Development of a Chess AI and an Interactive Learning Game

Quim Camprubí Casas

Abstract– Chess is one of the most ancient games in humanity's history. It has been studied and explored for centuries, and the resulting knowledge has been stored in books of theory, still used today to teach chess apprentices. Since their introduction in the 20th century, chess algorithms have proved that chess is very computationally intensive at high-level play. However, thanks to the ever-increasing computational prowess of modern computers, as well as software-side optimizations, the best modern chess engines have long surpassed human capabilities. In 1997, the famous computer Deep Blue made history by defeating reigning world champion Garry Kasparov in a series of 6 games, after losing to the world champion the previous year.
Being fascinated by the field of AI algorithms, and as a chess beginner myself, I set off to develop a game, called Easy Chess, as the Final Project of my Computer Engineering degree, that would help players hone their chess abilities and allow them to become better players. This article will go over the development of a chess engine within the computer game, by looking at my initial objectives, the methodology I followed to implement the algorithm, some development details, as well as the final results of the chess engine and the game's interface.
Keywords– Software development, chess, AI, Unity, chess engine, Decision Tree, Minimax, Alpha-Beta pruning, 2D game, interactive learning, computer video game

1 Introduction

The project consisted in the development of an Artificial Intelligence algorithm capable of playing chess at a high level, as well as the development of a 2D interactive game aimed at people who want to practice against a computer, both with the hopes of learning to play better, and with the intention of having fun while doing so.

In the context of the final game, this AI algorithm, or *engine*, as they're known in the chess world, serves as the human player's opponent for every one of the chess games the user plays. The engine analyzes all the possible legal moves it can make in a given position, and chooses the one it deems as the most optimal one, intending to defeat the user.

As the main goal of the project was to create an interactive learning game for chess aficionados, I decided to implement 2 main game modes for the game:

- **Competitive Mode**: In this game mode, the user plays against the AI without any help or clues from the UI.
- **Learning Mode**: In this game mode, the user interface displays additional information about the state of the chess game, such as position evaluation or the optimal move they can make. It also allows the user the possibility of undoing their last move. The game mode uses all these functionalities to allow the user to better understand the game their playing, and serves as a playground mode where they can have further insights into the game of chess.

On top of the different game modes, the game offers multiple difficulty levels, allowing for a more uniform experience for players of different chess abilities, and facilitating closer games between the user and the AI.

As for the engine's algorithm itself, it uses a decision tree [1] to represent all the moves it can make in a given position, as well as their possible continuations. The *depth* parameter indicates how many moves in succession the engine shall look at before deciding on the best possible move. For example, a depth of 2 indicates that, for each of the possible moves it can make, the engine will explore the possible responses from the opposing player. As can be expected, a higher depth will yield better accuracy from the engine, but at an exponentially higher computational cost.

Depth is also a crucial parameter when tweaking difficulties, as a lower depth will mean the engine makes poorer choices, which is ideal for an easier difficulty. Likewise, a higher difficulty will use a higher depth, as well as multiple heuristics that enhance the engine's ability to choose the best move. As the depth parameter is crucial to the quality of the engine, optimization and constant improvement of the AI algorithm was a fundamental aspect of the development of this project.

While this document will not go into high-level chess the-

• E-mail de contacte: quimcc08@gmail.com
• Menció realitzada: Computació
• Treball tutoritzat per: Ramón Baldrich (Ciències de la Computació)
• Curs 2021/22

ory, some aspects of the project's implementation will touch upon chess principles. If you deem it necessary to go over them, the rules of chess can be found on the US Chess Federation website [2].

2 Motivation and Goals

As a Computer Science student, I had always been very interested in the field of AI algorithms capable of playing human-designed games. I've also been a chess player for quite some time. As such, when I was given the chance to write my own Final Project proposal, I very quickly thought of developing a chess engine within a learning game, not only with the hopes of creating a strong program, but of hopefully improving my own chess abilities as well. The main goals of this project were the following:

- **Develop a chess engine capable of playing chess at a higher competitive level than the majority of humans.** As I wanted to create a game for players who were looking to improve their skills, I would have to create a strong engine. According to the US Chess Federation (USCF), 67% of all players who compete in official tournaments have an ELO score of 1400 or less [3]. Therefore, I set the goal of surpassing an ELO of 1400 as a good starting point, as that would mean that my engine would be able to best most players. It should be emphasized that this score does not reflect the chess ability of all human beings, or of all chess players, but rather only of those who engage in official tournaments in the United States, meaning a chess player with an ELO of 1400 should be much better at chess than the average person.

- **Develop a 2D computer game which would allow the player to face the chess AI.** This game would allow the player the possibility of choosing between a competitive game mode, and a more informative game mode designed for learning, rather than testing the player's ability. This game had to have an interactive user interface compatible with Windows, Linux, and Mac computers.

In order to fulfill these goals and evaluate their completion, I had to carry out the following crucial actions:

- **Evaluate the engine's performance against other chess engines with different ELO ratings.** With the goal of determining the quality of my chess engine, it had to be tested against players of high caliber, such as different chess engines. This would allow me to estimate an ELO for my own engine.

- **Evaluate the engine's performance against human players.** As the main goal of the project is to create an engine capable enough for humans to learn by playing against it, the engine had to be strong enough to play well against human opponents and surpass the goal of 1400 ELO.

- **Optimize the performance of the AI's algorithm.** Given that the engine had to be used in an interactive 2D game, response time was crucial. Regardless of how intelligent the engine was, it would not be suitable for the project if it took hours to choose a move, which meant that programming an efficient algorithm was one of the main requirements of the project.

3 State of the Art

Due to the nature of chess, decision trees are the graph types used to represent all the possible moves a player can make. When a player has to choose a move, it has a decision tree, where each of the different depth levels is a different decision by the Black and White players, respectively. Each possible move would represent a different branch within the same tree level. The depth parameter that was briefly explained in the Introduction section is the number that determines how many levels the algorithm will explore for each of the possible moves at the root of the decision tree.

Because finding a good move given a chess position is ineffective without looking at future moves, chess engines rely on a *Search* algorithm, which is used to traverse this decision tree structure, while min-maxing the different decisions, depending on the player the current level represents. The most commonly-used one is the Minimax algorithm [1], which is used to select the best movement given a position. In a chess decision tree, every move has a score (obtained using the *Evaluation* function), and the algorithm always selects the move that maximizes or minimizes said score, depending on which level of the decision tree it is currently in. If the depth level corresponds to the current player, the selected move will be the one with the maximum score at that level. On the other hand, if the current depth level corresponds to the opposing player, the selected move will be the one with the minimum score at that level, because the opposing player should always choose the best move for their interests, that is, the worst possible move from the current player's point of view. This technique allows the current player to choose the best move in a given position.

