the last move searched by the algorithm is a queen capturing a pawn. The evaluation function therefore will return a score of +100 centipawns as it just took this pawn. Notice that the search algorithm will not see that this pawn was protected, resulting in the queen being captured in the move after. This issue is called the **horizon effect**.

Quiescence Search will therefore keep calculating all tactical moves and stop when it hits a quiet position. As only the tactical moves will be evaluated, this evaluation will not take as much performance as a regular search. Additionally, pruning will result in about 50-90% of paths pruned. [18]

There are many approaches on implementing Quiescence Search. The definition of a quiet position also varies. Therefore, we define a quiet position as follows:

[ ]:
```
using Pkg
# Pkg.add("Chess")
using Chess
using NBInclude
using BenchmarkTools
```

Figure 3.109.: Julia package and module imports

Chess Programming Wiki "Quiescence Search"

[ ]:
```
@nbinclude("AdvancedBoard.ipynb")
```

[ ]:
```
@nbinclude("Memoization.ipynb")
```

Figure 3.110.: Including notebook files

3.14.1. Auxiliary functions

Function: hasCaptureMoves

This function takes in a chess game state and returns a boolean indicating whether there are any capture moves available for the opponent.

Arguments: 1. `State::Board` A chess board in the current state.

Returns whether there are any capture moves available for the opponent.

[ ]:
```
function hasCaptureMoves(State::Board)
    allPiecesSq = pieces(State, coloropp(sidetomove(State)))
    return any(isattacked(State, square, coloropp(sidetomove(State)))␣
 ↪for square in allPiecesSq)
end
```

Figure 3.111.: Implementation of the hasCaptureMoves function

Function: isTacticalMove

The function `isTacticalMove` checks whether the move done on a state is a tactical move.

A move is a tactical move if is one of the following is true: 1. The move is capturing a piece 1. The move is promoting a pawn 1. The move is an en passant capture 1. The move checks the opposing king

Arguments:

1. `State::Board` The current board
2. `move::Move` The move on the `State` that should be checked for tactical moves

Returns:

1. `true` if the move is a tactical move, otherwise false

[ ]:
```
function isTacticalMove(State::Board, move::Move)
    if pieceon(State, to(move)) != EMPTY || ispromotion(move) ||␣
 ↪epsquare(State) == to(move)
        return true
    end
    undoinfo = domove!(State, move)
    isNextMoveCheck = ischeck(State)
```

```
    undomove!(State, undoinfo)
    return isNextMoveCheck
end
```

Figure 3.112.: Implementation of the isTacticalMove function

3.14.2. Quiescence Min and Max

Due to performance issues, we had to simplify the quiescence search. The amount of capturing moves during the middlegame of chess is very high resulting in a large game tree when using the quiescence search. During development we reached positions about 20 plays deeper after starting the quiescence search. This lead to long calculation times. The simplified version of quiescence search is limited to a maximum of 3. This means that when the search-algorithm has reached its maximum depth the quiescence search will do an additional 3 steps if necessary. After that it will return the static evaluation of the board for this position.

The function `quiescenceMax` uses the quiescence algorithm to determine the best score (meaning maximizing the score) for positions where white moves. It only stops searching if the position is quiet and only searches through positions which have tactical moves.

Arguments:

1. `State::Board` The current board
2. `score::Int64` The score of the current board
3. `hash::UInt64` The hash of the current board
4. `depth::Int64` The depth of the search (initially always set to 0 and decremented by each step)
5. `alpha::Int64` The lower boundary of the score for this search
6. `beta::Int64` The upper boundary of the score for this search

Returns:

1. `score::Int64` The best score that can be achieved for the current State if both sides play optimally.

[ ]:
```
function quiesceMax(aBoard::AdvBoard, depth::Int64, alpha::Int64, beta::
↪Int64)::Int64
    # Check for Repetition as this is not covered by "evaluate"␣
↪function
```

```
    if get(aBoard.repCounter, aBoard.hash, 0) == 3
        return 0
    end
    value::Int64 = aBoard.score
    depth <= -4 && return aBoard.score
    value >= beta && return beta
    alpha = max(alpha, value)

    for move in moves(aBoard.state)
        isTacticalMove(aBoard.state, move) || continue
        undoinfo::Tuple{Int64, UInt64, UndoInfo} = domoveAdv!(aBoard,␣
↪move)
        value = quiesceMin(aBoard, depth-1, alpha, beta)
        undomoveAdv!(aBoard, undoinfo)
        value >= beta && return beta
        alpha = max(alpha, value)
    end
    return alpha
end
```

Figure 3.113.: Implementation of the quiesceMax function

The function `quiescenceMin` uses the quiescence algorithm to determine the best score (meaning minimizing the score) for positions where black moves. It only stops searching if the position is quiet and only searches through positions which have tactical moves.

Arguments:

1. `State::Board` The current board
2. `score::Int64` The score of the current board
3. `hash::UInt64` The hash of the current board
4. `depth::Int64` The depth of the search (initially always set to 0 and decremented by each step)
5. `alpha::Int64` The lower boundary of the score for this search
6. `beta::Int64` The upper boundary of the score for this search

Returns:

1. `score::Int64` The best score that can be achieved for the current State if both sides play optimally.

```
[ ]: function quiesceMin(aBoard::AdvBoard, depth::Int64, alpha::Int64, beta::
     ↪Int64)::Int64
        # Check for Repetition as this is not covered by "evaluate"␣
     ↪function
        if get(aBoard.repCounter, aBoard.hash, 0) == 3
            return 0
        end
        value = aBoard.score
        depth <= -4 && return aBoard.score
        value <= alpha && return alpha
        beta = min(value, beta)

        for move in moves(aBoard.state)
            isTacticalMove(aBoard.state, move) || continue
            undoinfo::Tuple{Int64, UInt64, UndoInfo} = domoveAdv!(aBoard,␣
     ↪move)
            value = quiesceMax(aBoard, depth-1, alpha, beta)
            undomoveAdv!(aBoard, undoinfo)
            value <= alpha && return alpha
            beta = min(value, beta)
        end
        return beta
    end
```

Figure 3.114.: Implementation of the quiesceMin function

3.14.3. Quiescence Search with see() function

The function `quiesceSee` uses the `see()` function from `Chess.jl` to determine an estimate whether any pieces may be captured in the near future. The see function sums captured pieces values in piece points. For example the see function returns that 5 points of material were lost when a rook is captured. Therefore this value is multiplied by 100 as this roughly delivers the centipawn value.

Arguments:

1. `State::Board` The current board
2. `score::Int64` The score of the current board
3. `alpha::Int64` The lower boundary of the score for this search
4. `beta::Int64` The upper boundary of the score for this search

Returns:

1. `score::Int64` The best score that can be achieved for the current State if both sides play optimally.

[ ]:
```
function quiesceSee(aBoard::AdvBoard, alpha::Int64, beta::Int64)
    if sidetomove(aBoard.state) == WHITE
        bestEstimate::Int64 = -100000
        for move in moves(aBoard.state)
            bestEstimate = max(bestEstimate, see(aBoard.state, move))
        end
    else
        bestEstimate = 100000
        for move in moves(aBoard.state)
            bestEstimate = min(bestEstimate, see(aBoard.state, move))
        end
    end

    return aBoard.score + bestEstimate * 100
end
```

Figure 3.115.: Implementation of the quiesceSee function

4. Results

In this chapter the results of these projects are summarized.

In the project we developed a notebook to play against any of the engines that were developed. It also includes saving and replaying games using Portable Game Notation (PGN) files.

During development, to show the correctness of the engine, we use chess puzzles. Chess puzzles are boards designed in such a way that only one good move exists. This can either lead to forced mate or the gain of material. Therefore, these positions must be solved by the engines as long as the depth of the engine goes far enough.

For further analyses, we choose to analyze two positions for each phase of the game and one for each side A.6 and A.7. Meaning that there will be six boards with different colors and phases to cover a wide variety of positions. In these positions, there are no forced good moves. Many moves lead to good positions, but some may be slightly better than others. The notebooks with the analyzed positions are included in the appendix.

