Minimax Algorithm Applied to Chess Engines

Calderan, Felipe V.
Institute of Science and Technology (ICT)
Federal University Of São Paulo (UNIFESP)
São José dos Campos, Brazil
fvcalderan@unifesp.br

***Abstract*—Minimax algorithm is widely used in games and has been a very important part of Chess Engines. Here it'll be analyzed how different evaluation functions of a chess board affects the performance of Minimax, as well as how the depth of search changes both how well the algorithm plays and how much time it needs to make a move. Finally, it'll be shown how the algorithm deals with mate-in-3 chess puzzles and the results of matches between the simple Minimax algorithm implemented and chess.com's chess engine in the state it's found at the time of this article's writing.**

***Index Terms*—chess, engine, artificial intelligence, Minimax**

I. Introduction

Artificial Intelligence has become a very significant area of study in the modern era. It has found uses in science, marketing, economy and in the gaming industry, to name a few of them.

Before AI was widespread in the gaming industry, single-player games were very different from what they are today. Basically, the game itself had to be single-player in its nature, e.g. patience card game. This means the players were challenged only by their own abilities, the threats that the game environment imposes and, in many cases, luck.

There was no way one could play a game like chess (apart from analyzing positions and playing against oneself). Fortunately, the will to create an artificial opponent for chess has been present since a long time ago, as it'll be discussed in the following section.

Today, top chess engines are virtually unbeatable by any human being in a standard match. However, Human-Computer chess matches still happen, but with the AI being handicapped [1], [2], that is, with one or more pieces missing or with move disadvantage.

Then, why are chess engines being continuously developed to this day, if they are already so strong? There are mainly two reasons for this:

- New AI methods are constantly being tested in games, because games are challenging. One recent example of games being useful in testing bleeding-edge AI technology is OpenAI in DotA. This is a game that runs continuously (there aren't turns), and is expected to be played in team. Program an AI that can thrive in an environment like this isn't an easy task. Also, OpenAI Gym [3] is a great toolkit for developing reinforcement learning algorithms, so it's useful for science outside the world of games.
- The developments in chess AIs also contribute for human improvement in the game, since the engines help humans analyze positions and games, increasing the speed in which players can learn and understand given situations. Magnus Carlsen, actual World Chess Champion said during an interview with Deutsche Welle (DW): "we've known for a long time that computers are better, so the computer never has been an opponent. It's a tool to help me analyze and to help me improve at chess." [4]

Therefore, it's important to continue developing chess engines both for the science of it and for chess. This paper, won't be covering the state of the art of chess engines in details, though. Rather, it'll cover a specific algorithm called Minimax, since it has been very relevant not only in chess, but in many other games too.

This is exactly what will be covered:

- A brief history of chess-playing machines, when Minimax was first introduced to the chess world and what is the state of the art.
- How Minimax algorithm is implemented and how alpha-beta pruning can help its performance.
- How the depth of search interferes in how well a computer plays and how much it has to think before moving.
- The importance of evaluation functions and how they are decisive when the algorithm has a limited depth of search.
- Minimax can perfectly mate-in-x, given enough depth.
- How well the implemented algorithm does against a "real world AI"

II. Brief History and Related Works

In the year 1770, Wolfgang Von Kempelen created a fake chess-playing machine called "The Turk" [5]. This machine was controlled by a human, that typically was a chess master inside its big cabinet. The Turk ended up playing (and defeating) many people, including Napoleon Bonaparte and Benjamin Franklin. Although it was fake, the machine was one of the first documented signs of human interest in attempting to create an artificial chess opponent.

After The Turk, two more famous automata were created: Ajeeb [6] in 1868 and Mephisto [7] in 1876. Ajeeb operated very similarly as The Turk, but it also played checkers. Mephisto, in other hand, didn't hide its chess master inside a box; it was controlled electromagnetically.

The next relevant innovation in Computer Chess was "El Ajedrecista", which is an automaton that was built in 1912

by Leonardo Torres y Quevedo. This is the first documented real chess-playing machine [8]. El Ajedrecista can checkmate a human in the following conditions: the machine should have a rook and a king (playing as white) and the human should only have a king (playing as black). This makes 1912 the first time where a human could play against an artificial opponent, though in a very limited way.

Fast-forwarding to 1948, Norbert Wiener published Cybernetics [9], in which he proposes a way to build a chess engine using Minimax with a depth-limited search and an evaluation function. This is very related to the studies presented in this article.

Shortly after, in 1949, Claude Shannon published "Programming a Computer for Playing Chess", which was one of the first papers discussing computer chess [10]. In this paper, the term Minimax was not coined, but the concept of Minimax is essentially what is presented. In other words: idea of minimizing and maximizing the game score based on board evaluation, where the algorithm foresees the future, by playing perfectly its turn, then the opponent turn, and so on.

Finally, in 1951, Alan Turing developed an algorithm capable of playing a full game of chess [11]. This was so advanced that a computer that could run such program didn't exist. Turing ran the program himself by hand, taking many minutes calculating each move. Although the algorithm wasn't strong, he believed that with enough computer power, it could outplay many human players. This is very relevant because, although the term AI wasn't coined until 1956, Turing's algorithm is basically a fully functional chess AI, since it considered many game play factors such as material value, position value and king safety.

As the years went by, Alpha-beta Minimax was explored by some researchers, such as in [12], where it's presented a mathematical model in which minimax does a good job, giving an end game chess example. At this point in time, mathematical models showing the effectiveness of Minimax were uncommon, therefore this kind of analysis is essential.

Meanwhile, other researchers were looking for alternatives to minimax [13], which was important since eventually Alpha-beta Minimax would be beaten by modern alternatives, not only in chess, but in Go and other games in the same category.

Today, the most modern and powerful chess engines use Monte Carlo Tree Search (MCTS) as the algorithm to play the games. The evaluation functions can be made using reinforcement learning. This method is used by Google's AlphaZero [14], and has proved to be a very good way to create a chess engine, since, although there are controversies, AlphaZero beat Stockfish, which was the previous strongest chess engine available.

It's also worth mentioning that there have been many other proposed ways to generate evaluation functions, such as genetic algorithms.

III. Methods

A. Minimax

Minimax is a decision algorithm that minimizes the loss for a worst case scenario (maximum loss). Its implementation is recursive and in a 2 players game, Minimax always considers that the players will play perfectly, according to its evaluation criteria. Therefore, in theory, it would be possible to generate perfect decisions (leading to perfect play in a game). In reality, it depends.

If the game in which Minimax is being applied is simple enough to generate a branching tree that a modern computer can run through in an acceptable time, then perfect play is, in fact, possible. In this case, no special heuristic is needed, since the evaluation would be the final state of the perfectly played game.

Unfortunately, a game like chess has a big enough tree that it would take a very long time to process. Take the Shannon Number [15]: 10^{120}. This number is generated by estimating that a game of chess will have 40 moves (where in each move there are two half-moves: white's and black's), and in each move a player choose between an average of 30 possible moves. The Minimax time complexity is $O(b^m)$ (since it performs a Depth-First Search (DFS) like procedure), where b is the branching factor and m is the maximum depth of the tree. From the Shannon Number, it's inferable that $b = 30$ and $m = 40 * 2$, therefore the time complexity would be $O(30^{40*2}) \approx O(10^{120})$.

Although there's a way to mitigate this effect by using Alpha-Beta pruning, there's no way a full branching tree of chess is viable, so limited depth is a must and, in consequence, so are heuristics to evaluate a given position. This leads to the building blocks for this chess-playing AI algorithm: A minimax decision tree, with limited depth and a heuristic function.

