Project 5 Report:

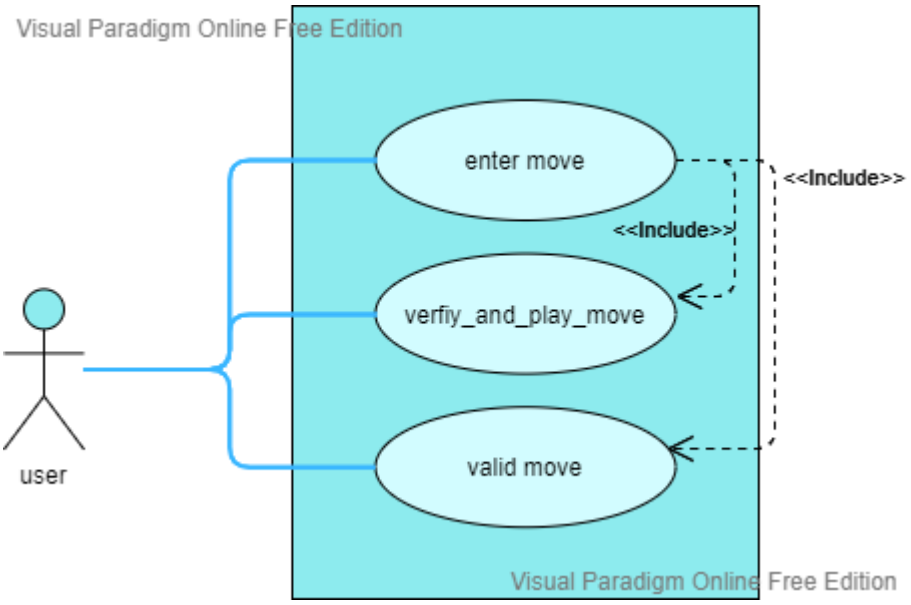
An Intelligent Chess Player using an Alpha-Beta Depth-First algorithm (designing & implementing at least 2 heuristic functions).

إسم الطالب	رقم الطالب	المستوى الدراسي	القسم / الشعبة
محمد عادل محمد حسني محمد	202000776	**الثالث**	**نظم معلومات**
محمد عادل محمد محمود الممشاوي	202000778	**الثالث**	**نظم معلومات**
احمد ممتاز سليم سعودي	202000089	**الثالث**	**نظم معلومات**
نور الدين علي محمد الحنوني	202001001	**الثالث**	**نظم معلومات**
فتحي احمد فتحي محمد	202000645	**الثالث**	**نظم معلومات**
مازن احمد عبد الفتاح عبد الله	202000704	**الثالث**	**نظم معلومات**

1. Introduction and Overview:

Build an artificial Chess player, that can play games against a human opponent. Chess is a two-player strategy board game played on a chessboard, a checkered game-board with 64 squares arranged in an 8×8 grid. The game is played by millions of people worldwide. Chess is believed to be derived from the Indian game chaturanga sometime before the 7th century. Chess reached Europe by the 9th century, due to the Umayyad conquest of Hispania. The pieces assumed their current powers in Spain in the late 15th century with the introduction of "Mad Queen Chess"; the modern rules were standardized in the 19th century. Play does not involve hidden information. Each player begins with 16 pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. Each of the six pieces types moves differently, with the most powerful being the queen and the least powerful the pawn. The objective is to checkmate the opponent's king by placing it under an inescapable threat of capture. To this end, a player's pieces are used to attack and capture the opponent's pieces, while supporting each other. During the game, play typically involves making exchanges of one piece for an opponent's similar piece, but also finding and engineering opportunities to trade advantageously, or to get a better position. In addition to checkmate, a player wins the game if the opponent resigns, or (in a timed game) runs out of time. There are also several ways that a game can end in a draw.