Algorithms	depth	Pos1	Pos2	Pos3	Pos4	Pos5	Pos6
Minimax	5	72.7	49.0	116.2	22.5	16.1	4.6
Minimax + Mem	5	55.5	35.7	61.5	17.8	4.5	1.8
$\alpha - \beta$	6	87.1	80.5	53.3	23.1	8.1	2.0
$\alpha - \beta$ + Mem	6	46.2	40.4	29.7	11.2	3.1	1.0
Iterative Deepening	6	20.0	17.6	42.0	12.7	3.0	0.6
Iter. Deep. + Quies.	6	42.3	41.2	89.8	41.7	4.6	1.1

Table 4.1.: Search time of all algorithms in seconds

Looking at these results, starting at the top, we can see that the Minimax algorithm takes the longest. Alpha-beta-pruning improves the time by a lot. It is possible to compute one more depth in a reasonable amount of time (The calculations do not take more than half an hour) compared to Minimax. Iterative Deepening further reduces the time of the Alpha-beta-pruning algorithm by a factor of about 0.22 (Pos2) to 0.79 (Pos3). Additionally, Memoization reduces the time by a factor of about 0.5 compared to the algorithm running without Memoization. Quiescence Search increases the time of the

calculation. Therefore, it consumes about double the amount that Iterative Deepening running without Quiescence Search would have needed.

To test the strength of the AI, we ran a test on the 6 positions used. The notebook used for this is included in the appendix A.7. In each position, the AI got to calculate the best move. These moves were then inputted into the engine Stockfish for its evaluation. These evaluations can be seen in the table below. The value calculated is taken from the iterative deepening algorithm. Results from this algorithm are identical to the ones from minimax and alpha-beta, due to them finding the best value in the complete game tree. The exception is if there are multiple best moves, then a random one will be chosen, resulting in a different result. Memoization also does not affect the outcome of the calculation and only improves the calculation speed.

depth	Pos1	Pos2	Pos3	Pos4	Pos5	Pos6
1	0.0	1.1	0.3	0.3	-6.8	-0.2
2	0.0	1.1	0.3	0.3	0.0	-0.2
3	0.0	1.1	0.1	7.0	0.0	-0.9
4	0.0	1.1	0.9	0.3	0.5	-0.9
5	0.0	1.1	0.3	2.2	0.3	-0.9
6	0.3	1.0	0.3	0.3	0.0	-0.9
Stockfish Evaluation	0.4	0.3	-0.7	0.1	0.5	-0.5

Table 4.2.: Stockfish Evaluation Compared to Calculation Depth

Using the results from the table, there is no significant tendency to be found. Nevertheless, the table shows, that the implemented engine can reliably find a good move not losing on the spot. An exception occurred in positions 4 and 5 where tactics were not found due to low depth.

Our chess bot achieved a 100% win rate against the 1500 ELO-rated bot, both with white and black pieces. Two examples of played games are provided in the appendix, one with white pieces A.2 and the other with black A.3. The chess engine used in the evaluation was estimated to have an ELO rating between 1800 A.5 and 2300 A.4.

5. Discussion

5.1. Conclusion

After winning a few games against a chess engine rated 1500 engines, we can approve the hypothesis that the engine we developed has reached an ELO rating of above 1500. The engine is exactly implemented as described in the chapters above and therefore meet the conditions that were set at the beginning of the project.

5.2. Perspective and Alternatives

The project is overall done and can be used to play on an intermediate to advanced level. Further improvements, especially in performance and features, can improve the strength of the engine even more. One suggestion is the implementation of Multi-threading and precalculation. The engine's greatest weakness at the moment is the calculation time. Therefore, multi-threading can make calculations in the background, increasing the available time for calculations. In addition to general small tweaks, the weaknesses of the Chess Engine in the opening phase, can be addressed to improve its performance. As observed in the evaluation, our bot only achieved 68% accuracy A.5 in the opening phase as suggested by `Chess.com`. Using an opening database and endgame table base can improve the strength of the engine in the opening and endgame phase of the game.
To improve the calculation, some improvement can be done by introducing better and more algorithms. For example a better evaluation function considering the pawn positioning can improve the evaluation. Additionally, the search algorithm can be improved by using Null-move-pruning or Monte-Carlo search. These are move complex algorithms than implemented in this project and can be useful to improve not only strength but also the performance of the engine.

Bibliography

[1] Claude E. Shannon. *Programming a Computer for Playing Chess*. Tech. rep. Bell Telephone Laboratories, Inc., Murray Hill, N.J., 1950. URL: http://archive.computerhistory.org/projects/chess/related_materials/text/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon/2-0%20and%202-1.Programming_a_computer_for_playing_chess.shannon.062303002.pdf.

[2] *Simplified Evaluation Function - Chessprogramming wiki*. 2021. URL: https://www.chessprogramming.org/Simplified_Evaluation_Function (visited on 10/01/2023).

[3] *Julia vs Python - Which Should You Learn?* 2023. URL: https://www.datacamp.com/blog/julia-vs-python-which-to-learn (visited on 04/23/2023).

[4] Aayush Parashar, Aayush Kumar Jha, and Manoj Kumar. "Analyzing a Chess Engine Based on Alpha–Beta Pruning, Enhanced with Iterative Deepening". In: *Expert Clouds and Applications*. Ed. by I. Jeena Jacob, Selvanayaki Kolandapalayam Shanmugam, and Robert Bestak. Vol. 444. Lecture Notes in Networks and Systems. Singapore: Springer Nature Singapore, 2022, pp. 691–700. ISBN: 978-981-19-2499-6. DOI: 10.1007/978-981-19-2500-9/51.

[5] *Julia Micro-Benchmarks*. 2022. URL: https://julialang.org/benchmarks/ (visited on 12/06/2022).

[6] *Variables Documentation*. 2023. URL: https://docs.julialang.org/en/v1/manual/variables/ (visited on 04/23/2023).

[7] *Types · The Julia Language*. 2022. URL: https://docs.julialang.org/en/v1/manual/types/ (visited on 12/06/2022).

[8] Jeff Bezanson, Stefan Karpinski Viral Shah Alan Edelman. *Julia Packages*. Nov. 2022. URL: https://julialang.org/packages/ (visited on 12/02/2022).

[9] PyPI. *PyPI · Der Python Package Index*. Dec. 2022. URL: https://pypi.org/ (visited on 12/02/2022).

[10] *Essentials · The Julia Language*. 2022. URL: https://docs.julialang.org/en/v1/base/base/ (visited on 12/02/2022).

[11] André Schulz. "Arpad Elo zum 100.Geburtstag". In: *ChessBase* (2003). URL: https://de.chessbase.com/post/arpad-elo-zum-100-geburtstag (visited on 04/16/2023).

[12] Chess.com. *Elo - Schachbegriffe*. 2023. URL: https://www.chess.com/de/terms/elo (visited on 04/16/2023).

[13] Sam Copeland. "The 7 Most Amazing Chess Records". In: *Chess.com* (2021). URL: https://www.chess.com/article/view/the-7-most-amazing-chess-records (visited on 04/16/2023).

[14] Prof. Dr. Karl Stroetmann. *An Introduction to Artificial Intelligence*. https://github.com/karlstroetmann/Artificial-Intelligence/blob/master/Lecture-Notes/artificial-intelligence.pdf. Accessed: 2023-04-17. 2023.

[15] Donald E. Knuth and Ronald W. Moore. *An analysis of alpha-beta pruning*. Vol. 6. 4. 1975, pp. 293–326. ISBN: 1237242320. DOI: 10.1016/0004-3702(75)90019-3.

[16] T. A. Marsland and M. Campbell. "Parallel Search of Strongly Ordered Game Trees". In: *ACM Computing Surveys (CSUR)* 14.4 (1982), pp. 533–551. ISSN: 15577341. DOI: 10.1145/356893.356895.

[17] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. 4th ed. Pearson Education Limited, 2021. ISBN: 978-1-292-40113-3.

[18] Chess Programming wiki. *Quiescence Search*. 2022. URL: https://www.chessprogramming.org/Quiescence_Search.

[19] *Weekly Classical rating distribution • lichess.org*. 4/16/2023. URL: https://lichess.org/stat/rating/distribution/classical (visited on 04/16/2023).

[20] *Weekly Rapid rating distribution • lichess.org*. 4/16/2023. URL: https://lichess.org/stat/rating/distribution/rapid (visited on 04/16/2023).