Minimax's evaluation of a position is given by the function (1), which is described in more detail in [16].

$$eval(s) = \begin{cases} heuristic\_eval(s) & \text{if done} \\ max_{a \in moves(s)} MM(play(s,a)) & \text{if max turn} \\ min_{a \in moves(s)} MM(play(s,a)) & \text{if min turn} \end{cases} \quad (1)$$

where s is the current game state, $heuristic\_eval(s)$ is the heuristic evaluation of the game state, $moves(s)$ is the set of available moves in the current state, $MM()$ is the Minimax function call and $play(s, a)$ means "play a from the s state".

It's also worth mentioning that "if done" can be broken into 2 separate cases:

- The depth limit has been reached
- The game has ended

This of course means that, the smaller the depth, the less likely the algorithm is to make the best move, especially considering situations like mid-game chess.

Pseudo-code 1 is a python-styled pseudo-code for Minimax, which is a simple algorithm to implement.

```
def minimax(board, depth, player):
  if depth == 0 or game_over():
    return heuristic_eval(board)

  movelist = [i for i in legal_moves]

  if player == MAX:
    best_eval = -infinity
    for move in movelist:
      play_move(board, move)
      eval = minimax(board, depth-1, MIN)
      undo_move(board)
      best_eval = max(eval, best_eval)
    return best_eval
  else:
    best_eval = infinity
    for move in movelist:
      play_move(board, move)
      eval = minimax(board, depth-1, MAX)
      undo_move(board)
      best_eval = min(eval, best_eval)
    return best_eval
```

Pseudo-code 1. Minimax

B. Alpha-Beta pruning

Alpha-Beta pruning in Minimax is a very straightforward optimization that can increase performance dramatically, as shown in the Experimental Analysis section. Basically, Alpha-Beta removes the necessity for the algorithm to run through the whole tree, and even better, returns the exact same result that standard Minimax would. So, considering a chess engine, there's essentially no reason to not optimize Minimax with Alpha-Beta pruning or other similar method.

As stated before, Minimax searches in a DFS fashion. With Alpha-Beta, for each node that isn't a leaf, 2 values are typically stored:

- **Alpha:** Maximum value found in MAX nodes descendants up to the current state.
- **Beta:** Minimum value found in MIN nodes descendants up to the current state.

When executing the algorithm, there are two possible scenarios for the pruning to occur: either when the node is a MAX node and $\alpha(node) \geq \beta(parent(node))$ or when the node is a MIN node and $\beta(node) \leq \alpha(parent(node))$.

Pseudo-code 2 shows how this is integrated inside Minimax.

```
def minimax(board, depth, player, a, b):
  if depth == 0 or game_over():
    return heuristic_eval(board)

  movelist = [i for i in legal_moves]

  if player == MAX:
    best_eval = -infinity
    for move in movelist:
      play_move(board, move)
      eval = minimax(board, depth-1, MIN, a, b)
      undo_move(board)
      best_eval = max(eval, best_eval)
      if (a := max(a, eval)) >= b: break
    return best_eval
  else:
    best_eval = infinity
    for move in movelist:
      play_move(board, move)
      eval = minimax(board, depth-1, MAX, a, b)
      undo_move(board)
      best_eval = min(eval, best_eval)
      if (b := min(b, eval)) <= a: break
    return best_eval
```

Pseudo-code 2. Minimax with Alpha-Beta pruning

Notice how there were barely any changes to the algorithm. Alpha-Beta introduced two new parameters to the Minimax function and two new lines of pseudo-code. Fig. 1 illustrates the procedure in action.

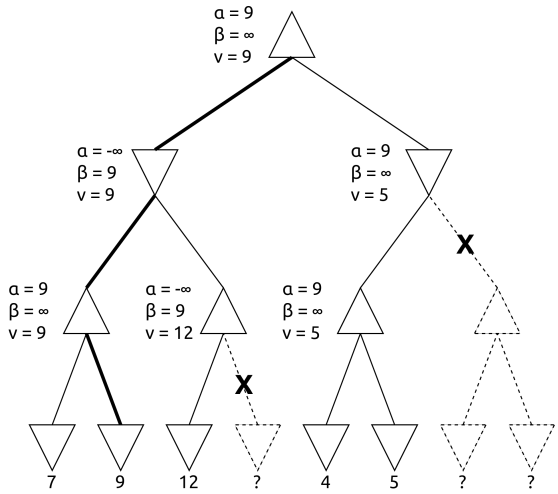

Fig. 1. Minimax with Alpha-Beta pruning tree.

C. Heuristics

Since Minimax will run with a limited depth, there must be a way to evaluate a given position mid-game. The heuristics explored here are related to chess in their given format, but they can be generalized to be applied in similar games. Also, heuristics can be set by using reinforcement learning, genetic algorithm and other techniques. For an in-depth view on these sophisticated methods, the reader can refer to [14] and [17], respectively, as these won't be covered in this paper.

1) Point Value: The simplest chess heuristic is to sum the point values, which are values given to the pieces relative to their strength, e.g. a pawn is worth 1 point, knight and bishop 3 points each, rook 5 points, queen 9 points and the king is invaluable.

Although this is a very limited evaluation function, it does work for very obvious scenarios, which include the execution of a good trade, avoidance of a bad trade or pointless material loss, assuming there are no further tricks involved, that is, there are no underlying strategies where giving up a piece, actually promotes an advantage.

Fig. 2 shows a situation where Minimax with depth 4 is guaranteed to detect the best move (***Qb7***) by considering only the point values.

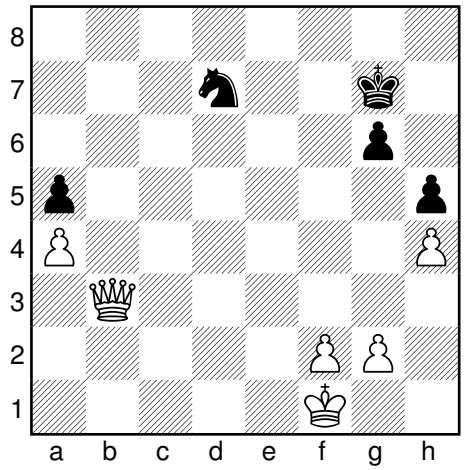

Fig. 2. White plays and wins a knight

This evaluation function is, however, incapable of doing a very essential task: checkmating the opponent. This happens because checkmate isn't as simple as taking the king, but rather a series of movements that eventually generate a position where the king is being attacked, but has nowhere to go.

If this algorithm were to do the same task as "El Ajedrecista", the game would probably end in a draw (75-move rule)[1], due to the AI moving randomly across the board without any strategy. A checkmate is possible only by luck.

Fortunately, this is solvable. It's sufficient to reward a checkmate position with a high score in magnitude (positive for white and negative for black) and have a big enough depth so that the algorithm can detect such position. The harder the mate is to perform, the bigger should be the depth. For very difficult mates, the best option would be to implement specific code for end-game scenarios.

2) Piece-Square Table: Another possible heuristic is Piece-Square Table. This approach also sums the values of the pieces, but the main difference is that besides the Point Value, each piece earns additional points depending on where they are in the board.