2.Main function:



Description:

Enter move: it's make player move the pieces of chess

Verify_and_play_move: it's check if the turn for player or not

Valid move: it check if the player's move right or not

3.Similar Application:

1.Chess.com -url: https://www.chess.com/play/online

2.Sprarks.com -url: https://www.sparkchess.com/

3.365chess.com -url: https://www.365chess.com/play_computer_online.php

4.chess-url:https://play.google.com/store/apps/details?id=com.pirinel.chess&hl=en&gl=US&pli=1

5.chess -url: https://www.mathsisfun.com/games/chess.html

6.Simplechess -url: https://www.chessanytime.com/play-vs-computer.html

4.Details of the algorithm:

Alpha-Beta:

What is Alpha-beta:

Alpha Beta Pruning *is a method that optimizes the minmax algorithm. The number of states to be visited by the minimax algorithm are exponential, which shoots up the time complexity. Some of the branches of the decision tree are useless, and the same result can be achieved if they were never visited. Therefore, Alpha Beta Pruning cuts off these useless branches and, in best-case, cuts back the exponent to half.*

Alpha Beta Pruning relieved its name because it uses two parameters called alpha and beta to make the decision of pruning branches,

–> Alpha is used and updated only by the Maximizer and it represents the maximum value found so far. The initial value is set to -∞. The value of alpha is passed down to children node, but the actual node values are passed up while backtracking.

–> Beta is used and updated only by the Minimizer and it represents the minimum value found. The initial value is ∞. Again, the value of beta is passed down to the children node, but actual node values are used to backtrack.

It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, Go, etc.).

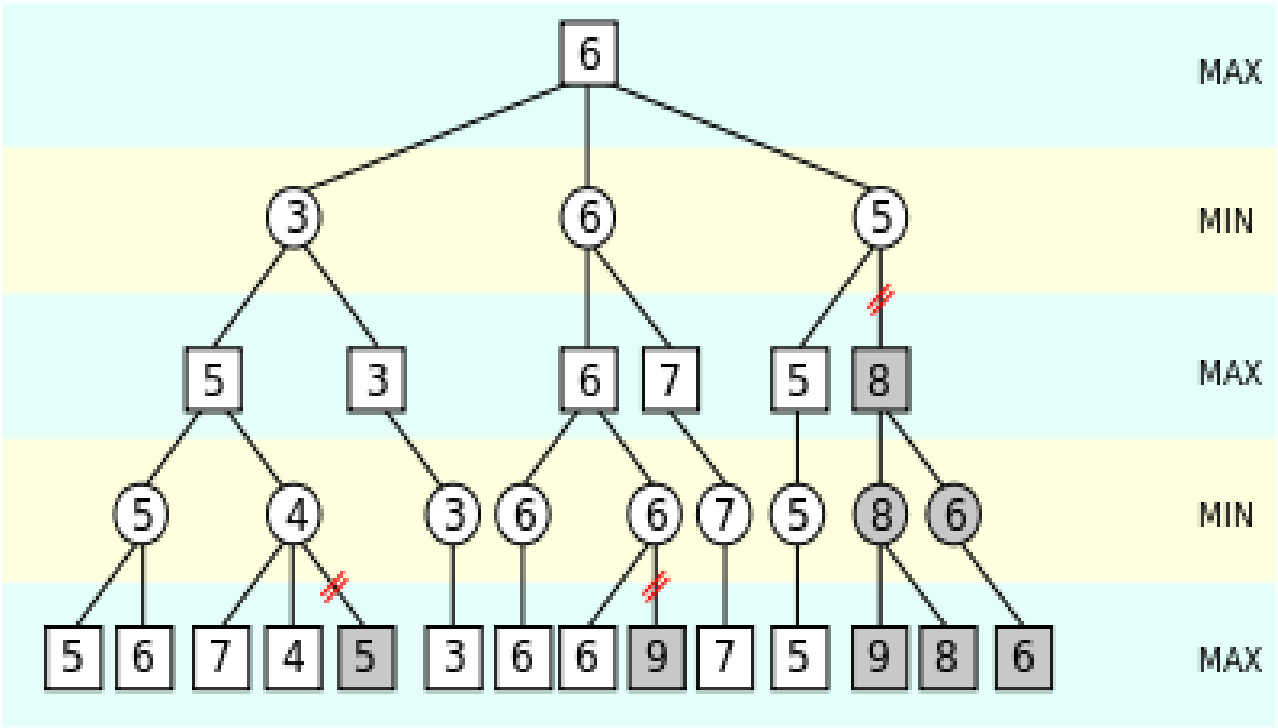
It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further.