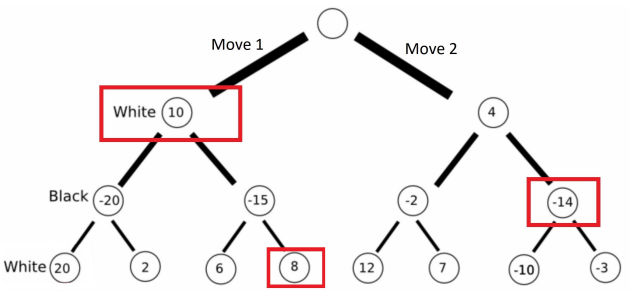

Fig. 1: Minimax example. The red rectangles represent the Min-Maxed evaluation of the chess position after the best moves are made at each depth level

An example of the Minimax algorithm can be found in Figure 1. The current player would be White, and for simplicity's sake, they would only have 2 moves to choose from (in the real chess implementation, there will be many more possible moves on most positions). Because White is the current player, depths 1 and 3 would be maximizing layers, while depth 2 would be a minimizing layer, as that would correspond with Black's choice. If the decision were to be taken with a depth of 1, White would choose Move 1, as it

maximizes the score ($10 > 4$).

On the other hand, if the decision were to be taken with a depth of 2, all the scores after the two successive moves would be minimized, meaning that Black would choose those with the lowest possible values: -20 for Move 1, and -14 for Move 2. White would have to choose Move 2, as it would maximize their score ($-20 < -14$).

Finally, if the search used a depth of 3, White would choose Move 1, as after the 2 maximizing layers and 1 minimizing layer, it would have to choose between a score of 8 (Move 1) and -3 (Move 2).

From this quick example, we can already see that higher depths allow us to make more informed decisions, by taking a look at more moves and responses. As we already previously introduced though, increasing depth will also exponentially increase computational complexity, as it will dramatically increase the number of nodes the algorithm has to evaluate. To mitigate this effect, one of the main goals of any chess engine is to reduce the number of branches it has to explore. This is usually done using the Alpha-Beta pruning algorithm [1], [4], [5], which uses the scores of previously explored moves to *prune* (ignore) those branches which prove to be worse than the previously explored ones without exploring them to their full depth.

This is the basic structure that most chess engines use, but there are also many different improvements, optimizations, heuristics, and techniques used both to speed up the execution of the Search algorithm, as well as to improve the evaluation capabilities of the engine:

- **Move ordering.** [1] It consists in ordering the different branches to explore based on preliminary scores (not the final evaluation), with the goal of exploring the best moves first. Because of the Alpha-Beta algorithm's behavior, exploring those moves with the best scores first will yield higher prune rates, because those poor moves will be detected earlier, by comparing them to the already explored good moves. This improvement is highly dependant on how the moves are scored, given that the preliminary score does not look at future moves, but rather uses heuristics and chess theory to estimate how good a possible move might be. This technique allows for higher depths and lower complexity, as it reduces the amount of nodes that are explored in a given position.

- **Quiescence.** [6] Quiescence Search is a limited search that is added at the maximum defined depth, to avoid the Horizon Effect [1]. This effect might occur when the last explored move is a capture of an enemy piece. Because captures benefit the player, a simple Minimax algorithm might deem a capturing move as the best option, without taking into account the consequences of the capture. One of the basic principles of chess is to protect pieces with other pieces, which means that the likelihood of a capture being followed by a recapture is high. However, due to the Horizon Effect, a basic Minimax algorithm will ignore this principle. Quiescence search is a short search added when the last move is a capture, with the intent of exploring whether or not that capture is actually a safe move or if it will have negative consequences for the player.

- **Transposition table.** [4], [1] A transposition table is a data structure used to reduce the computational complexity by reusing previously explored positions. Because of the nature of the decision tree, the Search algorithm will often evaluate positions it has previously explored. Exploring the position again would be computationally expensive, so the transposition table avoids doing that by storing the best moves and scores in a given position. When the algorithm reaches a position it has explored before, the transposition table will contain the best move and score, as well as the depth used in the exploration. If the depth of the entry is greater than, or equal to, the current depth of the search algorithm, the Minimax function will use that move and its score as the chosen one.

- **Evaluation heuristics.** [1] A basic evaluation function would use the value of the different pieces in the board to determine the final evaluation. However, there are also multiple heuristics that can be used to enhance this evaluation by using chess theory, which allows for more information to be used when choosing the optimal move. Some examples of these heuristics are pawn structure heuristics, piece-square tables, and null-move pruning.

The algorithm I've implemented uses all the techniques mentioned in this section, with various heuristics which will be explained in the Implementation section.

As for board representation, there are multiple techniques [1], and they are designed with the goal of reducing the complexity of the necessary calculations, and to allow for easier implementation. The main alternatives are:

- **Piece centric.** This technique uses a list with all the different pieces in the board, with the resulting structure containing information of all the pieces. Such information might be the piece type, its position on the board, etc. This approach is typically used to optimize move generation.

- **Square centric.** In this approach, there is usually a list or matrix containing all the squares in the board, with information for each of those squares, such as the piece it might contain. This is most typically represented with 8x8, 10x12 or 0x88 lists.

- **Hybrid solutions.** This approach uses both techniques due to their different strengths and weaknesses. For example, a hybrid approach might use piece lists for move generation, and a squares list to make moves.

As for my algorithm, it uses a hybrid solution, combining the best aspects of each technique. It uses piece lists for move generation, avoiding having to loop through all the squares, and only looping through all the pieces to generate their moves. At the same time, it uses an array of size 120 (10x12) to organize the board, display it on screen, and check whether a given square is empty, or what piece it contains.

4 METHODOLOGY

I used an Agile methodology to carry out the entirety of this project using three-week sprints. Because it was an individ-

ual endeavor, I decided against following more strict Scrum practices, since everything was managed by myself. Each sprint had a Sprint Goal, that is, the main goal of the sprint in question. On top of that, the sprint would have a number of tasks, which could be changed during the sprint based on priorities, development speed, or new circumstances.

At the end of the sprint, there was a Combat Testing session, where I tested the software's version with all the new developments. Based on the results of this testing, I would create tasks for the next sprint to fix or improve any aspect of the new developments if deemed necessary.

Other than the Combat Testing sessions, as soon as I had a playable version of the game with a preliminary engine implementation, I conducted multiple playtesting sessions with external users, with the goal of finding bugs and potential improvements.

In order to carry out this Agile methodology, I used Trello, a lightweight software tool designed to manage Agile teams. My workspace can be found in the following website: https://trello.com/treballfinaldegraulearnchess. The workspace stores multiple boards, one board for each sprint. As soon as the sprint ended, I would transfer all the unfinished tasks to the next board, and I would create all the necessary tasks for said sprint. This methodology allowed me to follow a planned development path, and helped me keep track of all the necessary actions and steps I had to take to complete my project.

As for version control, I used a GitHub repository where I kept updating all the new developments, with descriptions for all the changes that were made in each commit. This methodology allowed me to have good traceability, which helped me keep track of all the changes made to the code. The repository can be found in the following link: https://github.com/quimcamprubi/Learn-Chess.