This can generate very interesting interactions. For instance, take these matrices that represent the Piece-Square Tables for the white rook and king, respectively:

$$R = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 10 & 10 & 10 & 10 & 10 & 10 & 5 \\ -5 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\ -5 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\ -5 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\ -5 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\ -5 & 0 & 0 & 0 & 0 & 0 & 0 & -5 \\ 0 & 0 & 0 & 5 & 5 & 0 & 0 & 0 \end{bmatrix}$$

[1]Not to be confused with the 50-move rule, which has to be claimed

$$K = \begin{bmatrix} -30 & -40 & -40 & -50 & -50 & -40 & -40 & -30 \\ -30 & -40 & -40 & -50 & -50 & -40 & -40 & -30 \\ -30 & -40 & -40 & -50 & -50 & -40 & -40 & -30 \\ -30 & -40 & -40 & -50 & -50 & -40 & -40 & -30 \\ -20 & -30 & -30 & -40 & -40 & -30 & -30 & -20 \\ -10 & -20 & -20 & -20 & -20 & -20 & -20 & -10 \\ 20 & 20 & 0 & 0 & 0 & 0 & 20 & 20 \\ 20 & 30 & 10 & 0 & 0 & 10 & 30 & 20 \end{bmatrix}$$

The combination of these tables encourages castling, especially king-side, because the king leaves a position that is worth 0 points to go to one that is worth 10 or 30 points, while the rook unlocks possible interesting squares (*a4, a5*). All of this in one single move!

Other interesting things that this kind of heuristic enables is encouraging pieces development, center domination, specific strategies like bishops in Fianchetto and so on. This gets even stronger if the tables are dynamic, in other words, if they change according to what is more suitable depending on the pawn structures, the amount of remaining pieces, and many other factors. Take the following matrix representing a Piece-Square Table for a King in the end-game:

$$K_e = \begin{bmatrix} -50 & -40 & -30 & -20 & -20 & -30 & -40 & -50 \\ -30 & -20 & -10 & 0 & 0 & -10 & -20 & -30 \\ -30 & -10 & 20 & 30 & 30 & 20 & -10 & -30 \\ -30 & -10 & 30 & 40 & 40 & 30 & -10 & -30 \\ -30 & -10 & 30 & 40 & 40 & 30 & -10 & -30 \\ -30 & -10 & 20 & 30 & 30 & 20 & -10 & -30 \\ -30 & -30 & 0 & 0 & 0 & 0 & -30 & -30 \\ -50 & -30 & -30 & -30 & -30 & -30 & -30 & -5 \end{bmatrix}$$

In the end-game, there are fewer threats to protect the king from and it should be time to start pushing the pawns forward for a promotion. Although the king is invaluable, it is also useful as a piece, whether keeping the other king distant, or pursuing weak pawns. This means that having an active king is a wise decision, so there is no point in keeping it at the corner of the board.

Unlike the Point Value method, this evaluation method also provides strategies for the early and mid-game, due to the incentives to place the pieces in certain positions. A compilation of Piece-Square Tables can be found in [18].

3) The faster the checkmate is, the better: If the game has come to a position where a checkmate is viable, the faster this mate happens, the more rewarding it should be to the algorithm to avoid pointless game stalling. Function (2) shows how this is implemented mathematically.

$$heuristic\_eval(s) = \begin{cases} \infty - (max\_depth - depth) & \text{if MAX turn} \wedge \text{1-0} \\ -\infty + (max\_depth - depth) & \text{if MIN turn} \wedge \text{0-1} \end{cases} \quad (2)$$

where max_depth is the initial depth passed to Minimax.

IV. Experimental Analysis

In this section, it'll be described the environment in which the experiments were ran, how the tests were configured and measured, the results and a brief discussion about what was obtained.

A. Computational Environment

All the tests were executed in the same computational environment, which is a conventional low-spec modern laptop (Acer Aspire 3 A315-53-365Q) with increased RAM.

1) Hardware:

- **CPU**: Intel® Core™ i3-8130U[2] 2.2GHz Turbo: 3.4GHz
- **GPU**: Intel® UHD Graphics
- **RAM**: 12GB DDR4 2400MHz
- **HDD**: 1TB 5400rpm

2) Software:

- **OS**: Arch Linux 5.8.2-arch1-1 x86_64 GNU/Linux
- **Python**: 3.8.5 (default, Jul 27 2020 08:42:51)
- **GCC**: 10.1.0 on Linux
- **python-chess library**: 0.31.3

B. Analysis Criteria

For convenience, from now on Minimax without Alpha-Beta pruning will be called Minimax and Minimax with Alpha-Beta pruning will be called Alpha-Beta.

1) Time comparison between Minimax and Minimax with Alpha-Beta: 3 entire chess games between two identical AIs were played in each category, those being: Minimax (depth 1, 2, 3, 4) and Alpha-Beta (depth 1, 2, 3, 4), with a total of 24 games played.

For each category, it was taken the average of the computing time (time span in which the computer was thinking) of white's and black's moves between the 3 games e.g. In the Alpha-Beta with depth 3 category, the following results were obtained:

```
white's Average move time in game 1: 0.611708s
black's Average move time in game 1: 0.964356s
white's Average move time in game 2: 0.295526s
black's Average move time in game 2: 0.311997s
white's Average move time in game 3: 0.388936s
black's Average move time in game 3: 0.715847s
```

Given this small dataset, taking the average of the values, it would be obtained that the average move time for this category is $0.548062s$.

It's worth mentioning that the half-moves (white's or black's individual moves) were timed using python's *timeit* built-in library and that there were no noticeable differences between white's and black's average move times. There were games where white was faster, games where black was faster and games where they were even, but the average move times across all the games in the categories were essentially the same.

[2]only one core was used per game

2) Games played between the implemented AI against itself, varying the depth and evaluation function method: 10 entire games between Alpha-Beta (except for the single Random AI entry) AIs were played in each event, those being:

- Random AI vs Point-Value Depth 2
- Point Value Depth 2 vs Piece-Square Table Depth 2
- Point Value Depth 3 vs Piece-Square Table Depth 2
- Point Value Depth 3 vs Piece-Square Table Depth 3
- Point Value Depth 4 vs Piece-Square Table Depth 3
- Point Value Depth 4 vs Piece-Square Table Depth 4.

This makes for a total of 60 games played. In each event, each of the 2 AIs played 5 matches as white and 5 matches as black.

The analysis criterion consists of verifying how many points the AIs got in each event, where a win counts as 1 point, a draw as 1/2 points and a loss as 0 points. There is no time limit.

3) Implemented AI vs chess.com's AI: The main objective here was to see how far the implemented Alpha-Beta AI using Piece-Square Table with depth 4 could get playing against chess.com's "play vs AI" [19] computer, therefore the rules were: for each event, whichever engine is the first one to have 2 points more than the opponent wins. The first event starts with chess.com's AI playing in difficulty 2 (the difficulty ranges from 1 to 10) and for every event that the implemented AI wins, chess.com's AI difficulty level is increased by 1. When the implemented AI finally loses, the experiment is finished.

The point system is exactly the same as in the games played between the implemented AI against itself: 1 for a win, 1/2 for a draw and 0 for a loss.

The two AIs alternate sides each match, this way it's possible to give them the opportunity to play both as white and as black.

4) Mate-in-3 Puzzles: Mate-in-3 Puzzles were played by the Alpha-Beta AI using Point Value with Depth 5 as the side that is supposed to checkmate against Stockfish 11, which should be checkmated successfully.

The reason to use Point Value instead of Piece-Square Table is because in this situation, the latter simply has no use, since Minimax with depth 5 should, by itself, see the checkmate.