When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

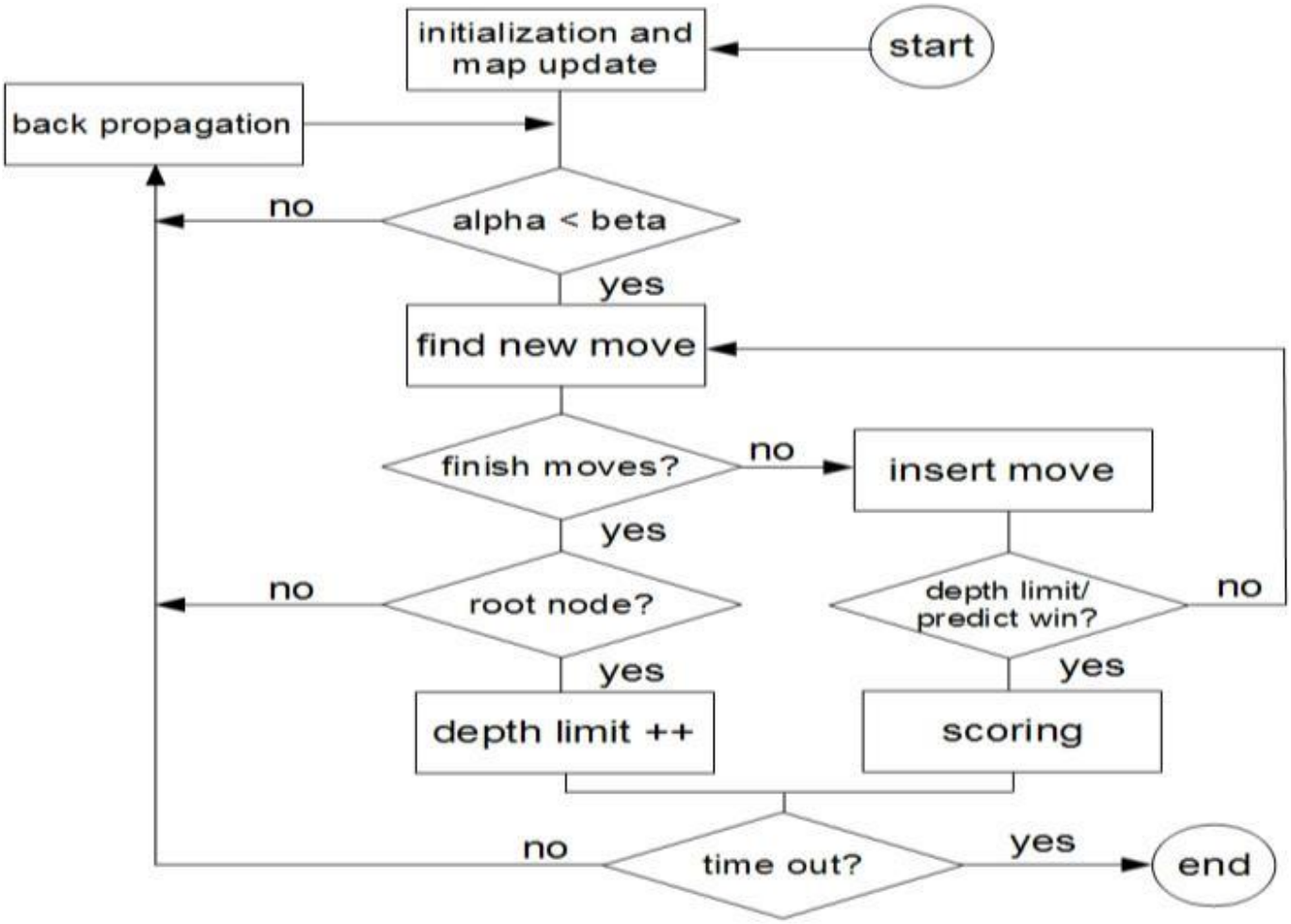
Disadvantages of\Known Problem:

So using Alpha-Beta pruning has a number of advantages to it in terms of space/time complexity gains over the original minimax algorithm. So you are probably wondering if this is the best that can be done. In fact it may not be. In addition it does not solve all of the problems associated with the original minimax algorithm. Below is a list of some disadvantages and some suggestions for better ways to achieve the goals of choosing the best move.