4.1 Technologies

I chose the Unity engine to develop the 2D game, as it is one of the most popular engines in the market, particularly for 2D games. Thanks to this, it is very well documented and it offers the functionality to build the game for many different platforms. While I had some previous experience working with Unity, it was very limited, which meant that the game's implementation would create a learning curve. Thankfully though, working with Unity was very smooth, as it is a very mature game engine, with many helpful official tutorials for the most common implementations.

The programming language I used comes predefined by Unity, as the only language it supports for its scripts is C#. C# is an Object-oriented programming language with some similarities to C++, although it is at a higher level of abstraction. Most chess engines use C or C++ [1], due to the fact that it allows for very fine memory management, and it provides many optimization opportunities. These engines do not usually have a graphical interface, instead opting for the Universal Chess Interface (UCI) to communicate with websites or other programs. I also had very limited experience with C# before starting this project, but thanks to the fact that it shares many similarities with other Object-oriented programming languages, programming with C# was a relatively smooth experience as well.

5 Implementation Details

This section contains in-depth explanations for the most relevant parts of the program, most importantly, the core engine algorithm responsible for playing chess. It also goes over some of the most important UI functionalities, such as the Learning Mode and the information it displays to the user.

5.1 Board Representation

As we already briefly discussed in the State of the Art section, my program uses a hybrid board representation structure, by combining both piece-centric approaches with square-centric approaches.

The most important aspect of the board's representation is the main *squares* array. This data structure with a size of 120 positions, is used to represent a 10x12 board, containing integers for each of the board's squares. This integer value represents the type of piece that is present in the given square, ranging from 1 to 12, if any is present at all. If there is no piece in the square, it has a value of 0.

As you may have noticed, the chosen structure contains more positions that is strictly necessary to represent a chess board, which has a size of 8x8, meaning a total of 64 squares. The array has two rows of extra squares over and below the actual chess board, as well as an extra column at each side of the board. I chose to use a 10x12 board because it makes it relatively simple to handle moves that would end up outside of the board, and that are therefore, illegal. If any move ended outside the chess board, it would return an *offboard* value, which would make the move illegal.

	100	100	100	100	100	100	100	100	100	100
	100	100	100	100	100	100	100	100	100	100
8	100	10	8	9	11	12	9	8	10	100
7	100	7	7	7	7	7	7	7	7	100
6	100	0	0	0	0	0	0	0	0	100
5	100	0	0	0	0	0	0	0	0	100
4	100	0	0	0	0	0	0	0	0	100
3	100	0	0	0	0	0	0	0	0	100
2	100	1	1	1	1	1	1	1	1	100
1	100	4	2	3	5	6	3	2	4	100
	100	100	100	100	100	100	100	100	100	100
	100	100	100	100	100	100	100	100	100	100
		A	B	C	D	E	F	G	H	

Fig. 2: Squares array at the starting position. Squares with a value of 0 are empty, with numbers ranging from 1 to 12 representing the different possible pieces. Squares with a value of 100 are *offboard*, and are used to identify the edges of the board while generating moves.

An example of the *squares* array corresponding to the initial chess position can be found in Figure 2.

This *squares* array is the one that is used internally throughout the entire program to generate moves, search positions, and make moves. However, there are still some operations that have to be done on a regular 64 squares array, which are mostly the ones related to the user interface. In order to translate between both of those arrays, the program has 2 translation matrices, used to translate 64 based indices to 120, and vice versa.

As for the pieces themselves, the program uses a matrix of 13x10 used to represent the piece list, that is, an array containing each of the pieces on the board. The dimensions are determined by the fact that there are 13 different pieces, with a maximum of 10 pieces for each piece. In any given position there could be, for example, 2 Rooks + 8 Rooks which were promoted as Pawns, for a total of 10 Rooks. Even though this will most likely never happen, I decided to have a fixed size array for optimization purposes, as it is usually better not to resize arrays during the Search algorithm, which explores millions of nodes/second.

5.2 Move Representation

Moves are represented using a 32-bit integer which stores the relevant information of the move in its 25 least significant bits using binary code.

Fig. 3: Move format. The 7 least-significant bits are used to store the move's starting square (0 to 119). The next 7 bits represent the move's destination, followed by the piece being captured (4 bits), the En Passant capture and Pawn Start flags (1 bit each), the promoted piece (4 bits), and the Castling flag (1 bit).

Figure 3 displays an example of how these 25 bits are used to encode a move that would go from square 6 to square 12, capturing piece 4 (which corresponds with a White Rook). The move is also not an En Passant capture (bit set to 0), nor a Pawn start (double forward move). The move is a promotion to piece 10 (Black Rook), and it is not a castling move, as the corresponding bit is set to 0.

In order to encode these moves, bitwise left shift operations are performed to move the relevant numbers to their corresponding positions, and OR operations are performed between both operands.

On the other hand, in order to extract information from the move, an AND bitwise operation is performed with the corresponding mask for each of the data that needs to be extracted.

All these bitwise move operations are encapsulated into higher level functions, with the goal of facilitating the development of the engine and improving readability.

The goal of representing moves as 32-bit integers is to optimize memory usage, as well as reduce the complexity of the Move class. Instead of using multiple fields for all the data that needs to be stored, a single integer suffices, and contains all the necessary data through very efficient bitwise operations.

5.3 Move Generation

The Move Generation algorithm is one of the most crucial aspects of the program. As I wanted to create a fully-legal and fleshed out chess game, I had to make sure that my program complied with the rules of chess [2].

At its core, the Move Generator iterates through all the pieces in the piece list (the 13x10 array we talked about in the previous section) and generates all the pseudo-legal moves for each piece. Pseudo-legal moves are moves that comply with the basic rules of a piece's movement without accounting for King Safety. A move is only fully legal if it does not discover the player's King to an opposing piece's attack after it is played. In order to generate the final legal moves, the pseudo-legal ones are played on the board, with King Safety being checked after they are played, then they are unmade, and finally, a boolean is returned regarding whether or not the King is in check.

In actuality though, the generation of move differs greatly depending on the different types of pieces. The move generation algorithm is split into 3 parts:

- **Pawns:** Given that pawns have many different types of moves, and that they attack in different directions than their regular moves, their moves are generated separately from other pieces. Pawns generate regular forward moves, double forward moves (pawn start), diagonal captures in both directions, En Passant captures, and promotions into Queens, Rooks, Bishops and Knights.

- **Sliding pieces:** Sliding pieces are those whose moves can extend until the edge of the board. Queens, Rooks and Bishops are all sliding pieces. In order to generate their moves, we loop through all their possible directions and keep advancing until the edge of the board, where the *squares* array returns an *offboard* value (100).

- **Non-sliding pieces:** Non-sliding pieces can move in a given set of directions only once. Kings and Knights are non-sliding pieces, and to generate their moves we simply iterate through their different directions.