It's also convenient to remind the reader that "The faster the checkmate is, the better" method is active, so it's guaranteed that the mate will occur in the given move limit, given that the algorithm was correctly implemented.

In total, 6 puzzles were tested, each one 5 consecutive times, making a total of 30 games played.

This last experiment is more of a sanity-check to whether Alpha-Beta is doing its job properly. The criterion is simple: if Alpha-Beta can checkmate Stockfish in the correct number of moves, it's approved, otherwise, it's rejected. Of course, it should do it every time that it's tested for a given puzzle.

Figs. 3,4,5,6,7 and 8 show all the puzzles given to the AI.

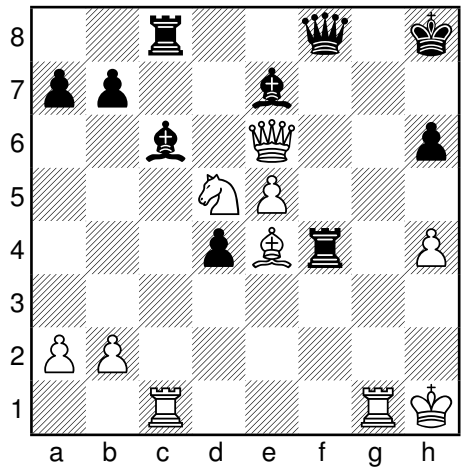

Fig. 3. White plays - Mate in 3

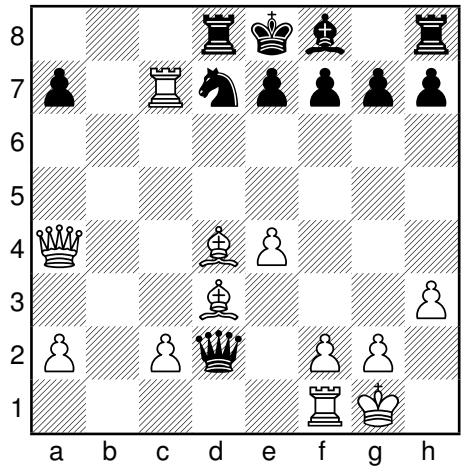

Fig. 4. White plays - Mate in 3

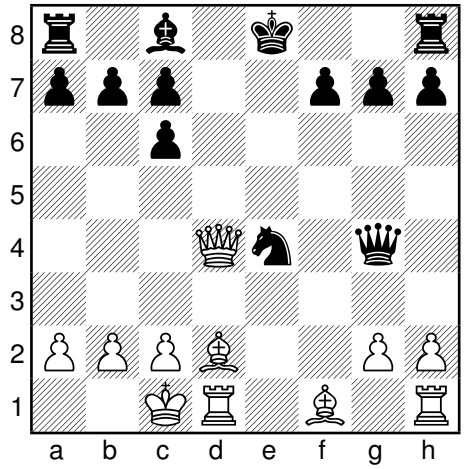

Fig. 5. White plays - Mate in 3

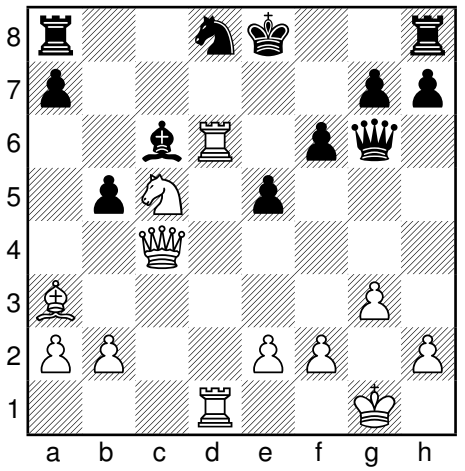

Fig. 6. White plays - Mate in 3

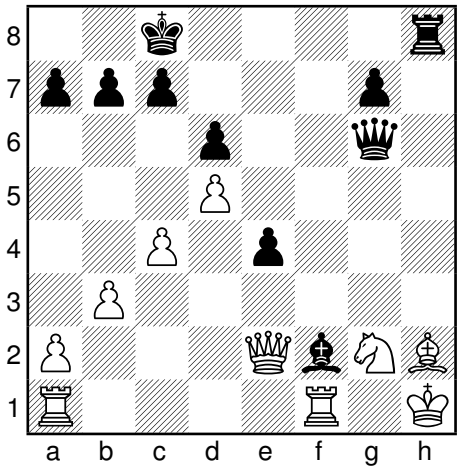

Fig. 7. Black plays - Mate in 3

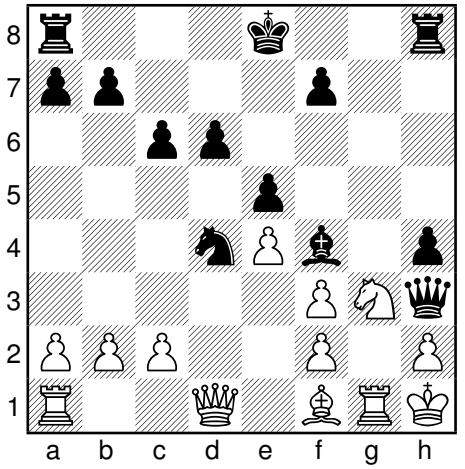

Fig. 8. Black plays - Mate in 3

C. Results and Discussion

Here follows the results obtained from the studies made.

1) Time comparison between Minimax and Minimax with Alpha-Beta: Since the time complexity of Minimax algorithm is $O(b^m)$, it tends to grow exponentially as the depth (m) is increased. Alpha-Beta mitigates this effect by reducing the number of branches that will need to be explored. Table I and Fig 9 show the growth of the time taken for the computer to calculate the best move using both: Minimax-only approach and Minimax with Alpha-Beta.

Depth	Minimax (s)	Alpha-Beta (s)
1	0.005761	0.005323
2	0.213530	0.080171
3	6.056677	0.548062
4	45.194300	3.846207

TABLE I
AVERAGE MOVE TIME X DEPTH

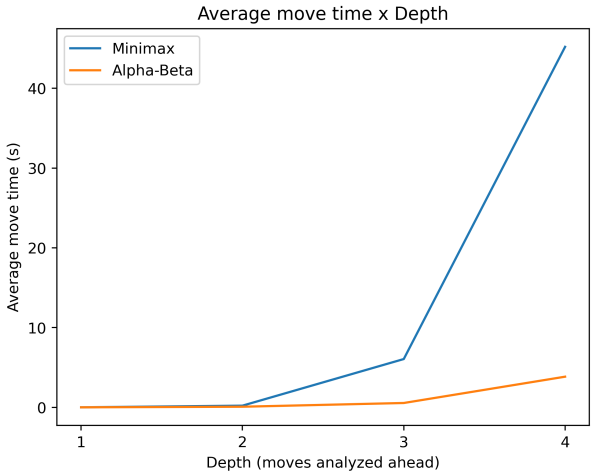

Fig. 9. Average move time x Depth

It's possible to see how effective Alpha-Beta pruning can be. Alpha-Beta with depth 4 was over 91% faster than Minimax with depth 4. In fact, it was 36% faster than Minimax with depth 3. Therefore, by optimizing with Alpha-Beta, it was possible to analyze the game 1 move further with the same quality of outcome and in less time.

Needless to say, it's possible to reduce this time significantly by using more sophisticated pruning methods and heuristics. Also, this was all done using only a single core. Of course, a multi-core approach would be faster, but the algorithm complexity (in terms of how difficulty the concepts behind are) for a utilizing multi-core increases substantially.