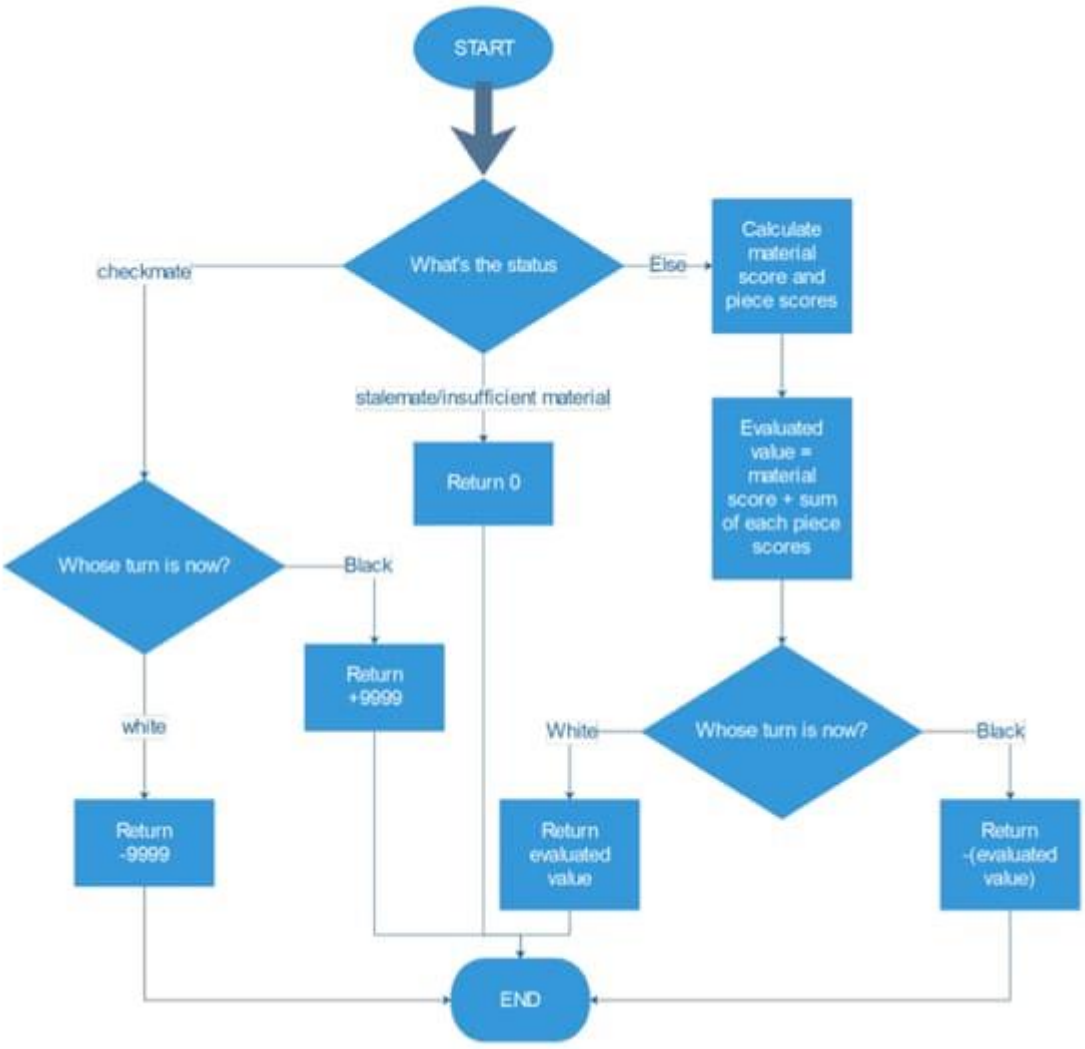
- Evaluations of the utility of a node are usually not exact but crude estimates of the value of a position and as a result large errors could be associated with them.
- We assume that the player will always choose the best possible move for his situation. It is possible that one could choose a poor move if the evaluation function is not that good.
- It is in most cases not feasible to search the entire game tree, a depth limit needs to be set. A notable example is Go which has a branching factor of 360! even with Alpha - Beta Pruning one can not look ahead more than 3 or 4 moves in the game tree.
- Alpha-Beta is designed to select a good move but it also calculates the values of all legal moves. A better method maybe to use what is called the *utility of a node expansion*. In this way a good search algorithm could select a node that had a high utility to expand (these will hopefully lead to better moves). This could lead to a faster decision by searching through a smaller decision space. A extension on those abilities would be the use of another technique called *goal-directed reasoning*. This technique focuses on having a certain goal in mind like capturing the queen in chess. So far no one has successfully combined these techniques into a fully functional system.