One major aspect of move generation is scoring. As we introduced in the State of the Art section, all generated moves get a preliminary score based on chess theory, with the goal of improving move ordering, and thus, improving the amount of nodes that the Alpha-Beta algorithm is able to prune. In order to score the moves, the program uses multiple different heuristics, which can be divided between capture moves and quiet moves (non-captures):

- **Capture moves:** Capture moves are scored using the MVV-LVA Heuristic (Most Valuable Victim - Least Valuable Attacker) [1], which awards a score based on the value of the attacker and its victim. It prioritizes moves where a low-value attacker captures a high-value victim, as that would mean that, even if the attacker is put at risk by the capture, being able to trade it for a higher-value piece is a net win for the player.

- **Quiet moves**: Quiet moves use 2 different heuristics:

- **Killer Heuristic** [1]: This heuristic prioritizes moves that caused a Beta cutoff in a different position to the one being currently evaluated, as those moves are probably going to be good even if coming from a different prior move combination. My engine has two killer moves, the most recent one with a higher score than the second most recent one, with both of them being scored higher than regular quiet moves.
- **History Heuristic** [1]: This heuristic adds a score based on the number of cutoffs a given move has produced before, irrespective of the position they were tested in. This heuristic aggregates the number of previous beta cutoffs the move produced and adds that counter to the move's preliminary score.

After the moves have been generated, and their preliminary scores assigned, the algorithm returns the list of moves to the main Search algorithm which iterates through each of them to explore the decision tree.

5.3.1 Perft Testing

As the Move Generator algorithm is an integral part of the engine, it had to be tested thoroughly, to ensure that no hidden bugs or errors were present. Perft Testing [1] is the technique used to precisely test whether a Move Generation algorithm works as expected or if it has issues hidden within. It consists in counting the terminal nodes there are in a given position with a certain depth, and comparing the result with the correct numbers [1]. If the numbers align at a certain depth, it means that the algorithm is very likely working correctly at the given depth, as the amount of nodes quickly reaches the range of millions of nodes.

My program has fully correct Perft Testing results with multiple tested positions at depth 7. The test's results can be found in Appendix 3.

5.4 Move Ordering

As we've previously introduced, Move Ordering is a crucial aspect in the efficiency of the Minimax algorithm, good move ordering ensures that the best variations are explored first and a large number of nodes are pruned. The Move Ordering in my program uses the following criteria:

1. Hash Move: The Hash move is the one stored in the transposition table, that is, the first move in the Principal Variation. It is the best move found in previous searches for the current position. Therefore, it is a good starting point for the current search.
2. Captures (MVV-LVA): Capture moves are always important, and those with highest scores (low-value attackers, high-value victims) are explored first.
3. Killer Move 0
4. Killer Move 1
5. History move
6. Rest of the moves

5.5 Position Evaluation

The Evaluation algorithm is used to extract an integer score for how good or bad a position is for the side to play. This score is fundamental to the engine's performance, as it is the scored that is used to determine how good or bad a given position is for the engine, and therefore, to choose which move to play at every moment of the game. It is different from move scores because those are preliminary, and only used to order moves. The position evaluation score is the deciding factor between different moves, and it is the value used to iterate through the Alpha-Beta algorithm.

The basic aspect of this function is a count of the material for both sides. Material count is the sum of the value of the pieces present in the board. In my algorithm, Pawns have a value of 100, Knights and Bishops have a value of 325, Rooks have a value of 550, Queens have a value of 1000, and Kings have a value of 50000. These values change slightly depending on the different engines [1], but they are always similar to the scores I used. The evaluation function uses these scores, as well as the different heuristics to determine the quality of the position.

The following heuristics come into play:

- **Piece-Square tables**: Piece-square tables store an integer that indicates how good or bad a given square is for a certain piece in a given situation on the board. They can be understood as a way to introduce book theory to the Evaluation function, by applying chess knowledge to the engine. Each piece has at least one table, with some of them having multiple tables (depending on the situation of the board). Each square on the table contains a score for how good or bad a square is for the piece. This way, when 2 moves look very similar due to material count, these tables are used to break the tie and choose the theoretical best move. As an example, the King's Opening table can be seen in Figure 4.

  As can be seen by its values, the King is encouraged to stay in the corners of its side of the board while on the Opening phase of the game, with forward moves being increasingly penalized, as those will compromise its safety. The King has a different board for later stages of the game, where it is encouraged to move to the center of the board to help its remaining pieces.

```
static readonly int[] KingOpening = {
    0   ,   5   ,   5   ,  -10  ,  -10  ,   0   ,  10  ,   5   ,
   -30  ,  -30  ,  -30  ,  -30  ,  -30  ,  -30  ,  -30  ,  -30  ,
   -50  ,  -50  ,  -50  ,  -50  ,  -50  ,  -50  ,  -50  ,  -50  ,
   -70  ,  -70  ,  -70  ,  -70  ,  -70  ,  -70  ,  -70  ,  -70  ,
   -70  ,  -70  ,  -70  ,  -70  ,  -70  ,  -70  ,  -70  ,  -70  ,
   -70  ,  -70  ,  -70  ,  -70  ,  -70  ,  -70  ,  -70  ,  -70  ,
   -70  ,  -70  ,  -70  ,  -70  ,  -70  ,  -70  ,  -70  ,  -70
};
```

Fig. 4: King Opening table. Each square has a score, related to how good or bad each square is for the King in the opening phase of the game.

- **Pawn structure heuristics**: Pawn structure is elemental in chess theory, and it teaches chess learners to keep a cohesive pawn structure that allows the pawns to de-

fend one another. Isolated and Doubled pawns are often discouraged [7], while Passed pawns tend to be valued much higher than regular pawns, as they are free of enemy pawns and have a clear way to promotion at the end of the board. For this reason, the Evaluation algorithm applies penalties to pawns in compromised positions (isolated pawns), and bonuses to passed pawns. In the case of the passed pawns, the closer they are to the end of the board, the more valuable they become.

In order to carry out these calculations, pawn bitboards are used, which are 64-bit unsigned integers. Each bit represents a square, and its value (1 or 0) represents whether a pawn is located in the square, or not. To determine whether a pawn is isolated, for example, an AND bitwise operation is carried out between the pawns bitboard and the isolated pawn bit mask for the current pawn, which is used to check whether or not the pawn has allied pawns in the neighboring files. If the operation returns a 0, it means that the pawn is indeed isolated, and therefore a penalty of -10 score is applied, meaning that the final value of the pawn is 90.

- **Rooks and Queens heuristics**: Rooks and Queens on open files get a +10 bonus to their score, while those located on semi-open files get a +5 bonus to their score. This heuristic applies chess theory, by making sure that Rooks and Queens are covering the highest amount of usable squares, which is when they are known to be most useful.

- **Bishop heuristics**: When a side has the Bishop Pair (both Bishops alive), they receive a +30 bonus, as chess theory tells us that Bishops are much more valuable when used in conjunction with one another.