2) Games played between the implemented AI against itself, varying the depth and evaluation function method: The results are presented in Table II.

Player	Depth	White	Black
Random AI	*	0	0.5
Point Value	2	4.5	5
Point Value	2	1	0
Piece-Square Table	2	5	4
Point Value	3	3.5	4
Piece-Square Table	2	1	1.5
Point Value	3	0.5	0.5
Piece-Square Table	3	4.5	4.5
Point Value	4	2	1.5
Piece-Square Table	3	3.5	3
Point Value	4	1.5	1.5
Piece-Square Table	4	3.5	3.5

*: Does not apply, since the algorithm plays randomly

TABLE II
RESULTS OF THE MATCHES BETWEEN IMPLEMENTED AIS

These are some interesting results. The Point Value with depth 2 stalemated the Random AI a single time. Fig 10 shows the final position.

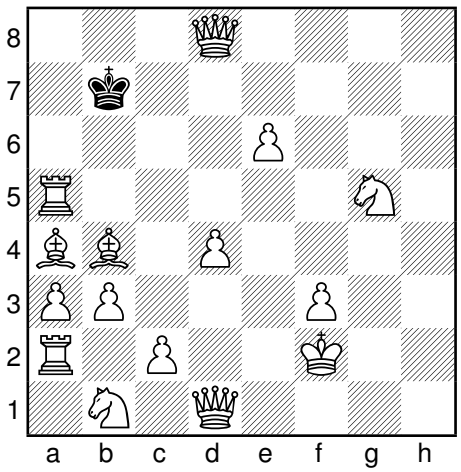

Fig. 10. Minimax with Point Value stalemates Random AI

The final move was ***54. Qd8***. The suggested moves by Stockfish 11 were ***54. Bc5 Kc7 55. Ra7#***, which need at least depth 3 to be accomplished consistently, explaining why the stalemate happened.

For the other games, a pattern may seem to emerge, where Point Value loses to Piece-Square Table if using the same depth, and wins if using a depth 1 bigger than Piece-Square Table. This however, turns out to be false, since Piece-Square Table with depth 3 beat Point Value with depth 4, by a good margin.

This shows that, with limited depth, strategy (or heuristics for an engine) is a very important concept for strong play. Event if the computer could see many moves into the future, if those moves have no meaning, this won't result in a stronger game play.

An analysis can also be made comparing the evaluation functions of the Piece-Square Table method and Stockfish 11. This analysis will be done using one of the games played between Point Value with depth 4 (as white) and Piece-Square Table, also with depth 4 (as black).

Fig. 11 is a graph that shows how both of these evaluate a game, board state by board state. Since the implemented algorithm uses a different evaluation scale than Stockfish, it was mapped to the same range. Also, consider 30 points (positive or negative) "mate-in-x" for Stockfish.

It's also worth mentioning that a positive number indicates that white is better, whereas a negative one shows that black is better.

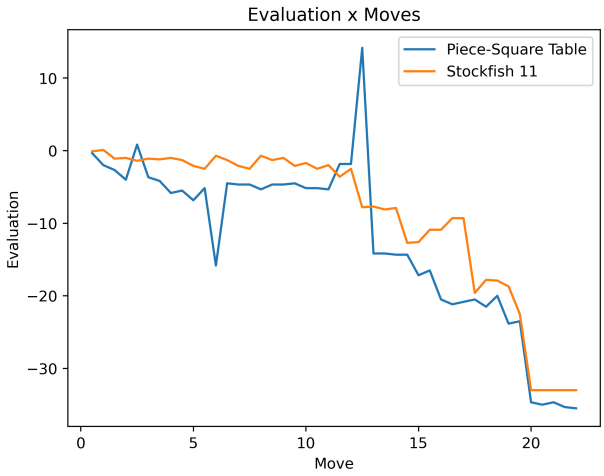

Fig. 11. Evaluation x Moves

Observing Fig. 11, it's possible to see a big spike in Piece-Square Table's evaluation. This shows a big difference between how it and Stockfish evaluate the board: its evaluation is static, it refers to a given moment. Stockfish evaluation also considers the future.

However, there's a big drop around the spike for both the implemented algorithm and for Stockfish, the difference is that Stockfish saw it first. Fig. 12 shows the move that caused the spike.

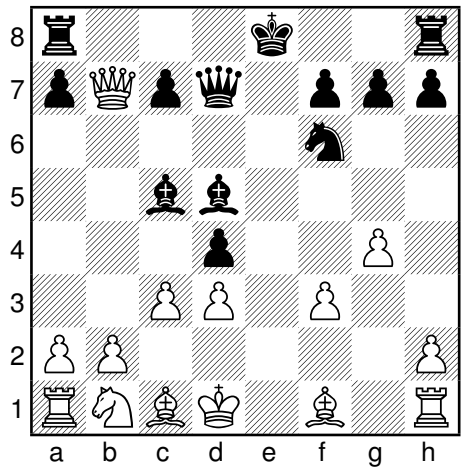

Fig. 12. White played ***13. Qxa8+***

White's ***13. Qxa8+*** was actually the move suggested by Stockfish, since ***13. Qa6 Bxf3+ 14. Kc2 Bxh1*** causes white to lose a pawn, lose the ability to castle and lose a Rook. If white tries to defend the king with ***Be2*** instead of ***Kc2***, it's even worse, because then what happens is ***14. Be2 Bxe2+ 15. Kxe2 Qxg4+ 16. Kd2 Qg2+ 17. Kd1 Qxh1+ 18. Kc2***.

Other than ***Qxa8+*** and ***Qxa6***, there are no other acceptable moves for white's queen. This situation could have been avoided, had Minimax algorithm have a depth bigger than 4. Basically, white wouldn't have put its queen in this situation in the first place.

After this, the game went very poorly for white and this is visible in Fig 11. The interesting fact to observe is that, although the Piece-Square Table algorithm is a much less sophisticated evaluation method, both evaluation methods showed a more or less stable game evaluation in the first half of the graph and then a huge increase in the value magnitude afterwards (starting from the queen situation).

Another point to be made is how Piece-Square Table has a much better structure than Point Value. In Fig. 12, white has barely developed its pieces, in fact, only the queen and some pawns are active. Meanwhile, black has its bishops in a strong spot in the board, a somewhat developed knight and is preparing to castle (which does happen some moves after).This kind of structure can be observed in most of the games with enough depth and will be shown in more details in the following experiments (against chess.com's AI).

3) Implemented AI vs chess.com's AI: The results of the matches between the implemented AI (Minimax with Alpha-Beta pruning, depth 4 and using Piece-Square Table heuristic) and chess.com's AI can be seen in Table III. The reader should keep in mind the rules already stated in the previous section.

Player	Difficulty	White	Black
Implemented AI	Depth 4	1	1
chess.com's AI	Difficulty 2	0	0
Implemented AI	Depth 4	1	1
chess.com's AI	Difficulty 3	0	0
Implemented AI	Depth 4	1	1
chess.com's AI	Difficulty 4	0	0
Implemented AI	Depth 4	1	0
chess.com's AI	Difficulty 5	2	1

TABLE III
RESULTS OF THE MATCHES BETWEEN THE IMPLEMENTED AI AND CHESS.COM'S AI

It's clear that the implemented AI dominated the games up to chess.com's difficulty 4, but difficulty 5 was too much for it to deal with. It was still able to get a game, which was impressive.