Block Diagram:



Flow chart Diagram:



5.An initial literature review of Academic publications for chess ai:

chess is not a game. Chess is a well-defined form of computation. You may not be able to work out the answers, but in theory, there must be a solution, a right procedure in any position—John von Neumann

There are various schools of thought in chess composition, each of which placing different emphasis on the complexity of problems. Chess studies originally became popular in the 19th century. They are a notoriously difficult kind of puzzle, involving detailed calculations and tactical motifs. As such, they provide a good benchmark for AI studies.

We choose to analyse how chess engines respond to Plaskett's Puzzle, one of the most well-known endgame studies in history. The endgame is the final phase of a chess game. There are many features of Plaskett's Puzzle that make it particularly hard—not just the depth required (15 moves) but also the use of underpromotion (a particularly counterintuitive kind of move in chess) and subtle tactical strategies. Though the puzzle was initially created by a Dutch composer, it achieved notoriety in 1987 when English grandmaster Jim Plaskett posed the problem at a chess tournament, stumping all players except Mikhail Tal.

The advent of powerful computer chess engines enables us to revisit chess studies where the highest level of tactics and accuracy are required. We evaluate the performance of Stockfish 14 [1] and Leela Chess Zero (LCZero) [2] on Plaskett's Puzzle. We discuss the implications of their performance for both end users and the artificial intelligence community as a whole. We find that Stockfish solves the puzzle with much greater efficiency than LCZero. Our work suggests that building a chess engine with a broad and efficient search may still be the most robust approach.

Chess engine research has implications for problems requiring large-scale tree search. LCZero and its predecessor, AlphaZero [3], use neural networks to guide a Monte Carlo tree search. Researchers have since used this

idea to construct procedures for chemical synthesis [4] and optimize the dynamics of quantum systems [5]. Stockfish heavily relies on alpha-beta search [6], which has been used to verify neural networks' robustness to adversarial attacks [7].

Artificial general intelligence (AGI) is the ability of an algorithm to achieve human intelligence in a multitude of tasks. Ref. [8] famously claimed that the rules of chess could be determined from empirical observation, a central tenet of artificial general intelligence. Historical records provide insight into human performance on Plaskett's Puzzle. By comparing this with machine performance, we can better assess the state of current progress on the long road to building AGI. We further discuss which algorithms possess greater potential in the field of AGI by reviewing their ability to generalize to different domains.

On the theoretical side, we describe a central tool for solving dynamic stochastic programming problems: the Bellman equation. This framework, together with the use of deep neural networks to model the value and policy functions, provides a solution for maximising the probability of winning the chess game. In doing so, the algorithm offers an optimal path of play. Given the enormous number of possible paths of play (the Shannon number), the use of search methods and the depth of the algorithm become important computational aspects. It is here that high-level chess studies provide an important mechanism for testing the ability of a given AI algorithm.

The rest of the paper is outlined as follows. The next subsection provides historical perspective on Plaskett's Puzzle. **Section 2** provides background material on current AI algorithms used in chess, namely, Stockfish 14 and LCZero. **Section 3** describes our software configuration and experimental approach. **Section 4** provides our detailed analysis of Plaskett's Puzzle using these chess engines. Finally, **Section 5** concludes with directions for future research. In particular, we discuss the implications of our work for human–AI interaction in chess [9] and AI development.

Plaskett's Study

The story of Plaskett's Puzzle dates to a Brussels tournament in 1987. The study was originally composed by Gijs van Breukelen in 1970. In 1987, it famously stumped multiple super grandmasters—players at the highest level in chess—when presented by James Plaskett in a tournament press room. It was finally solved that day by legendary attacking player Mikhail Tal, who figured it out during a break at the park [10].