5.6 Minimax algorithm

The Minimax algorithm is the most important aspect of the engine's performance. It contains the main loop used to explore the tree, the Alpha-Beta implementation, and the Quiescence Search used to limit the Horizon Effect. As it is a tree exploration algorithm, it is a recursive algorithm, which explores the tree and propagates the results of the exploration upwards through the function's return statements. More precisely, my program uses a Negamax algorithm [1], which is a Minimax subtype. Traditionally, there would be a Maximizing function and a Minimizing function, and they wouble be called in alternating order while exploring further into the tree. Negamax implements the same functionality, but applies the mathematical relation $max(a, b) = -min(-a, -b)$ to avoid the usage of two different functions, instead using a single function. A simplified pseudocode version of the RecursiveAlphaBeta() function can be seen in Algorithm 1.

Algorithm 1 RecursiveAlphaBeta(alpha,beta,depth)

```
if depth = 0 then
  return QuiescenceSearch(alpha, beta, parameters)
end if
CheckTimeLimit()
bestMove ← hashTable.GetMove()
if bestMove not null then
  return bestMove.score
end if
NullMovePruning()
movesList ← GenerateAllMoves()
movesList ← OrderMoves(movesList)
bestScore ← −Infinite
for each move in movesList do
  MakeMove(move)
  if move is not legal then continue
  score ← −RecAlphaBeta(−B, −A, depth − 1)
  UnmakeMove()
  if score > bestScore then
    bestScore ← score
    bestMove ← move
    if score > alpha then
      if score ≥ beta then
        if move not capture then
          StoreKillerMove()
        end if
        StoreHashMove(BETA)
        return beta
      end if
      alpha ← score
      if move not capture then
        StoreHistoryMove()
      end if
    end if
  end if
end for
CheckForMates()
if alpha has improved then
  StoreHashMove(ALPHA)
end if
return alpha
```

5.6.1 Quiescence Search

The Quiescence Search algorithm is very similar to the regular Alpha-Beta algorithm, with the difference being that it only evaluates capture moves, as it seeks to mitigate the Horizon Effect. Therefore, it does not use a depth parameter but instead keeps exploring the tree while captures are possible on the board. Otherwise, it functions very similarly to the regular Alpha-Beta function. If there are no more captures to make in a given position, the function returns the final Evaluation of the position.

5.7 Iterative Deepening

Iterative Deepening [1] is a technique used to constraint move searching using time constraints instead of a depth limit. An approximate pseudocode to the Search algorithm can be found in Algorithm 2.

As you can see in the pseudocode, the algorithm keeps increasing depth until the set depth limit, which is the one corresponding to the difficulty that has been set. That is done in order to limit the engine's quality, as the lower difficulties use very low depths, which allows the Search algorithm to finish exploring the tree within milliseconds. The loop can also be stopped if the search has been stopped from within the AlphaBeta function.

It would seem that repeating searches with higher depths

Algorithm 2 SearchPosition

$bestScore \leftarrow -Infinite$
$bestmove \leftarrow null$
for $depth = 1; depth < maxDepth; depth++$ **do**
  $bestScore \leftarrow RecAlphaBeta(Inf, -Inf, depth)$
  if $search$ is stopped **then**
    break
  end if
  $bestMove \leftarrow hashTable.GetPvMove()$
  if $player = Black$ **then**
    $currentEvaluation \leftarrow -currentEvaluation$
  end if
end for
return $bestMove$

would be highly inefficient, as we would be repeating explorations of the same tree, but actually, Iterative Deepening allows for a more efficient search and a more responsive program. It is so because, during the Search algorithm, we store the Principal Variation (the best move combination) in the Hash Table (Transposition Table), which allows the algorithm to start the next search with the best move variation from the previous search through move ordering.

5.8 Front-End and User Experience

This section will go over the most important functionalities in the program's Front-End, that is, the game's UI, the different game modes and difficulties, as well as some interactive aspects of the game.

When the game launches, it allows the user to choose their desired game mode and difficulty. After that, the chess game is started, with the user being defaulted to White. The user is allowed to switch sides at any point during the game, which will end the current game and start a new one with the opposing side.

5.8.1 Game modes

As for the game's interface, the Challenge Mode has a very simple interface. Other than the game's controls themselves, it only allows the player to go back to the main menu or to switch sides and restart the game. It is a competitive game mode, intended only for the user to test themselves against the engine. On the other hand, the Learning Mode offers a plethora of interactive UI elements designed to help the user improve their chess abilities. A screenshot of a Learning Mode game can be found in Figure 5.

The most important aspects of this game mode are the fact that it allows the user to undo their previous moves (using the Undo Move button in the bottom right-hand portion of the screen), and the Heat map functionality, which aims to display the most crucial and contested squares in the current position.

The Learning Mode displays further information designed to help the user understand how the game is going, and if the moves they are playing are strengthening or weakening their position.

On the right-hand side of the screenshot we can see a positive float value (+0.75), which is the current evaluation of the board. This value tells the user how well the game is going from their point of view. In this case, White has a +0.75 evaluation, which means that, according to the engine, they have a very slight advantage. This evaluation value is accompanied by a short status message, in this case, "Close game!". If the evaluation is showing a more decisive advantage for either player, it displays a "White is winning!" or "Black is winning!" message.

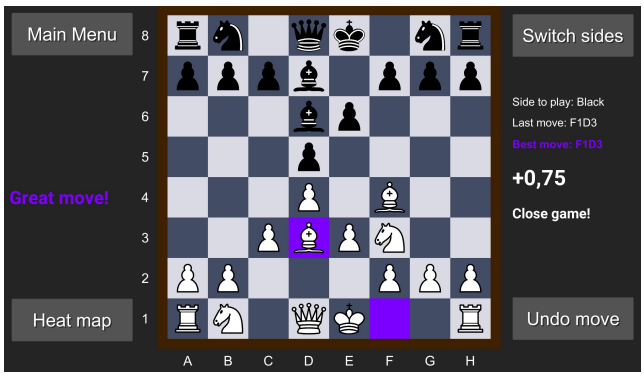

Fig. 5: Learning mode UI. The interface allows the user to navigate to the main menu, switch sides, toggle the Heat Map, and undo the last move. The best move in the position is displayed in purple.

Another functionality of the Learning Mode is the feedback it provides on the moves being played. In the example screenshot, the user just played the move F1D3, which coincides with the best possible move, according to the engine. Because of this, the message "Great move!" appears on the left-hand side of the screen, while visual feedback is also displayed on the board. On the other hand, if the user had played a bad move, such as a move that hang a piece, a "Blunder" message would be displayed instead, with red-colored visual feedback being displayed on the board. All of this feedback is designed to help the user understand the flow of the game, and to see how their moves are impacting their position.

5.8.2 Heat Map Analysis

The Heat Map functionality is a toggle that the user can enable to show the most contested, and therefore, the most important squares in a given position. It can also be enabled after the game has ended (using the Snapshot Button) to display the cumulative heat map distribution, that is, the most important squares throughout the game. It is only available in the Learning Mode, as it is information that might help the user improve their play.