A. Appendix

A.1. Weekly Chess Rating distribution

ELO Distribution in the classical format of chess:

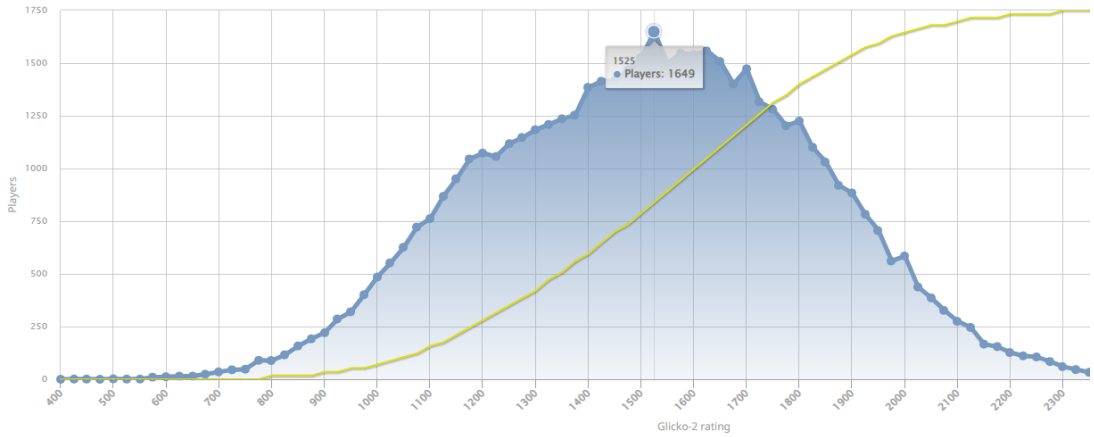

Figure A.1.: Weekly Classical Chess Rating distribution – In this figure, the y-axis represents the number of players, while the x-axis represents the ELO ratings. The majority of players cluster around the point 1649, with a rating of 1525. The yellow line indicates the cumulative distribution, giving us a broader perspective on the overall player distribution.[19]

ELO Distribution in the rapid format of chess:

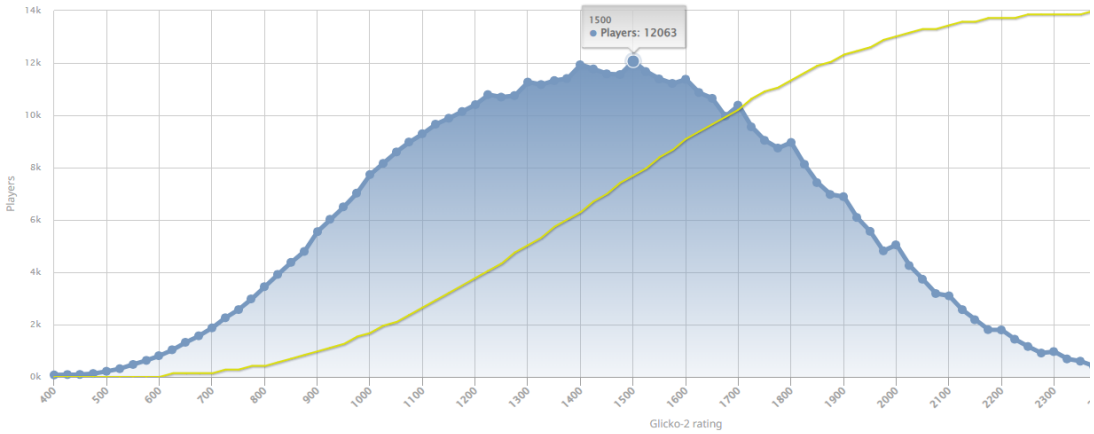

Figure A.2.: Weekly Rapid Chess Rating distribution - In this figure, the y-axis represents the number of players, while the x-axis represents the ELO ratings. The majority of players cluster around the point 12063, with a rating of 1500. The yellow line indicates the cumulative distribution, giving us a broader perspective on the overall player distribution [20]

A.2. Play as White against ELO 1500 Bot

```
[Event "vs Computer"]
[Site "Chess.com"]
[Date "2023.04.16"]
[Round "-"]
[White "DHBW Bot"]
[Black "Antonio-BOT"]
[Result "1-0"]
[CurrentPosition "4Q1k1/pp3npp/2p5/8/4P3/5P2/PPP3PP/4K2R b K - 0 22"]
[Timezone "UTC"]
[UTCDate "2023.04.16"]
[UTCTime "14:04:10"]
[WhiteElo "??"]
[BlackElo "1500"]
[TimeControl "1/0"]
[Termination "DHBW Bot won by checkmate"]
[StartTime "14:04:10"]
[EndDate "2023.04.16"]
[EndTime "15:01:13"]
[Link "https://www.chess.com/game/computer/59022697"]

1. e4 e5 2. Nf3 Nc6 3. Nc3 Bc5 4. Nxe5 Nxe5 5. d4 Bd6 6. dxe5 Bxe5
7. Nd5 c6 8.Bf4 d6 9. Bxe5 dxe5 10. Nc3 Qb6 11. Qd6 Qd4 12. Qc7
Nh6 13. Rd1 O-O 14.Rxd4 exd4 15. Ne2 Re8 16. f3 Be6 17. Nxd4 Rab8
18. Nxe6 fxe6 19. Bc4 Nf7 20.Bxe6 Rxe6 21. Qxb8+ Re8 22. Qxe8# 1-0
```

Figure A.3.: Play as White against ELO 1500 Bot

A.3. Play as Black against ELO 1500 Bot

```
[Event "vs Computer"]
[Site "Chess.com"]
[Date "2023.04.16"]
[Round "-"]
[White "Wendy-BOT"]
[Black "DHBW-BOT"]
[Result "0-1"]
[CurrentPosition "5r1k/p5pp/1n6/8/1P3p2/5P2/3r3P/5q1K w - - 0 33"]
[Timezone "UTC"]
[UTCDate "2023.04.16"]
[UTCTime "20:16:24"]
[WhiteElo "1500"]
[BlackElo "??"]
[TimeControl "1/0"]
[Termination "DHBW-BOT won by checkmate"]
[StartTime "20:16:24"]
[EndDate "2023.04.16"]
[EndTime "20:56:54"]
[Link "https://www.chess.com/game/computer/59102053"]

1. d4 d5 2. c4 dxc4 3. e4 b5 4. a4 Nc6 5. Nf3 e5 6. d5 Bb4+
7. Nc3 Nce7 8. axb5 Bg4 9. Bd2 Nf6 10. Bxc4 Bxc3
11. Bxc3 Nxe4 12. Bxe5 f6 13.Bxc7 Qxc7 14. O-O Qxc4 15. d6 Rd8
16. Ra6 Qxb5 17. Qd4 Bxf3 18. gxf3 Qxa6 19. d7+ Rxd7 20. Qxe4 f5
21. Qb4 O-O 22. Re1 Nd5 23. Qd2 Qg6+ 24. Kh1 f4 25. Rg1 Qf5 26. Qd4 Qc2
27. Re1 Nb6 28. Qe4 Qxf2 29. b4 Rd2 30. Qe6+ Kh8 31. Qh3 Qxe1+
32. Qf1 Qxf1# 0-1
```

Figure A.4.: Play as Black against ELO 1500 Bot

A.4. Suggestion Rating Chess.com about Game with White

Figure A.5.: Suggestion Rating Chess.com about Game with White

A.5. Suggestion Rating Chess.com about Game with Black

Figure A.6.: Suggestion Rating Chess.com about Game with Black

A.6. Engine Comparing

[13]:
```
using Pkg
# Pkg.add("Chess")
using Chess
using Random
# Pkg.add("NBInclude")
using NBInclude
```

Figure A.7.: Julia package and module imports1

[14]:
```
@nbinclude("AdvancedBoard.ipynb")
```

[14]:
```
terminal_evaluation (generic function with 2 methods)
```

A.6.1. Engine

Importing Engines

[15]:
```
@nbinclude("RandomChess.ipynb")
@nbinclude("Minimax.ipynb")
@nbinclude("AlphaBetaPruning.ipynb")
@nbinclude("IterativeDeepening.ipynb")
```