In the matches against the difficulty 2, it's possible to see how much better Alpha-Beta develops its pieces compared to chess.com's AI. Fig 13 and 14 show the board state in move 15 of the games.

Notice how in Fig. 13 Alpha-Beta has built a solid defense: it had already castled and developed bishops and knights, while black side is very weak, with already no castling possibility and two pawns and one knight in disadvantage.

Looking at Fig. 14, black is dominating the board with a very strong center aggression, is ready to castle, haven't lost a single pawn and already have one extra knight (and 2 pawns).

Considering that the queens are about to be exchanged and the white king, again, has no castling possibility, the game has pretty much ended at this point.

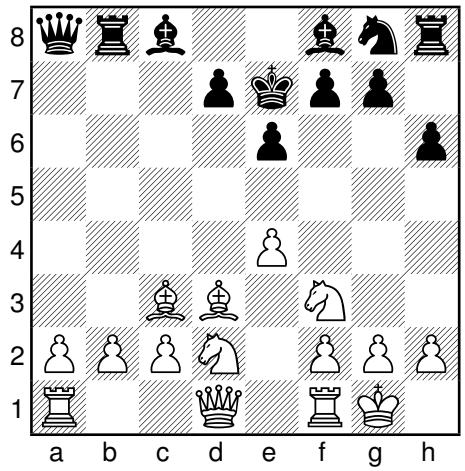

Fig. 13. Game 1 - Alpha-Beta (white) vs Difficulty 2 (black) - Move 15

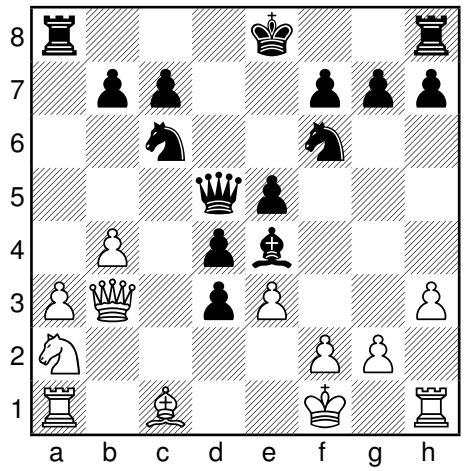

Fig. 14. Game 2 - Difficulty 2 (white) vs Alpha-Beta (black) - Move 15

For the matches with chess.com's AI in difficulty 3, in the first game, Alpha-Beta managed to place a pawn in the 7th rank, which was fiercely defended until the end of the game, even though it ended up never promoting. Fig. 15 shows the moment when the pawn was placed in the 7th rank and Fig. 16 shows the ending position.

In the second game, nothing remarkable happened. By the move 15, Stockfish evaluated the board giving a small advantage for Alpha-Beta. After move 15, this advantage started to snowball, since chess.com's AI started losing many pieces one after another.

Of course, as expected, increasing the difficulty lead to more structured and competitive games in general and this will become visible in later games with difficulty 4 and 5, when Alpha-Beta finally couldn't keep up.

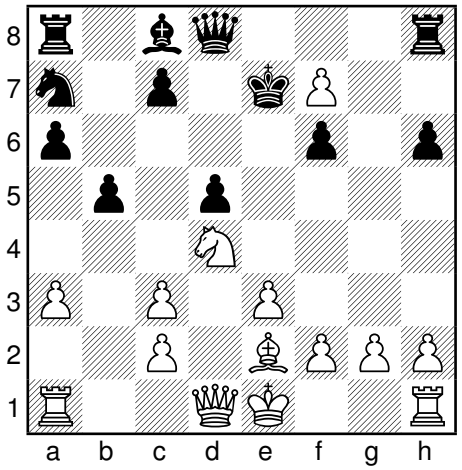

Fig. 15. Game 3 - Alpha-Beta (white) vs Difficulty 3 (black) - Move 14

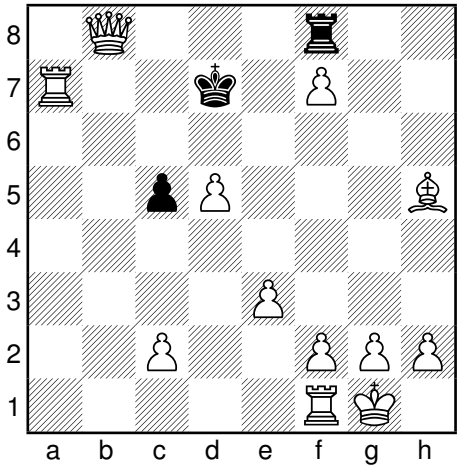

Fig. 16. Game 3 - Alpha-Beta (white) vs Difficulty 3 (black) - Move 33

The games in difficulty 4 were much more interesting. The first game was played in a very defensive manner by both sides. Until move 14, only a single exchange had happened: white's and black's knights. The game was very balanced at this point as it's visible in Fig. 17. Stockfish 11's evaluation was +1.1, confirming that the game was, in fact, very competitive.

By move 20, Stockfish rated the board as 0.0 and if this wasn't interesting enough by itself, by move 50 the board was rated as 0.0 again! Since both parts wouldn't ask for a draw (for experimentation purposes), the game kept going and eventually Alpha-Beta started gaining the upper-hand, leading to a victory.

The second game, where Alpha-Beta played as black, was very open. In move 16 many pieces have already been exchanged, including the queens. Unfortunately, for chess.com's AI, it was already in a bad spot at this point, having material disadvantage over Alpha-Beta (missing a knight and a pawn). This lead to its defeat, which was an interesting looking checkmate, composed of two pawns, a bishop and a rook, as seen in Fig. 18.