This functionality exists in an effort to help the user understand positional play. It can be understood as a visualization of the most important parts of the board in the game. By showing this information to the user, they might be able to focus their play on trying to better control these squares, which might help them improve their game.

The Heat Map algorithm works by calculating a score for each square, which takes multiple parameters into account, such as the amount of pieces the square is directly attacked by, the value of those pieces, the value of the piece residing in the square (if any), and more. This information is then stored in a Heat matrix which is used to display a color gradient that represents how valuable each square is for the

game at hand.

5.8.3 Difficulty

The game allows the user to choose from 5 difficulties that can be chosen irrespective of the desired game mode, and they affect how well the engine plays against the user. The available difficulties are: Beginner, Amateur, Intermediate, Hard, and Master.

In order to achieve these different difficulties, the game defines different search parameters that artificially limit the engine's intelligence. The parameters can be seen in Figure 6.

Difficulty	Depth	Time limit (seconds)	Quiescence	Transposition table
Beginner	1	2	False	False
Amateur	3	2	False	False
Intermediate	5	5	True	True
Hard	7	5	True	True
Master	20	10	True	True

Fig. 6: Preset difficulty parameters

After testing multiple difficulty parameters, the final ones shown in Figure 6 were chosen to provide the user with the possibility of playing against an engine version that is close to their level, enabling a more engaging user experience. The parameters can be changed easily, ensuring that, in the event users found particular difficulties to be too easy or too hard, they could be tweaked in a future update.

It must be noted that, as was explained in the Iterative Deepening section, the depth parameter is a maximum limit. When it comes to the Master difficulty, for example, the game will almost never reach a depth of 20, because it will reach the time limit of 10 seconds per move before it can reach said depth level. On the lower difficulties, however, the opposite phenomenon tends to occur, where the depth limit is reached before the time limit, which leads to the engine waiting until the time limit to make its move. This is done to improve the consistency of the User Experience, as previous versions of the game showed that having inconsistent move timings provided a confusing experience for the user.

6 Results

6.1 Engine performance

After the development of the engine, I tested Easy Chess' performance by making it play against engines with a known ELO rating, as that would make it easier for me to determine its approximate rating. At the same time, I tested the engine against a human player with an official competitive rating of 1800 ELO, with the goal of determining whether the engine was suitable for high-level players.

Chess.com's engine [8] was chosen as a rival because it allows the user to specify the desired ELO rating of the computer, which meant that I was able to increase difficulty progressively, and determine an approximate rating for my engine. All the matches were carried out with Easy Chess being set to its maximum difficulty setting (Master). The results of these tests can be found in Figure 7.

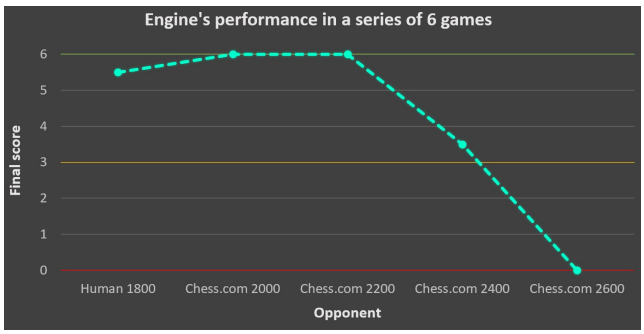

Fig. 7: Engine performance results. Easy Chess wins against all opponents, except for the Chess.com engine with 2600 ELO.

The performance score is an aggregate of the results in the 6 games, with a win awarding 1 point, a draw awarding 0.5 points, and a loss awarding 0 points, for a maximum score of 6.

As can be seen on the line graph, the engine did very well against the human with a rating of 1800, winning 5 games and drawing 1, yielding a final score of 5.5. The draw occurred when the engine had an advantage in an Endgame situation which was insufficient to yield a win. With best play from both sides, the resulting Endgame was a draw, and the human player was able to force that draw after a Threefold Repetition.

As for Chess.com's engine, Easy Chess won all games against the versions with a rating of 2000 and 2200 ELO, proving to be more capable than those engines. My engine started to encounter difficulties against the Chess.com engine with a rating of 2400 ELO but was still able to overall win with a final score of 3 victories, 1 draw, and 2 losses, for a performance rating of 3.5. Against the Chess.com engine with a rating of 2600 ELO, however, my algorithm was beaten thoroughly, with a final score of 0 wins and 6 losses. The full games report table can be found in Appendix 4.

By looking at these results, it seems that the approximate performance of the Easy Chess engine is around the 2400 ELO mark, which greatly exceeds the initial goal of 1400 ELO. To put things in perspective, the 2400 rating corresponds with the International Master title [9], which is the second-highest attainable chess title. While this rating is not entirely accurate, as it would have to be tested against top-level human players, and in more games, it is enough to conclude that the performance of the engine is much higher than the initial goal, and more than sufficient for the purposes of this project.

6.2 Game interface

After numerous playtesting sessions, the UX has been tweaked and improved throughout the development of the project, to the point where I am very satisfied with the final results. Although the most important aspect of the project is the engine's performance, the UX is both functional and interactive, and offers more options that I initially envisioned.

The final game offers all the functionalities I originally envisioned, and additional ones which were added during the development phase. A short demonstration of the final version of the game can be found in the following video:

https://www.youtube.com/watch?v=UkeD40ddjUQ

As for the Heat Map functionality, an example of the resulting heat map during a game can be seen in Figure 8. As you can see in the screenshot, the central squares on the board seem to be the most important according to the algorithm, as well as the D and B files, which are controlled by Black's rooks. Looking at this analysis, the user might notice that the central squares of the board are the most important at the current position, and might therefore adapt its play to better control them. The final snapshot corresponding to this game can be found in Appendix 5.

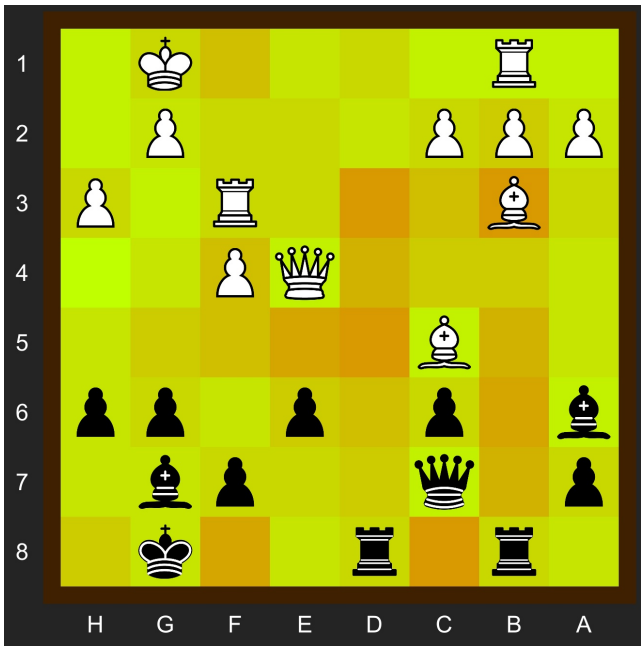

Fig. 8: Heat Map example. Each square has a color representing how important it is within the chess board.