[15]:
```
iterativeDeepening (generic function with 5 methods)
```

Figure A.8.: Including notebook1 files to compare the engine

[18]:
```
function testTime(board)
    display(board)
    boardScore = evaluate_position(board)
    aB = AdvBoard(board)
    println("TIME MEASUREMENT")
    println("--------Random Engine-------")
    @time generateRandomMove(board)
    println("--------Minimax no Mem-------")
    for i in 1:5
        print("depth ", i , ": ")
        @time minimax_noMem(board, boardScore, i)
    end
    println("--------Minimax -------")
```

```
    for i in 1:5
        print("depth ", i , ": ")
        @time minimax(aB, i)
    end

    println("--------Alpha-Beta Pruning no Mem-------")
    for i in 1:6
        print("depth ", i , ": ")
        @time alphaBetaPruning_noMem(aB, i)
    end

    println("--------Alpha-Beta Pruning-------")
    for i in 1:6
        print("depth ", i , ": ")
        @time alphaBetaPruning(aB, i)
    end
    println("--------Iterative Deepening-------")
    for i in 1:6
        cache = initCache()
        print("depth ", i , ": ")
        @time iterativeDeepening(aB, i, cache, false,  0.0, false)
    end

    println("--------Iterative Deepening with Quiescence-------")

    for i in 1: 6
        cache = initCache()
        print("depth ", i , ": ")
        @time iterativeDeepening(aB, i, cache, true,  0.0, false)
    end
end
```

Figure A.9.: Implement a function textTime

```
[18]: testTime (generic function with 1 method)
```

A.6.2. Analyse Time

Opening White: Berlin Opening

[19]:
```
b1 = fromfen("r1bqkb1r/pppp1ppp/2n2n2/1B2p3/4P3/5N2/PPPP1PPP/RNBQK2R w␣
 ↪KQkq - 0 1")
testTime(b1)
```

Figure A.10.: Result of comparing

```
Board (r1bqkb1r/pppp1ppp/2n2n2/1B2p3/4P3/5N2/PPPP1PPP/RNBQK2R w KQkq -):
 r  -  b  q  k  b  -  r
 p  p  p  p  -  p  p  p
 -  -  n  -  -  n  -  -
 -  B  -  -  p  -  -  -
 -  -  -  -  P  -  -  -
 -  -  -  -  -  N  -  -
 P  P  P  P  -  P  P  P
 R  N  B  Q  K  -  -  R

TIME MEASUREMENT
--------Random Engine-------
  0.000012 seconds (2 allocations: 1.797 KiB)
--------Minimax no Mem-------
depth 1:    0.000548 seconds (1.27 k allocations: 75.328 KiB)
depth 2:    0.001714 seconds (16.13 k allocations: 1.057 MiB)
depth 3:    0.049412 seconds (497.69 k allocations: 32.469 MiB)
depth 4:    2.326756 seconds (14.56 M allocations: 958.160 MiB, 6.21% gc␣
 ↪time)
depth 5:   72.696503 seconds (474.55 M allocations: 30.189 GiB, 11.02% gc␣
 ↪time)
--------Minimax -------
depth 1:    0.000216 seconds (872 allocations: 78.172 KiB)
depth 2:    0.003114 seconds (17.46 k allocations: 1.650 MiB)
depth 3:    0.096103 seconds (460.74 k allocations: 39.334 MiB)
depth 4:    2.074168 seconds (8.53 M allocations: 683.118 MiB, 7.75% gc␣
 ↪time)
depth 5:   55.528636 seconds (159.57 M allocations: 11.712 GiB, 15.70% gc␣
 ↪time)
--------Alpha-Beta Pruning no Mem-------
depth 1:    0.000168 seconds (665 allocations: 47.797 KiB)
```

```
depth 2:   0.003263 seconds (14.28 k allocations: 1.042 MiB)
depth 3:   0.069759 seconds (279.09 k allocations: 20.418 MiB)
depth 4:   1.333764 seconds (2.78 M allocations: 211.468 MiB, 49.63% gc␣
 ↪time)
depth 5:   4.773487 seconds (20.91 M allocations: 1.555 GiB, 4.41% gc␣
 ↪time)
depth 6:  87.052948 seconds (342.89 M allocations: 26.744 GiB, 5.59% gc␣
 ↪time)
--------Alpha-Beta Pruning-------
depth 1:   0.000175 seconds (705 allocations: 51.375 KiB)
depth 2:   0.006326 seconds (13.73 k allocations: 1.127 MiB)
depth 3:   0.054002 seconds (197.78 k allocations: 17.171 MiB)
depth 4:   0.474614 seconds (1.76 M allocations: 142.366 MiB, 7.70% gc␣
 ↪time)
depth 5:   3.552408 seconds (11.81 M allocations: 1.002 GiB, 5.94% gc␣
 ↪time)
depth 6:  46.215837 seconds (129.77 M allocations: 10.748 GiB, 8.05% gc␣
 ↪time)
--------Iterative Deepening-------
depth 1: 295
  0.031345 seconds (10.90 k allocations: 597.686 KiB, 98.28% compilation␣
 ↪time)
depth 2: -50
  0.001592 seconds (4.49 k allocations: 496.555 KiB)
depth 3: 150
  0.036325 seconds (126.09 k allocations: 11.307 MiB)
depth 4: -20
  0.080812 seconds (260.19 k allocations: 27.922 MiB)
depth 5: 130
  3.191031 seconds (8.40 M allocations: 807.412 MiB, 4.62% gc time)
depth 6: -80
 20.016089 seconds (44.70 M allocations: 4.356 GiB, 6.13% gc time)
--------Iterative Deepening with Quiescence-------
depth 1: 40
  0.001339 seconds (1.10 k allocations: 125.461 KiB)
depth 2: -10
  0.006990 seconds (17.29 k allocations: 2.182 MiB)
depth 3: 35
```

```
  0.054489 seconds (141.79 k allocations: 17.286 MiB)
depth 4: -5
  0.338703 seconds (923.01 k allocations: 110.078 MiB)
depth 5: 30
  5.542486 seconds (11.26 M allocations: 1.275 GiB, 13.97% gc time)
depth 6: 0
 42.290137 seconds (100.33 M allocations: 11.377 GiB, 6.82% gc time)
```

Opening Black: King's Indian Defence

[20]:
```
b2 = fromfen("rnbqkb1r/pppppp1p/5np1/8/2PP4/2N5/PP2PPPP/R1BQKBNR b KQkq␣
↪- 1 3")
testTime(b2)
```