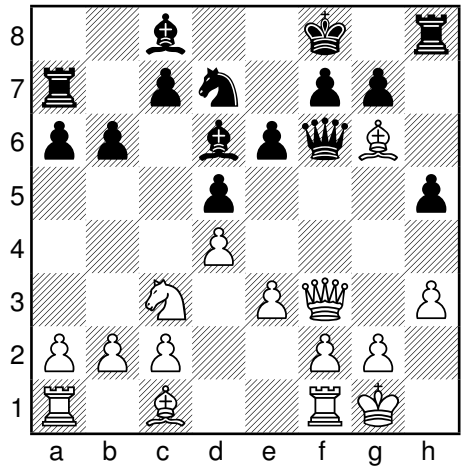

Fig. 17. Game 5 - Alpha-Beta (white) vs Difficulty 4 (black) - Move 14

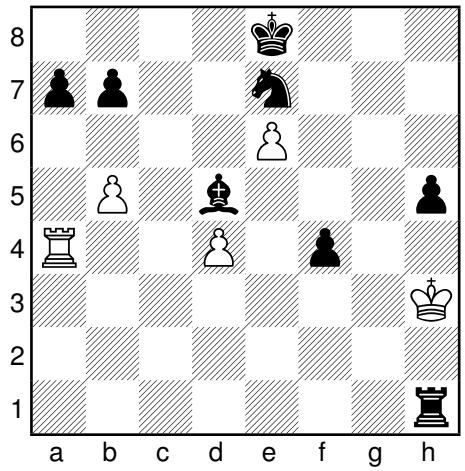

Fig. 18. Game 6 - Difficulty 4 (white) vs Alpha-Beta (black) - Move 33

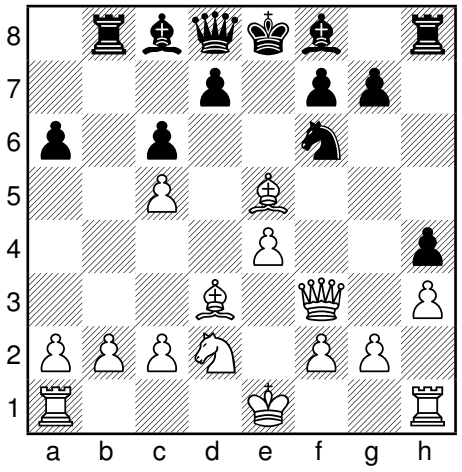

Fig. 19. Game 7 - Alpha-Beta (white) vs Difficulty 5 (black) - Move 14

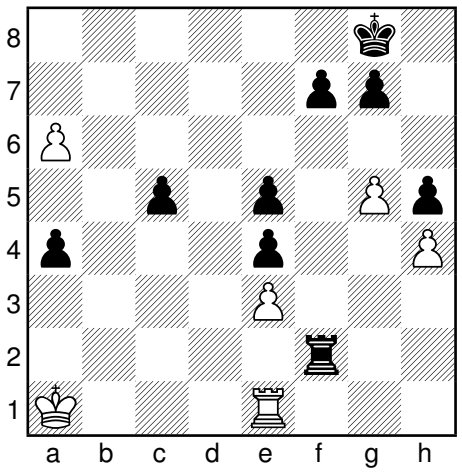

Fig. 20. Game 8 - Difficulty 5 (white) vs Alpha-Beta (black) - Move 40

Now, the games where chess.com's AI played in difficulty 5. The first game (and the only game Alpha-Beta won) had a very questionable opening strategy by chess.com's AI (playing as black).

The opening was *Zukertort Opening: Queen's Gambit Invitation* or more simply put: *Queen's Pawn Game* (***1. Nf3 e6 2. d4 c5 3. dxc5***). Typically, the next move by black would be either ***3. Nf6*** (suggested by Stockfish) or ***3. Bxc5*** as seen in [20] and [21], but in this game ***Qa5+*** was played, which was followed by ***4. Bd2 Qd8 5. Bc3 h6***. This is not a good choice, since there is a momentum loss that comes with it.

After that, another questionable move was made by black. In the position described in 20, instead of ***15. d6***, black gave up a rook playing ***15. Rb5***. This lead to a big material loss. Now black had a bishop for a rook, and two less pawns. These bad choices cause chess.com's AI to eventually lose the match.

In the second game, it was Alpha-Beta (playing as black) that made a big mistake. In the position pictured in Fig. 22, Alpha-Beta played ***40. Re2***, giving the rook **and the game** for free. The recommended move by Stockfish 11 was ***40. Rd2***, which makes sense and is a way better option.

This was unexpected, so it was worth investigating. The first step was to see if this was reproducible and it was. Minimax by it self analyzed 43952 complete game lines (depth 4), while Alpha-Beta analyzed 7023. As expected, both return the same move.

Since the behavior was strange, the next natural step was to see if this move was, in fact, what the algorithm believed to be the best. Looking at Table IV it's possible to see that it is (following its criteria) the best option.

Move	Eval	Move	Eval
Re2	165	Rh2	180
Rf1	170	Rd2	180
Rb2	170	Kh7	185
Ra2	170	Kh8	185
a3	170	g6	190
Rf4	170	f5	190
c4	170	Kf8	195
Rc2	175	Rf3	195
Rf5	175	f6	200
Rg5	175	Rf6	400

TABLE IV
POSSIBLE FIRST MOVES AND THEIR OUTCOME

Because Alpha-Beta was playing as black, the lower its evaluation is, the better the move is considered.

With this analysis, it's possible to conclude that the cause of this behavior was the lack of depth to see the best move or at least a better one.

In the next game, chess.com's AI (playing as black) crushed Alpha-Beta with a powerful attack in the king's side. Fig. 22 shows how the attack started and Fig. 23 pictures the checkmate, which happened in move 40.

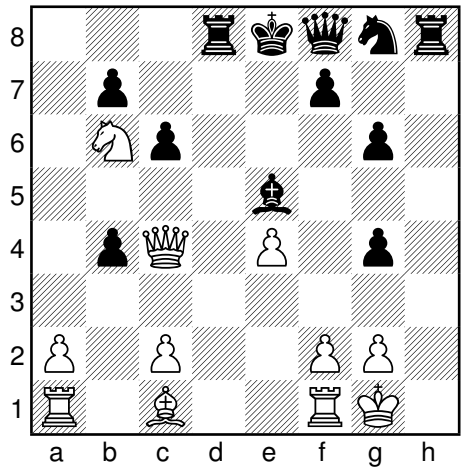

Fig. 21. Game 9 - Alpha-Beta (white) vs Difficulty 5 (black) - Move 24

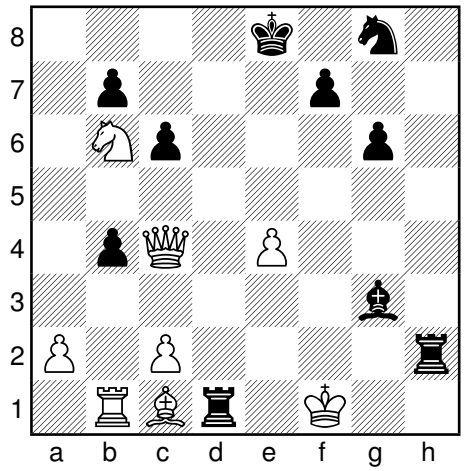

Fig. 22. Game 9 - Alpha-Beta (white) vs Difficulty 5 (black) - Move 40

The final game had an interesting board structure for both sides, as pictured in Fig. 23. Alpha-Beta was threatening ***Bh3*** (move that eventually happened) and put a lot of pressure throughout the game. Unfortunately it had a bad exchange, losing a bishop in the process, helping its ultimate defeat.

The pgn of the final game is: ***1.d4 Nc6 2.Nf3 Nf6 3.h3 e6 4.c4 Bb4+ 5.Nbd2 O-O 6.Qc2 d6 7.Qb3 Ne4 8.e3 Bxd2+ 9.Nxd2 Na5 10.Qa4 Nxd2 11.Bxd2 Nc6 12.Bd3 Qf6 13.Bc3 Ne7 14.O-O e5 15.Rfe1 Qg5 16.Qb3 Rd8 17.Qc2 Bxh3 18.Bxh7+ Kh8 19.Be4 Bc8 20.Bd3 Bh3 21.f3 exd4 22.f4 Qxg2+ 23.Qxg2 Bxg2 24.exd4 Nd5 25.Bd2 Bf3 26.Kf2 b5 27.b3 bxc4 28.bxc4 Nxf4 29.Bxf4 Bc6 30.Be3 Rab8 31.Reb1 Rxb1 32.Rxb1 Kg8 33.Bf5 Re8 34.a3 d5 35.c5 g6 36.Rg1 Rb8 37.Bxg6 Rb2+ 38.Kf3 Kf8 39.Bh6+ Kg8 40.Bf5+ Kh8 41.Bg7+ Kg8 42.Bf6+ Kf8 43.Rh1 Rb3+ 44.Kf2 Rb2+ 45.Kg3 Rb3+ 46.Kf4 Rh3 47.Bxh3 a6 48.Bg4 a5 49.Bf5 Bd7 50.a4 Kg8 51.Rh8# 1-0***

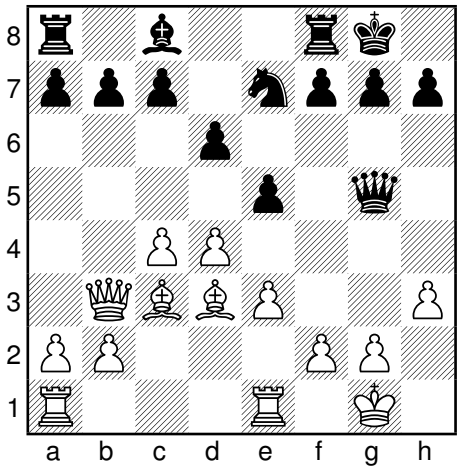

Fig. 23. Game 10 - Difficulty 5 (white) vs Alpha-Beta (black) - Move 16

With a better evaluation function (one that takes into account pawn-structures, number of pieces defending important positions or other pieces, attack strategies and other additions), a opening book, end-game book and a multi-threaded system to increase the depth without causing the algorithm to run very slow, it's possible to beat a few more levels of difficulty.