The highly inventive puzzle involves multiple underpromotions and was originally designed to be a checkmate in 14. There is a mistake in the original puzzle whereby Black can escape checkmate, though White is still winning in the final position. This mistake can be corrected by moving the Black knight on g5 to h8. Harold van der Heijden's famous *Endgame Study Database* (which contains over $58{,}000$ studies) proposes another corrected version where he moves the Black knight from g5 to e5.

Refs. [11,12,13] pioneered the development of AI algorithms for chess. The Shannon number, 10^{120}, provides a lower bound on the total number of possible games, making chess a daunting computational challenge. A major advance over pure look-ahead search methods was the use of deep neural networks to approximate the value and policy functions. Then, the self-play of the algorithms allowed for quick iterative solution paths. See work by [14] for further discussion.

The dynamic programming method breaks the decision problem into smaller subproblems. Bellman's principle of optimality describes how to do this:

Bellman Principle of Optimality: An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

[15]

Backwards induction identifies what action would be most optimal at the last node in the decision tree (i.e., checkmate). Using this information, one can then determine what to do at the second-to-last time of decision. This process continues backwards until one has determined the best action for every possible situation (i.e., solving the Bellman equation).

2.1. Q-Values

The optimal sequential decision problem is solved by calculating the values of the *Q*-matrix, denoted by $\mathrm{Q}(\mathrm{s},\mathrm{a})$ for state *s* and action *a*. One iterative process for finding these values is known as *Q*-learning [16], which can be converted into a simulation algorithm as shown by [17]. The *Q*-value matrix describes the value of performing action *a* (a chess move) in our current state *s* (the chess board position) and then acting optimally henceforth. The current optimal value and policy function are as follows:

$$\mathrm{V}(\mathrm{s})=\max_{\mathrm{a}}\mathrm{Q}(\mathrm{s},\mathrm{a})=\mathrm{Q}(\mathrm{s},\mathrm{a}\star(\mathrm{s}))$$

(1)

$$a\star(s)=\text{argmax}_a Q(s,a).$$

(2)

For example, chess engines such as LCZero simply take the probability of winning as the objective function. Hence, at each stage, $V(s)$ measures the probability of winning. This is typically reported as a pawn advantage, since the pawn is the most basic unit of material in chess. Advantages based on the arrangement of the pieces are heuristically factored into this metric as well. The greater a side's pawn advantage, the greater their chance of winning. Ref. [18] provides an equation for converting between pawn advantage and win probability.

The Bellman equation for *Q*-values (assuming instantaneous utility $u(s,a)$ and a time-inhomogeneous *Q* matrix) is the constraint

$$Q(s,a)=u(s,a)+\sum_{s\star\in S} P(s\star||s,a)\max_a Q(s\star,a).$$

(3)

Here, $P(s\star||s,a)$ denotes the transition matrix of states and describes the probability of moving to a new state $s\star$ given the current state *s* and action *a*. The new state $s\star$ is the board position after the current player has played action *a* and the opponent has responded. Bellman's optimality principle is therefore simply describing the constraint for optimal play as one in which the current value is a sum over all future paths of the probabilistically weighted optimal future values of the next state.

Taking the maximum value of $Q(s,a)$ over the current action *a* yields

$$V(s)=\max_a\{u(s,a)+\sum_{s\star\in S} P(s\star||s,a)V(s\star)\}\text{where}V(s\star)=\max_a Q(s\star,a).$$

(4)

Here, $u(s,a)$ is an instantaneous utility obtained from action *a* in state *s*. For all nonterminal board positions, $u(s,a)$ is zero. At the end of the game, $u(s,a)$ is -1 after a loss, 0 after a draw, and 1 after a win.

The Artificial Intelligence algorithms developed for human play utilize many different kinds of principles. While it is complex to discuss what each engine uses for its functionality and working, we know that a few popular engines like Alpha zero make use of neural networks, deep learning, and neural net-like automation. Leela Chess Zero utilizes an open-source implementation of AlphaZero, which learns chess through self-play games and deep reinforcement learning.