Figure A.11.: Results of Black King's Indian opening

```
Board (rnbqkb1r/pppppp1p/5np1/8/2PP4/2N5/PP2PPPP/R1BQKBNR b KQkq -):
 r  n  b  q  k  b  -  r
 p  p  p  p  p  p  -  p
 -  -  -  -  -  n  p  -
 -  -  -  -  -  -  -  -
 -  -  P  P  -  -  -  -
 -  -  N  -  -  -  -  -
 P  P  -  -  P  P  P  P
 R  -  B  Q  K  B  N  R

TIME MEASUREMENT
--------Random Engine-------
  0.000014 seconds (2 allocations: 1.797 KiB)
--------Minimax no Mem-------
depth 1:   0.002380 seconds (907 allocations: 54.531 KiB)
depth 2:   0.013981 seconds (13.19 k allocations: 875.375 KiB)
depth 3:   0.066947 seconds (322.84 k allocations: 21.638 MiB)
depth 4:   1.821286 seconds (10.83 M allocations: 705.026 MiB, 8.73% gc␣
↪time)
depth 5:  48.978339 seconds (293.17 M allocations: 19.036 GiB, 11.48% gc␣
↪time)
--------Minimax -------
depth 1:   0.000156 seconds (635 allocations: 57.922 KiB)
depth 2:   0.002464 seconds (14.68 k allocations: 1.393 MiB)
```

```
depth 3:   0.062807 seconds (325.06 k allocations: 27.794 MiB)
depth 4:   1.804331 seconds (6.65 M allocations: 527.208 MiB, 11.20% gc␣
 ↪time)
depth 5:  35.725778 seconds (105.73 M allocations: 7.702 GiB, 12.06% gc␣
 ↪time)
--------Alpha-Beta Pruning no Mem-------
depth 1:   0.000113 seconds (469 allocations: 34.734 KiB)
depth 2:   0.002521 seconds (10.68 k allocations: 796.688 KiB)
depth 3:   0.615078 seconds (128.92 k allocations: 10.315 MiB, 94.89% gc␣
 ↪time)
depth 4:   0.662823 seconds (2.59 M allocations: 192.104 MiB, 11.27% gc␣
 ↪time)
depth 5:   6.327221 seconds (28.70 M allocations: 2.227 GiB, 4.52% gc␣
 ↪time)
depth 6:  80.521438 seconds (333.77 M allocations: 24.325 GiB, 4.66% gc␣
 ↪time)
--------Alpha-Beta Pruning-------
depth 1:   0.000120 seconds (500 allocations: 38.172 KiB)
depth 2:   0.002186 seconds (10.44 k allocations: 913.453 KiB)
depth 3:   0.022103 seconds (88.96 k allocations: 7.760 MiB)
depth 4:   0.372140 seconds (1.43 M allocations: 116.190 MiB, 3.96% gc␣
 ↪time)
depth 5:   3.916451 seconds (13.57 M allocations: 1.171 GiB, 5.71% gc␣
 ↪time)
depth 6:  40.377559 seconds (111.49 M allocations: 9.048 GiB, 7.38% gc␣
 ↪time)
--------Iterative Deepening-------
depth 1: -5
  0.000353 seconds (335 allocations: 31.289 KiB)
depth 2: 45
  0.001244 seconds (2.85 k allocations: 316.633 KiB)
depth 3: -85
  0.012954 seconds (33.44 k allocations: 3.252 MiB)
depth 4: 95
  0.672057 seconds (389.25 k allocations: 38.614 MiB, 69.31% gc time)
depth 5: -65
  1.412445 seconds (2.88 M allocations: 278.089 MiB)
depth 6: 160
```

```
 17.631898 seconds (41.48 M allocations: 4.049 GiB, 7.98% gc time)
--------Iterative Deepening with Quiescence-------
depth 1: -5
  0.000444 seconds (349 allocations: 37.164 KiB)
depth 2: 25
  0.002475 seconds (4.53 k allocations: 647.602 KiB)
depth 3: 15
  0.031143 seconds (76.47 k allocations: 9.030 MiB)
depth 4: 45
  0.294207 seconds (704.77 k allocations: 86.767 MiB, 11.53% gc time)
depth 5: 15
  2.332126 seconds (6.04 M allocations: 716.815 MiB, 6.19% gc time)
depth 6: 55
 41.191397 seconds (98.76 M allocations: 11.712 GiB, 7.14% gc time)
```

Middlegame White: Ding Liren - Ian Nepomniachtchi, 2023

[21]:
```
b3 = fromfen("4r1k1/p1p1r1pp/1p2Pp1q/2pP1n1b/2P2p2/2Q2B1P/P2NRPP1/4R1K1␣
 ↪w - - 3 26")
testTime(b3)
```

Figure A.12.: Results of Middlegame Ding Liren vs. Ian Nepommiachtchi, 2023

```
Board (4r1k1/p1p1r1pp/1p2Pp1q/2pP1n1b/2P2p2/2Q2B1P/P2NRPP1/4R1K1 w - -):
 -  -  -  -  r  -  k  -
 p  -  p  -  r  -  p  p
 -  p  -  -  P  p  -  q
 -  -  p  P  -  n  -  b
 -  -  P  -  -  p  -  -
 -  -  Q  -  -  B  -  P
 P  -  -  N  R  P  P  -
 -  -  -  -  R  -  K  -

TIME MEASUREMENT
--------Random Engine-------
  0.000012 seconds (2 allocations: 1.797 KiB)
--------Minimax no Mem-------
depth 1:   0.000894 seconds (1.47 k allocations: 86.344 KiB)
depth 2:   0.003642 seconds (18.49 k allocations: 1.211 MiB)
depth 3:   0.102836 seconds (638.50 k allocations: 41.170 MiB)
```

```
depth 4:   2.930403 seconds (19.09 M allocations: 1.224 GiB, 12.82% gc␣
 ↪time)
depth 5: 116.217525 seconds (689.84 M allocations: 43.402 GiB, 10.45% gc␣
 ↪time)
--------Minimax -------
depth 1:   0.000941 seconds (1.04 k allocations: 90.172 KiB)
depth 2:   0.010570 seconds (19.57 k allocations: 1.843 MiB)
depth 3:   0.263006 seconds (577.55 k allocations: 48.694 MiB, 39.56% gc␣
 ↪time)
depth 4:   2.790992 seconds (11.36 M allocations: 859.141 MiB, 8.71% gc␣
 ↪time)
depth 5:  61.519359 seconds (205.68 M allocations: 14.741 GiB, 18.73% gc␣
 ↪time)
--------Alpha-Beta Pruning no Mem-------
depth 1:   0.000194 seconds (765 allocations: 54.516 KiB)
depth 2:   0.003924 seconds (16.18 k allocations: 1.181 MiB)
depth 3:   0.046269 seconds (209.07 k allocations: 16.205 MiB)
depth 4:   0.604036 seconds (2.75 M allocations: 208.181 MiB)
depth 5:   6.844802 seconds (26.03 M allocations: 1.981 GiB, 14.95% gc␣
 ↪time)
depth 6:  53.332331 seconds (238.12 M allocations: 18.202 GiB, 6.45% gc␣
 ↪time)
--------Alpha-Beta Pruning-------
depth 1:   0.000173 seconds (810 allocations: 58.172 KiB)
depth 2:   0.003373 seconds (16.37 k allocations: 1.294 MiB)
depth 3:   0.030998 seconds (134.41 k allocations: 11.060 MiB)
depth 4:   0.379823 seconds (1.77 M allocations: 139.463 MiB, 8.59% gc␣
 ↪time)
depth 5:   2.800395 seconds (11.74 M allocations: 958.058 MiB, 5.77% gc␣
 ↪time)
depth 6:  29.732603 seconds (101.28 M allocations: 7.786 GiB, 7.58% gc␣
 ↪time)
--------Iterative Deepening-------
depth 1: 285
  0.000345 seconds (525 allocations: 45.117 KiB)
depth 2: -40
  0.002135 seconds (4.94 k allocations: 547.531 KiB)
depth 3: 155
```

```
  0.072921 seconds (196.80 k allocations: 17.029 MiB)
depth 4: -165
  0.344495 seconds (885.85 k allocations: 90.469 MiB, 20.28% gc time)
depth 5: 80
  2.912986 seconds (9.78 M allocations: 965.132 MiB, 6.54% gc time)
depth 6: -170
 41.994410 seconds (129.29 M allocations: 12.711 GiB, 8.02% gc time)
--------Iterative Deepening with Quiescence-------
depth 1: -20
  0.000959 seconds (2.69 k allocations: 310.102 KiB)
depth 2: -20
  0.009658 seconds (30.13 k allocations: 3.600 MiB)
depth 3: -25
  0.067559 seconds (272.14 k allocations: 30.588 MiB)
depth 4: -30
  0.564261 seconds (2.13 M allocations: 239.266 MiB)
depth 5: -30
  5.006717 seconds (15.16 M allocations: 1.668 GiB, 20.41% gc time)
depth 6: -35
 89.847517 seconds (301.03 M allocations: 33.376 GiB, 8.47% gc time)
```

Middlegame Black: Fabiano Caruana - Magnus Carlsen, 2018

[22]:
```
b4 = fromfen("2bqk1nr/1p1p1pbp/6p1/1pp1p3/1P2P3/5N2/2PP1PPP/BN1QR1K1 b␣
↪k - 0 10")
testTime(b4)
```