For what was implemented in this study, being able to beat Difficulty 4 and even get one match in Difficulty 5 is a very nice result, since the algorithm is very simple and uses only very basic principles to play.

4) Mate-in-3 Puzzles: As expected, Minimax with depth 5 was enough to solve all the mate-in-3 puzzles consistently, since it can see the checkmate in its search tree.

Here are the solutions provided by Minimax to the puzzles described (the opponent is Stockfish 11, but this is meaningless, since the checkmates are guaranteed).

1. Rg8+ Qxg8 2. Qxh6+ Qh7 3. Qxh7# 1-0
1. Qxd7+ Rxd7 2. Rc8+ Rd8 3. Bb5# 1-0
1. Qd8+ Kxd8 2. Bg5+ Ke8 3. Rd8# 1-0
1. Rxd8+ Rxd8 2. Qe6+ Kf8 3. Nd7# 1-0
1... Rxh2+ 2. Kxh2 Qg3+ 3. Kh1 Qh3# 0-1
1... Qxh2+ 2. Kxh2 hxg3+ 3. Kg2 Rh2# 0-1

This holds true for mate-in-x. Given enough depth, Minimax will be able to perfectly checkmate every time. This also reflects what was discussed in the *METHODS* section: if a game is simple enough to the point where the computer can generate the complete tree of movements in an acceptable time, perfect play is viable. Mate-in-x puzzles are one of these "simple enough games" for a low x.

V. Conclusions

It took around 180 years since the initial documented interest in creating an artificial chess opponent until a true chess artificial intelligence, that could play a full chess game, was developed.

This paper discussed about the path that led to Minimax and how in the latest years it has become less present in new engines, since new techniques like MCTS were developed and applied to chess. Nevertheless, Minimax is still a powerful tool in decision making and is worth studying. This is specially true if the implemented algorithm uses Alpha-Beta pruning, since Stockfish, which is one of the strongest chess engines, uses this technique.

Alpha-Beta pruning reduces significantly the amount of time required to process the next best move. Comparing Alpha-Beta Minimax with its naive counterpart using depth 4 showed a improvement of over 90% in time consumption. Since it adds basically no complexity to the code (in terms of how difficult the implementation is), while giving the same results, there's no reason not to do it in a chess engine.

Heuristics are fundamental when dealing with scenarios where a full game search tree isn't possible, which is the case of a standard chess game. Therefore, choosing the most adequate heuristic is a decisive factor on how strong a chess engine will be. It's also worth mentioning that, contrary to what was implemented in this paper, heuristics do not need to be static. They can be selected and modified on demand, depending on the situation.

There are multiple ways to search for the best heuristics, but the two most relevant today are domain knowledge and reinforcement learning, the latter requires a huge amount of computer power, while the former depends solely on how much human knowledge about the game is put into it.

Like shown in one of the experiments, in a very limited depth scenario, the evaluation function is more important than having more depth and a inferior evaluation function.

As it was tested, even a basic heuristic is already able to play acceptable games. Alpha-Beta with depth 4 against chess.com's AI in difficulties 4 and 5 lead to very structured games, with interesting positions that are worth analyzing.

Those games also have shown the limitations that come with a small depth or insufficient evaluation function, like the free rook move by Alpha-Beta or even the questionable opening by chess.com's AI.

Finally, it was shown that in an environment where Minimax has enough depth to see perfect play (mate-in-x puzzles), it will consistently do it no matter how non-intuitive it might seem to a human player. In the history of chess, there were very sophisticated forced checkmates in a big number of moves and with enough depth (same number as the number of required half-moves), Minimax would do all of them without problems.

References

[1] https://en.chessbase.com/post/komodo-9-odds-matches-against-gms
[2] https://www.chess.com/news/view/smerdon-beats-komodo-5-1-with-knight-odds
[3] Greg Brockman and Vicki Cheung and Ludwig Pettersson and Jonas Schneider and John Schulman and Jie Tang and Wojciech Zaremba (2016), "OpenAI Gym", arXiv:1606.01540 [cs.LG]
[4] https://www.dw.com/en/world-chess-champion-magnus-carlsen-the-computer-never-has-been-an-opponent/a-19186058
[5] Bradley Ewart (1980). "Chess, man vs. machine". A S Barnes Co. ISBN 0-498-02167-X.
[6] Schaeffer, Jonathan (1997). "One jump ahead". Springer. pp. 90. ISBN 0-387-94930-5. Retrieved 2009-03-10. "ajeeb chess."
[7] Sunnucks, Anne (1976). "The Encyclopaedia of Chess". London: Hale. p. 314. ISBN 0-7091-4697-3.
[8] Williams, Andrew (2017). "History of Digital Games: Developments in Art, Design and Interaction". CRC Press. ISBN 9781317503811.
[9] https://www.livinginternet.com/i/ii_wiener.htm
[10] Claude Shannon (1949). "Programming a Computer for Playing Chess", The Computer History Museum
[11] https://www.history.com/news/in-1950-alan-turing-created-a-chess-computer-program-that-prefigured-a-i
[12] Beal, D. F. (1982), "Benefits of Minimax Search" In: Advances in Computer Chess, Page 17-24, Elsevier BV, https://doi.org/10.1016/B978-0-08-026898-9.50005-X
[13] Harris L.R. (1983) "The heuristic search: An alternative to the alpha—beta minimax procedure." In: Frey P.W. (eds) Chess Skill in Man and Machine. Springer, New York, NY. https://doi.org/10.1007/978-1-4612-5515-4_7
[14] David Silver and Thomas Hubert and Julian Schrittwieser and Ioannis Antonoglou and Matthew Lai and Arthur Guez and Marc Lanctot and Laurent Sifre and Dharshan Kumaran and Thore Graepel and Timothy Lillicrap and Karen Simonyan and Demis Hassabis (2017), "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm", arXiv:1712.01815 [cs.AI]
[15] http://mathematics.chessdom.com/shannon-number
[16] Russell, S. J., Norvig, P., Davis, E. (2010). "Artificial intelligence: a modern approach". 3rd ed. Upper Saddle River, NJ: Prentice Hall.
[17] https://github.com/HariSurayagari/Computer-Chess-using-Genetic-Algorithm
[18] https://www.chessprogramming.org/Simplified_Evaluation_Function
[19] https://www.chess.com/play/computer
[20] https://www.chessgames.com/perl/chessgame?gid=1945827
[21] https://www.chessgames.com/perl/chessgame?gid=1167852