Looking at this analysis, I am happy with the performance of the Heat Map functionality, as it provides with all the insight I was hoping it would produce.

7 Conclusions

After reaching the conclusion of the project, I am very satisfied by the fact that all the main goals have been fulfilled. Not only that, but there have been multiple extra functionalities added to the project, even though they were not initially planned.

As for the engine's performance, it exceeds my initial goals and expectations by a large margin, and is much more capable than would have been strictly necessary.

The actual game itself allows the user to interact with all the functionality that I initially devised, while doing so with an intuitive and interactive UI.

In terms of the development of the project itself, it had its challenges, particularly when it comes to the implementation of the engine's algorithm. The Transposition Table, the Alpha-Beta, and the Move Generator algorithms were all challenging to program, but they ended up performing better than I anticipated.

All in all, the development of this project has been very engaging and interesting from a personal point of view, which has allowed me to maintain a regular development pace. I have grown as a developer, and as a project manager, which will surely help me in future projects.

Acknowledgements

I would like to thank Mr. Joan Camprubí, the chess player with an 1800 ELO rating, who agreed to play many games against my engine to test and improve its performance. He helped uncover bugs and areas of improvement, and it is greatly appreciated.

I also wish to show my appreciation to Mr. Ramón Baldrich and Mr. Xim Cerdà, my 2 tutors for the duration of the project. I am very grateful to both of them for their help.

References

[1] **Appendix 1 contains all the section references for Chessprogramming Wiki**

"Main page," Chessprogramming wiki. [Online]. Available: https://www.chessprogramming.org/Main_Page [Accessed: 20-Jun-2022].

[2] "J. J. S. Layton, "Learn to play chess," US Chess Federation. [Online]. Available: http://www.uschess.org/index.php/Learn-About-Chess/Learn-to-Play-Chess.html#section8 [Accessed: 12-Jun-2022].

[3] "US Chess Federation Ratings Database," US Chess Federation. [Online]. Available: http://www.uschess.org/archive/ratings/ratedist.php [Accessed: 05-Apr-2022].

[4] T. A. Marsland, Computer Chess and Search. Edmonton: Dept. of Computing Science, University of Alberta, 1991.

[5] Chess - Cornell University. [Online]. Available: https://www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/algorithms.html [Accessed: 04-Mar-2022].

[6] "Quiescence," RedHotPawn.com. [Online]. Available: https://www.redhotpawn.com/rival/programming/quiescence.php [Accessed: 04-Mar-2022].

[7] "Pawn Structure in Chess," Chess.com. [Online]. Available: https://www.chess.com/terms/pawn-structure [Accessed: 12-Jun-2022].

[8] "Play vs the Computer," Chess.com. [Online]. Available: https://www.chess.com/play/computer [Accessed: 12-Jun-2022].

[9] "Chess titles," Chess.com. [Online]. Available: https://www.chess.com/terms/chess-titles [Accessed: 12-Jun-2022].

APPENDIX

A.1 Chessprogramming Wiki sections references

- Search Tree [Online]. Available: https://www.chessprogramming.org/index.php?title=Search_Tree [Accessed: 04-Mar-2022].
- Alpha-Beta [Online]. Available: https://www.chessprogramming.org/Alpha-Beta [Accessed: 04-Mar-2022].
- Move ordering [Online]. Available: https://www.chessprogramming.org/Move_Ordering [Accessed: 05-Apr-2022].
- Horizon Effect [Online]. Available: https://www.chessprogramming.org/Horizon_Effect [Accessed: 11-Jun-2022].
- Transposition Table [Online]. Available: https://www.chessprogramming.org/Transposition_Table [Accessed: 05-Apr-2022].
- Evaluation heuristics [Online]. Available: https://www.chessprogramming.org/Evaluation#Basic_Evaluation_Features [Accessed: 05-Apr-2022].
- Board Representation [Online]. Available: https://www.chessprogramming.org/Board_Representation [Accessed: 06-Mar-2022].
- Chess Programming Languages [Online]. Available: https://www.chessprogramming.org/Languages [Accessed: 05-Apr-2022].
- MVV-LVA [Online]. Available: https://www.chessprogramming.org/MVV-LVA [Accessed: 12-Jun-2022].
- Killer Heuristic [Online]. Available: https://www.chessprogramming.org/Killer_Heuristic [Accessed: 12-Jun-2022].
- History Heuristic [Online]. Available: https://www.chessprogramming.org/History_Heuristic [Accessed: 12-Jun-2022].
- Perft Testing [Online]. Available: https://www.chessprogramming.org/Perft [Accessed: 08-Apr-2022].
- Perft Results [Online]. Available: https://www.chessprogramming.org/Perft_Results [Accessed: 08-Apr-2022].
- Piece values [Online]. Available: https://www.chessprogramming.org/Point_Value [Accessed: 12-Jun-2022].
- Negamax [Online]. Available: https://www.chessprogramming.org/Negamax [Accessed: 19-May-2022].
- Iterative Deepening [Online]. Available: https://www.chessprogramming.org/Iterative_Deepening. [Accessed: 12-Jun-2022].

A.2 Chess Engine tutorial series

For those interested in seeing the process of developing a chess engine, I recommend the YouTube series ”Programming a Chess Engine in C”, from Bluefever Software. The series can be found in the following link: https://youtube.com/playlist?list=PLZ1QII7yudbc-Ky058TEaOstZHVbT-2hg.

Even though the series uses the C programming language, a command-line input interface, and the Universal Chess Interface, I used this video series as a reference and guide for my program many times, as it goes over implementation details on all the important aspects of the engine’s development. I consider this series a great starting point for anyone trying to learn about the world of chess programming.