Figure A.13.: Results of Middlegame Black: Fabiano Caruana - Magnus Carlsen, 2018

```
Board (2bqk1nr/1p1p1pbp/6p1/1pp1p3/1P2P3/5N2/2PP1PPP/BN1QR1K1 b k -):
 -  -  b  q  k  -  n  r
 -  p  -  p  -  p  b  p
 -  -  -  -  -  -  p  -
 -  p  p  -  p  -  -  -
 -  P  -  -  P  -  -  -
 -  -  -  -  -  N  -  -
 -  -  P  P  -  P  P  P
 B  N  -  Q  R  -  K  -
```

```
TIME MEASUREMENT
--------Random Engine-------
  0.000023 seconds (2 allocations: 1.797 KiB)
--------Minimax no Mem-------
depth 1:    0.001408 seconds (993 allocations: 59.422 KiB)
depth 2:    0.002733 seconds (12.11 k allocations: 816.297 KiB)
depth 3:    0.048236 seconds (292.75 k allocations: 19.515 MiB)
depth 4:    1.319973 seconds (8.23 M allocations: 544.033 MiB, 15.33% gc␣
 ↪time)
depth 5:   22.557116 seconds (219.79 M allocations: 14.228 GiB, 9.53% gc␣
 ↪time)
--------Minimax -------
depth 1:    0.000562 seconds (695 allocations: 62.781 KiB)
depth 2:    0.003179 seconds (13.78 k allocations: 1.232 MiB)
depth 3:    0.042599 seconds (272.38 k allocations: 22.804 MiB)
depth 4:    0.839788 seconds (4.60 M allocations: 375.718 MiB, 13.28% gc␣
 ↪time)
depth 5:   17.827212 seconds (73.97 M allocations: 5.449 GiB, 18.98% gc␣
 ↪time)
--------Alpha-Beta Pruning no Mem-------
depth 1:    0.000111 seconds (517 allocations: 37.906 KiB)
depth 2:    0.002185 seconds (10.72 k allocations: 803.289 KiB)
depth 3:    0.020822 seconds (99.44 k allocations: 8.021 MiB)
depth 4:    0.275518 seconds (1.27 M allocations: 96.808 MiB)
depth 5:    4.217109 seconds (20.19 M allocations: 1.547 GiB, 12.86% gc␣
 ↪time)
depth 6:   23.091845 seconds (129.04 M allocations: 9.901 GiB, 5.23% gc␣
 ↪time)
--------Alpha-Beta Pruning-------
depth 1:    0.000128 seconds (550 allocations: 41.375 KiB)
depth 2:    0.002173 seconds (10.04 k allocations: 892.273 KiB)
depth 3:    0.016545 seconds (71.42 k allocations: 6.424 MiB)
depth 4:    0.154464 seconds (844.21 k allocations: 72.981 MiB, 7.84% gc␣
 ↪time)
depth 5:    1.513764 seconds (7.45 M allocations: 639.327 MiB, 6.13% gc␣
 ↪time)
depth 6:   11.240097 seconds (50.05 M allocations: 3.892 GiB, 9.92% gc␣
 ↪time)
```

```
--------Iterative Deepening-------
depth 1: -170
  0.000441 seconds (366 allocations: 33.531 KiB)
depth 2: -20
  0.001486 seconds (4.12 k allocations: 439.180 KiB)
depth 3: -265
  0.009000 seconds (30.41 k allocations: 2.842 MiB)
depth 4: -15
  0.079465 seconds (271.23 k allocations: 27.912 MiB)
depth 5: -155
  0.590377 seconds (2.26 M allocations: 215.591 MiB, 6.46% gc time)
depth 6: 40
 12.656808 seconds (40.50 M allocations: 4.056 GiB, 6.64% gc time)
--------Iterative Deepening with Quiescence-------
depth 1: -85
  0.002264 seconds (3.21 k allocations: 421.945 KiB)
depth 2: -20
  0.014515 seconds (27.35 k allocations: 3.367 MiB)
depth 3: -70
  0.041690 seconds (96.92 k allocations: 11.862 MiB)
depth 4: -30
  0.386206 seconds (1.26 M allocations: 151.813 MiB)
depth 5: -65
  1.972303 seconds (5.49 M allocations: 638.235 MiB, 18.69% gc time)
depth 6: -35
 41.718884 seconds (140.94 M allocations: 16.114 GiB, 8.31% gc time)
```

Endgame White: Magnus Carlsen - Ian Nepomniachtchi, 2021

[23]:
```
b5 = fromfen("8/R7/6k1/3q3p/5P2/4P1P1/4NK2/8 w - - 1 83")
testTime(b5)
```

Figure A.14.: Results of Endgame White: Magnus Carlsen - Ian Nepomniachtchi, 2021

```
Board (8/R7/6k1/3q3p/5P2/4P1P1/4NK2/8 w - -):
 -  -  -  -  -  -  -  -
 R  -  -  -  -  -  -  -
 -  -  -  -  -  -  k  -
 -  -  -  q  -  -  -  p
 -  -  -  -  -  P  -  -
```

```
 -  -  -  -  P  -  P  -
 -  -  -  -  N  K  -  -
 -  -  -  -  -  -  -  -

TIME MEASUREMENT
--------Random Engine-------
  0.000023 seconds (2 allocations: 1.797 KiB)
--------Minimax no Mem-------
depth 1:    0.000730 seconds (946 allocations: 56.734 KiB)
depth 2:    0.002554 seconds (11.59 k allocations: 781.391 KiB)
depth 3:    0.056835 seconds (243.43 k allocations: 16.606 MiB)
depth 4:    0.836546 seconds (5.65 M allocations: 382.534 MiB, 9.38% gc␣
 ↪time)
depth 5:   16.098696 seconds (119.55 M allocations: 7.988 GiB, 11.00% gc␣
 ↪time)
--------Minimax -------
depth 1:    0.000137 seconds (642 allocations: 59.516 KiB)
depth 2:    0.001894 seconds (12.98 k allocations: 1.202 MiB)
depth 3:    0.037043 seconds (225.15 k allocations: 19.279 MiB)
depth 4:    0.535150 seconds (2.69 M allocations: 214.239 MiB, 13.54% gc␣
 ↪time)
depth 5:    4.528674 seconds (20.95 M allocations: 1.564 GiB, 9.44% gc␣
 ↪time)
--------Alpha-Beta Pruning no Mem-------
depth 1:    0.000157 seconds (489 allocations: 36.078 KiB)
depth 2:    0.002362 seconds (10.22 k allocations: 769.266 KiB)
depth 3:    0.023431 seconds (86.01 k allocations: 7.152 MiB)
depth 4:    0.152980 seconds (677.62 k allocations: 55.009 MiB)
depth 5:    1.132724 seconds (4.76 M allocations: 424.414 MiB, 21.22% gc␣
 ↪time)
depth 6:    8.059857 seconds (39.47 M allocations: 3.182 GiB, 8.81% gc␣
 ↪time)
--------Alpha-Beta Pruning-------
depth 1:    0.000129 seconds (521 allocations: 39.531 KiB)
depth 2:    0.001974 seconds (10.34 k allocations: 783.164 KiB)
depth 3:    0.015759 seconds (64.33 k allocations: 5.953 MiB)
depth 4:    0.096362 seconds (458.33 k allocations: 38.547 MiB)
depth 5:    0.493821 seconds (2.07 M allocations: 185.978 MiB, 16.76% gc␣
 ↪time)
```

```
depth 6:   3.084263 seconds (14.03 M allocations: 1.164 GiB, 7.98% gc␣
 ↪time)
--------Iterative Deepening-------
depth 1: 150
  0.000258 seconds (346 allocations: 32.055 KiB)
depth 2: 95
  0.001248 seconds (3.47 k allocations: 386.930 KiB)
depth 3: 155
  0.005625 seconds (19.70 k allocations: 2.016 MiB)
depth 4: 105
  0.083798 seconds (308.62 k allocations: 33.851 MiB)
depth 5: 165
  0.236009 seconds (897.45 k allocations: 92.987 MiB)
depth 6: 95
  2.966159 seconds (9.94 M allocations: 1.039 GiB, 8.88% gc time)
--------Iterative Deepening with Quiescence-------
depth 1: 145
  0.000275 seconds (705 allocations: 84.305 KiB)
depth 2: 135
  0.004186 seconds (8.51 k allocations: 1.091 MiB)
depth 3: 125
  0.009401 seconds (40.38 k allocations: 4.865 MiB)
depth 4: 125
  0.068156 seconds (310.33 k allocations: 37.791 MiB)
depth 5: 135
  0.357750 seconds (1.37 M allocations: 165.845 MiB)
depth 6: 145
  4.587911 seconds (14.79 M allocations: 1.838 GiB, 9.42% gc time)
```

Endgame black: Mikhail Botvinnik - Robert James Fischer, 1962

[24]:
```
b6 = fromfen("8/p5R1/1p4p1/4k3/r6P/6K1/5P2/8 b - - 4 50")
testTime(b6)
```

Figure A.15.: Results of Endgame black: Mikhail Botvinnik - Robert James Fischer, 1962

```
Board (8/p5R1/1p4p1/4k3/r6P/6K1/5P2/8 b - -):
 -  -  -  -  -  -  -  -
 p  -  -  -  -  -  R  -
 -  p  -  -  -  -  p  -
```

```
 -  -  -  -  k  -  -  -
 r  -  -  -  -  -  -  P
 -  -  -  -  -  -  K  -
 -  -  -  -  -  P  -  -
 -  -  -  -  -  -  -  -

TIME MEASUREMENT
--------Random Engine-------
  0.000007 seconds (2 allocations: 1.797 KiB)
--------Minimax no Mem-------
depth 1:   0.000397 seconds (915 allocations: 55.016 KiB)
depth 2:   0.000906 seconds (6.92 k allocations: 495.625 KiB)
depth 3:   0.019513 seconds (115.50 k allocations: 8.043 MiB)
depth 4:   0.260482 seconds (1.94 M allocations: 137.087 MiB)
depth 5:   4.560528 seconds (34.07 M allocations: 2.332 GiB, 10.41% gc␣
 ↪time)
--------Minimax -------
depth 1:   0.000136 seconds (657 allocations: 58.844 KiB)
depth 2:   0.001080 seconds (6.70 k allocations: 675.945 KiB)
depth 3:   0.018269 seconds (106.25 k allocations: 9.275 MiB)
depth 4:   0.244298 seconds (969.38 k allocations: 83.440 MiB, 39.14% gc␣
 ↪time)
depth 5:   1.783893 seconds (9.20 M allocations: 727.305 MiB, 6.25% gc␣
 ↪time)
--------Alpha-Beta Pruning no Mem-------
depth 1:   0.000095 seconds (477 allocations: 35.219 KiB)
depth 2:   0.000879 seconds (5.05 k allocations: 417.789 KiB)
depth 3:   0.010971 seconds (56.45 k allocations: 4.612 MiB)
depth 4:   0.043381 seconds (241.54 k allocations: 21.087 MiB)
depth 5:   0.500465 seconds (2.42 M allocations: 212.167 MiB, 19.05% gc␣
 ↪time)
depth 6:   2.048069 seconds (10.21 M allocations: 911.737 MiB, 9.07% gc␣
 ↪time)
--------Alpha-Beta Pruning-------
depth 1:   0.000123 seconds (508 allocations: 38.656 KiB)
depth 2:   0.001250 seconds (5.25 k allocations: 451.500 KiB)
depth 3:   0.007707 seconds (36.69 k allocations: 3.183 MiB)
depth 4:   0.032439 seconds (160.58 k allocations: 14.033 MiB)
depth 5:   0.239041 seconds (1.16 M allocations: 111.176 MiB)
```

```
depth 6:   0.968330 seconds (4.08 M allocations: 373.944 MiB, 13.59% gc␣
 ↪time)
--------Iterative Deepening-------
depth 1: -220
  0.000314 seconds (338 allocations: 31.438 KiB)
depth 2: -120
  0.001198 seconds (3.77 k allocations: 396.664 KiB)
depth 3: -200
  0.003960 seconds (14.79 k allocations: 1.577 MiB)
depth 4: -60
  0.022767 seconds (90.05 k allocations: 10.183 MiB)
depth 5: -180
  0.159083 seconds (622.84 k allocations: 67.308 MiB)
depth 6: -65
  0.639822 seconds (2.17 M allocations: 239.613 MiB, 13.98% gc time)
--------Iterative Deepening with Quiescence-------
depth 1: -120
  0.000676 seconds (2.29 k allocations: 298.484 KiB)
depth 2: -120
  0.002859 seconds (8.86 k allocations: 1.148 MiB)
depth 3: -130
  0.005940 seconds (18.21 k allocations: 2.142 MiB)
depth 4: -110
  0.030342 seconds (137.88 k allocations: 18.350 MiB)
depth 5: -120
  0.206876 seconds (812.79 k allocations: 100.478 MiB)
depth 6: -120
  1.082518 seconds (3.72 M allocations: 469.485 MiB, 8.69% gc time)
```

A.7. Strength comparison

[11]:
```
using Pkg
# Pkg.add("Chess")
using Chess
using Random
# Pkg.add("NBInclude")
using NBInclude
```

Figure A.16.: Julia package and module imports

[12]:
```
@nbinclude("AdvancedBoard.ipynb")
```

[12]:
```
terminal_evaluation (generic function with 2 methods)
```

A.7.1. Engine

Importing Engines

[13]:
```
@nbinclude("RandomChess.ipynb")
@nbinclude("Minimax.ipynb")
@nbinclude("AlphaBetaPruning.ipynb")
@nbinclude("IterativeDeepening.ipynb")
```

[13]:
```
iterativeDeepening (generic function with 5 methods)
```

[14]:
```
function testStrength(board)
    display(board)
    boardScore = evaluate_position(board)
    aB = AdvBoard(board)
    println("Move Calculation")
    println("--------Alpha-Beta Pruning-------")
    for i in 1:6
        print("depth ", i , ": ")
        println(alphaBetaPruning(aB, i))
    end
    println("--------Iterative Deepening-------")
    for i in 1:6
        cache = initCache()
        print("depth ", i , ": ")
        println(iterativeDeepening(aB, i, cache, false, false, false))
```

```
    end
    println("--------Iterative Deepening with Quiescence-------")
    for i in 1:6
        cache = initCache()
        print("depth ", i , ": ")
        println(iterativeDeepening(aB, i, cache, true,  false, false ))
    end
end
```

Figure A.17.

```
[14]: testStrength (generic function with 1 method)
```

Opening White: Berlin Opening

```
[15]: b1 = fromfen("r1bqkb1r/pppp1ppp/2n2n2/1B2p3/4P3/5N2/PPPP1PPP/RNBQK2R w␣
      ↪KQkq - 0 1")
      testStrength(b1)
```

Figure A.18.: Result Comparing Strength of the Opening White: Berlin Opening

```
Board (r1bqkb1r/pppp1ppp/2n2n2/1B2p3/4P3/5N2/PPPP1PPP/RNBQK2R w KQkq -):
 r  -  b  q  k  b  -  r
 p  p  p  p  -  p  p  p
 -  -  n  -  -  n  -  -
 -  B  -  -  p  -  -  -
 -  -  -  -  P  -  -  -
 -  -  -  -  -  N  -  -
 P  P  P  P  -  P  P  P
 R  N  B  Q  K  -  -  R

Move Calculation
--------Alpha-Beta Pruning-------
depth 1: (295, Move(b5c6))
depth 2: (-50, Move(b5c6))
depth 3: (150, Move(b1c3))
depth 4: (-20, Move(b1c3))
depth 5: (130, Move(b1c3))
depth 6: (-80, Move(d2d3))
--------Iterative Deepening-------
depth 1: (295, Move(b5c6))
```

```
depth 2: (-50, Move(b5c6))
depth 3: (150, Move(b1c3))
depth 4: (-20, Move(b1c3))
depth 5: (130, Move(b1c3))
depth 6: (-80, Move(d2d3))
--------Iterative Deepening with Quiescence-------
depth 1: (40, Move(b5c6))
depth 2: (-10, Move(b1c3))
depth 3: (35, Move(b1c3))
depth 4: (-5, Move(b1c3))
depth 5: (30, Move(b1c3))
depth 6: (0, Move(b1c3))
```

Opening Black: King's Indian Defense

[22]:
```
b2 = fromfen("rnbqkb1r/pppppp1p/5np1/8/2PP4/2N5/PP2PPPP/R1BQKBNR b KQkq␣
↪- 1 3")
testStrength(b2)
```