A.3 Perft Testing results

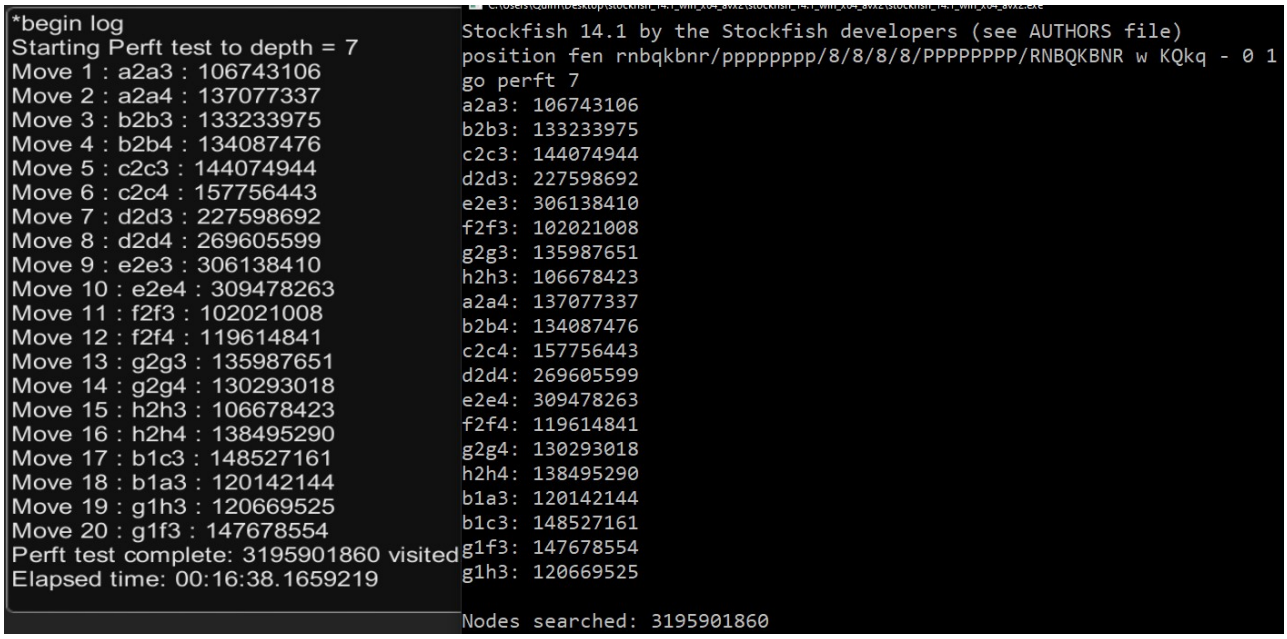

Fig. 9: Output of the Perft Testing algorithm on my program (left-hand side) and the output of the Stockfish engine, one of the best in the world (right-hand side). Even though some moves are generated in a different order, the amount of terminal nodes for each move is exactly the same, meaning the Move Generation algorithm works correctly.

Figure 9 shows that when testing the number of nodes for the same position and with a depth of 7, both algorithms return the same exact number of terminal nodes, a total of 3,195,901,860 nodes. Given these results, we can safely assume that the Move Generator works correctly, and generates the right amount of moves given a certain position.

Perft Testing helped me fix multiple bugs in my Move Generation algorithm, which were mostly related to castling and King Safety. In order to fix these bugs, I compared each of the move's number of nodes with the correct number from the Stockfish engine. When I found a move with an incorrect number, I played it on the board, and kept repeating the process until I found the precise move that was being generated incorrectly.

A.4 Full game results

Game Results	Human 1800	Chess.com 2000	Chess.com 2200	Chess.com 2400	Chess.com 2600
Game 1 White	Win	Win	Win	Win	Loss
Game 2 White	Win	Win	Win	Win	Loss
Game 3 White	Win	Win	Win	Loss	Loss
Game 1 Black	Draw	Win	Win	Win	Loss
Game 2 Black	Win	Win	Win	Loss	Loss
Game 3 Black	Win	Win	Win	Draw	Loss
Record (W-D-L)	5-1-0	6-0-0	6-0-0	3-1-2	0-0-6
Performance	5.5	6	6	3.5	0

Fig. 10: Full game results table

Figure 10 shows the full game report from section 6.1. Easy Chess played 6 games against each of its opponents, alternating between playing as White and Black, for a total of 3 times on each side.

A.5 Heat Map Final Snapshot

Figure 11 shows the final snapshot of the same game that was played to extract the results displayed in section 6.2. Comparing this snapshot to the results shown there, we can see that the central squares are highlighted the most, as they were the most important squares throughout the game. We can also see the traces of the Rooks controlling the B and D files, as we saw in the results section. All in all, I believe that the Snapshot feature is useful to obtain a final summary of the most important squares in the game, and to extract additional information pertaining the reason why the player might have won or lost.

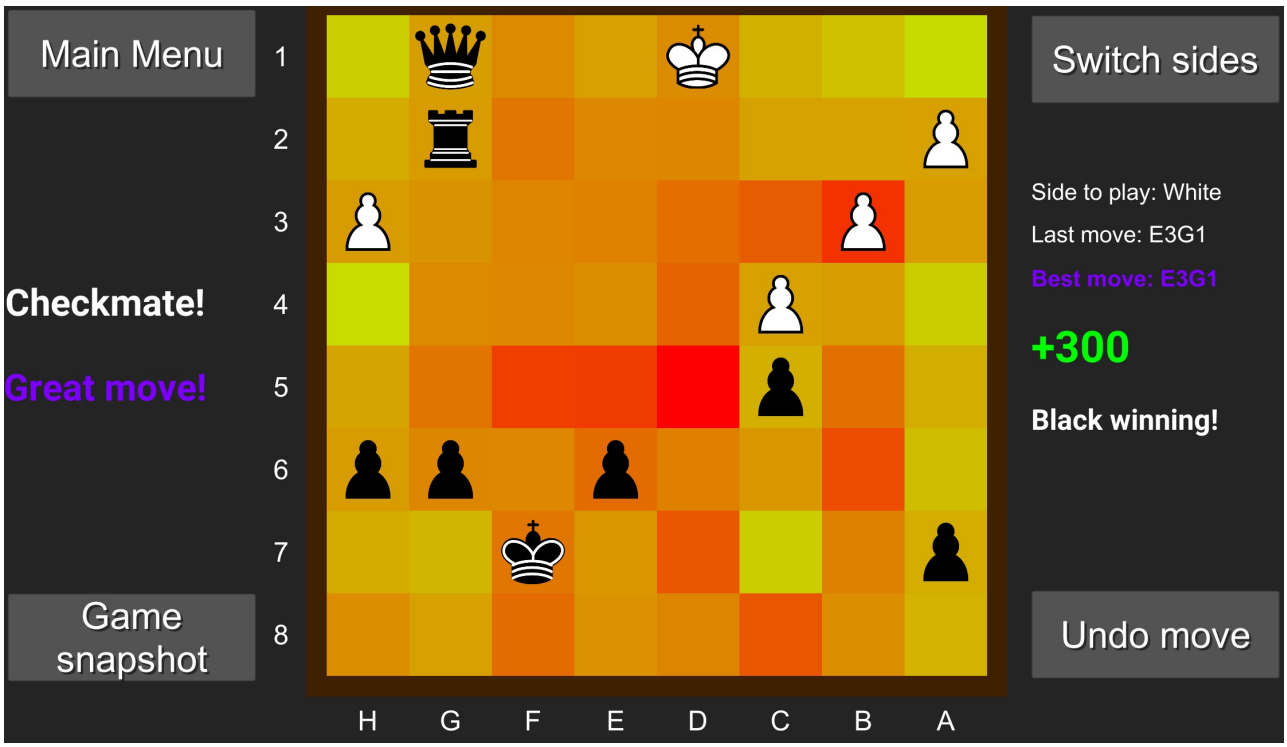

Fig. 11: Final Snapshot example