Figure A.19.: Result Comparing Strength of the Opening Black: King's Indian Defense

```
Board (rnbqkb1r/pppppp1p/5np1/8/2PP4/2N5/PP2PPPP/R1BQKBNR b KQkq -):
 r  n  b  q  k  b  -  r
 p  p  p  p  p  p  -  p
 -  -  -  -  -  n  p  -
 -  -  -  -  -  -  -  -
 -  -  P  P  -  -  -  -
 -  -  N  -  -  -  -  -
 P  P  -  -  P  P  P  P
 R  -  B  Q  K  B  N  R

Move Calculation
--------Alpha-Beta Pruning-------
depth 1: (-5, Move(b8c6))
depth 2: (45, Move(b8c6))
depth 3: (-85, Move(b8c6))
depth 4: (95, Move(b8c6))
depth 5: (-65, Move(b8c6))
depth 6: (160, Move(e7e6))
--------Iterative Deepening-------
```

```
depth 1: (-5, Move(b8c6))
depth 2: (45, Move(b8c6))
depth 3: (-85, Move(b8c6))
depth 4: (95, Move(b8c6))
depth 5: (-65, Move(b8c6))
depth 6: (160, Move(e7e6))
--------Iterative Deepening with Quiescence-------
depth 1: (-5, Move(b8c6))
depth 2: (25, Move(d7d5))
depth 3: (15, Move(b8c6))
depth 4: (45, Move(b8c6))
depth 5: (15, Move(b8c6))
depth 6: (55, Move(e7e6))
```

Middlegame White: Ding Liren - Ian Nepomniachtchi, 2023

[23]:
```
b3 = fromfen("4r1k1/p1p1r1pp/1p2Pp1q/2pP1n1b/2P2p2/2Q2B1P/P2NRPP1/4R1K1␣
↪w - - 3 26")
testStrength(b3)
```

Figure A.20.: Result Comparing Strength of the Middlegame White: Ding Liren - Ian Nepomniachtchi, 2023

```
Board (4r1k1/p1p1r1pp/1p2Pp1q/2pP1n1b/2P2p2/2Q2B1P/P2NRPP1/4R1K1 w - -):
 -  -  -  -  r  -  k  -
 p  -  p  -  r  -  p  p
 -  p  -  -  P  p  -  q
 -  -  p  P  -  n  -  b
 -  -  P  -  -  p  -  -
 -  -  Q  -  -  B  -  P
 P  -  -  N  R  P  P  -
 -  -  -  -  R  -  K  -

Move Calculation
--------Alpha-Beta Pruning-------
depth 1: (285, Move(f3h5))
depth 2: (-40, Move(f3h5))
depth 3: (155, Move(d5d6))
depth 4: (-165, Move(d5d6))
depth 5: (80, Move(d2e4))
depth 6: (-170, Move(f3h5))
```

```
--------Iterative Deepening-------
depth 1: (285, Move(f3h5))
depth 2: (-40, Move(f3h5))
depth 3: (155, Move(d5d6))
depth 4: (-165, Move(d5d6))
depth 5: (80, Move(d2e4))
depth 6: (-170, Move(f3h5))
--------Iterative Deepening with Quiescence-------
depth 1: (-20, Move(d2e4))
depth 2: (-20, Move(d2e4))
depth 3: (-25, Move(f3h5))
depth 4: (-30, Move(d2e4))
depth 5: (-30, Move(f3h5))
depth 6: (-35, Move(f3h5))
```

Middlegame Black: Fabiano Caruana - Magnus Carlsen, 2018

[18]:
```
b4 = fromfen("2bqk1nr/1p1p1pbp/6p1/1pp1p3/1P2P3/5N2/2PP1PPP/BN1QR1K1 b␣
↪k - 0 10")
testStrength(b4)
```

Figure A.21.: Result Comparing Strength of the Middlegame Black: Fabiano Caruana - Magnus Carlsen, 2018

```
Board (2bqk1nr/1p1p1pbp/6p1/1pp1p3/1P2P3/5N2/2PP1PPP/BN1QR1K1 b k -):
 -  -  b  q  k  -  n  r
 -  p  -  p  -  p  b  p
 -  -  -  -  -  -  p  -
 -  p  p  -  p  -  -  -
 -  P  -  -  P  -  -  -
 -  -  -  -  -  N  -  -
 -  -  P  P  -  P  P  P
 B  N  -  Q  R  -  K  -

Move Calculation
--------Alpha-Beta Pruning-------
depth 1: (-170, Move(c5b4))
depth 2: (-20, Move(c5b4))
depth 3: (-265, Move(d8f6))
depth 4: (-15, Move(c5b4))
depth 5: (-155, Move(d8e7))
```

```
depth 6: (40, Move(c5b4))
--------Iterative Deepening-------
depth 1: (-170, Move(c5b4))
depth 2: (-20, Move(c5b4))
depth 3: (-265, Move(d8f6))
depth 4: (-15, Move(c5b4))
depth 5: (-155, Move(d8e7))
depth 6: (40, Move(c5b4))
--------Iterative Deepening with Quiescence-------
depth 1: (-85, Move(d7d6))
depth 2: (-20, Move(c5b4))
depth 3: (-70, Move(c5b4))
depth 4: (-30, Move(c5b4))
depth 5: (-65, Move(c5b4))
depth 6: (-35, Move(c5b4))
```

Endgame White: Magnus Carlsen - Ian Nepomniachtchi, 2021

[19]:
```
b5 = fromfen("8/R7/6k1/3q3p/5P2/4P1P1/4NK2/8 w - - 1 83")
testStrength(b5)
```

Figure A.22.: Result Comparing Strength of the Endgame White: Magnus Carlsen - Ian Nepomniachtchi, 2021

```
Board (8/R7/6k1/3q3p/5P2/4P1P1/4NK2/8 w - -):
 -  -  -  -  -  -  -  -
 R  -  -  -  -  -  -  -
 -  -  -  -  -  -  k  -
 -  -  -  q  -  -  -  p
 -  -  -  -  -  P  -  -
 -  -  -  -  P  -  P  -
 -  -  -  -  N  K  -  -
 -  -  -  -  -  -  -  -

Move Calculation
--------Alpha-Beta Pruning-------
depth 1: (150, Move(e3e4))
depth 2: (95, Move(a7c7))
depth 3: (155, Move(e2c3))
depth 4: (105, Move(e2d4))
depth 5: (180, Move(a7e7))
```

```
depth 6: (85, Move(a7a6))
--------Iterative Deepening-------
depth 1: (150, Move(e3e4))
depth 2: (95, Move(e2c3))
depth 3: (155, Move(e2c3))
depth 4: (105, Move(e2d4))
depth 5: (180, Move(a7e7))
depth 6: (85, Move(a7a6))
--------Iterative Deepening with Quiescence-------
depth 1: (145, Move(e2d4))
depth 2: (135, Move(e2c3))
depth 3: (125, Move(e2d4))
depth 4: (125, Move(e2d4))
depth 5: (135, Move(e2d4))
depth 6: (135, Move(e2d4))
```

Endgame black: Mikhail Botvinnik - Robert James Fischer, 1962

[20]:
```
b6 = fromfen("8/p5R1/1p4p1/4k3/r6P/6K1/5P2/8 b - - 4 50")
testStrength(b6)
```

Figure A.23.: Result Comparing Strength of the Endgame Black: Mikhail Botvinnik - Robert James Fischer, 1962

```
Board (8/p5R1/1p4p1/4k3/r6P/6K1/5P2/8 b - -):
 -  -  -  -  -  -  -  -
 p  -  -  -  -  -  R  -
 -  p  -  -  -  -  p  -
 -  -  -  -  k  -  -  -
 r  -  -  -  -  -  -  P
 -  -  -  -  -  -  K  -
 -  -  -  -  -  P  -  -
 -  -  -  -  -  -  -  -

Move Calculation
--------Alpha-Beta Pruning-------
depth 1: (-220, Move(a4h4))
depth 2: (-120, Move(a4a3))
depth 3: (-200, Move(e5f6))
depth 4: (-60, Move(e5f6))
depth 5: (-180, Move(e5f5))
```

```
depth 6: (-50, Move(e5f6))
--------Iterative Deepening-------
depth 1: (-220, Move(a4h4))
depth 2: (-120, Move(a4a3))
depth 3: (-200, Move(e5f6))
depth 4: (-60, Move(e5f6))
depth 5: (-180, Move(e5f5))
depth 6: (-50, Move(e5f6))
--------Iterative Deepening with Quiescence-------
depth 1: (-120, Move(a4a3))
depth 2: (-120, Move(a4a3))
depth 3: (-130, Move(a4a3))
depth 4: (-110, Move(e5f5))
depth 5: (-120, Move(a4a1))
depth 6: (-120, Move(a4a3))
```