Nowadays, modern chess engines are so well-developed that they will not drop a single game to human players. Even the current reigning, defending world champion failed to beat the modern best chess engine even once in a span of 100 games. The world champion has a FIDE rating of over 2800 across all formats. The contest usually takes place in a classical time format. The match was against Stockfish 9.

It has a rating of 3438 (the engine ratings are not FIDE ratings, but the player pool for engines is much stronger than for humans, so theoretically a FIDE rating for Stockfish 9 would be even higher). Check out the reference section for more details on this topic.

The results of the two modern clashes of chess engines are as follows as referenced Wikipedia.

- 2017 — AlphaZero, a neural net-based digital automaton, beats Stockfish 28–0, with 72 draws, in a 100-game match.
- 2019 — Leela Chess Zero (LCZero v0.21.1-nT40.T8.610) defeats Stockfish 19050918 in a 100-game match 53.5 to 46.5 for the TCEC season 15 title.

These modern results with the help of chess engines and neural networks and deep learning-based chess networks taking over the chess world by storm is a huge sign of the potential of greater and enormous possibilities. Let us dwell a bit more into the potential influence of Artificial Intelligence in the Universe of Chess.

Artificial Intelligence has influenced the way in which chess games are played at the top level. Most of the Grandmasters and Super Grandmasters (Rated at a FIDE above 2700) utilize these modern Artificial Intelligence

chess engines to analyze their games as well as the games of their competitors. There is a complete turnaround in the way in which chess games are now played.

The basic opening theories and other analytical concepts are thoroughly analyzed. In classical formats of chess, you will usually see these high-level players make about 10–15 of the first moves from previously analyzed games or the top engine recommendations.

The quality of the top-level games has also drastically improved because of the help of these engines. It is almost impossible to rate or compete a modern player against a legendary player of the older decades due to the fact of enormous improvement with the help of these chess engines.

While some may oppose that these chess engines have impacted the game of chess in a negative manner because it is more about theory rather than actual practice and play as compared to the older ages. Others argue that this influence of AI on chess is an overall drastic improvement and further advancements are yet to be made for challenging the modern players.

And ultimately making them much better with both opening theory and other tricks to pack up their sleeves. There is still room for human error. These errors can be exploited by players to gain advantages and have interesting games of attacking chess.

Reference: https://www.mdpi.com/1099-4300/24/4/550/htm, https://towardsdatascience.com/ai-in-chess-the-evolution-of-artificial-intelligence-in-chess-engines-a3a9e230ed50

(Chess AI: Competing Paradigms for Machine Intelligence),(AI In Chess: The Evolution of Artificial Intelligence In Chess Engines)

5.1An initial literature review of Academic publications for algorthim:

The alogorhim is very important for make chess ai move ,and human for many years try to improve the algorthim to be more efficient and faster to beat human. How do you get a computer to play chess? A simple way would be to program it to make random moves. A harder way, of course, requires finding out which moves are actually good. That is, which moves, if there are any, will lead to checkmate the quickest?

Theoretically we could take a given board and exhaustively search every move from that position until the end of the game. If we had all of this data available we could trace out which moves will guarantee a checkmate or, if winning is impossible, which moves will delay being checkmated the longest. Unfortunately this is not feasible as the number of different positions to keep track of grows exponentially, outpacing any machine's memory capacity. Instead, this search is typically limited to a certain depth, i.e. looking 4 moves ahead. Since checkmates don't often happen in only 4 moves, we need to introduce some intermediate measure that estimates how likely it is that the given side will win.

This measure is known as the evaluation function. It can be as simple as an indicator for whether a given side has been checkmated, but for limited-depth search this isn't useful. A slightly more mature estimate is in counting the pieces on each side in a weighed way. If white has no queen but black does, then the position is unbalanced and white is in trouble. If white is three pawns down but also has an extra bishop, the position is likely to be balanced. This evaluation function scores pieces in terms of how many pawns they're equivalent to.

An algorithm playing with this evaluation function will have general rules down. Taking pieces, especially rooks and queens, is typically good and sacrificing important pieces to take out pawns is typically bad. Sadly, its positional playing will be abhorrent.

A slightly move advanced method of scoring would not only take that into account, but also take into account the position of each piece on the board. This gives algorithms a surprising amount of positional intuition: pawns and knights should be in the center, rooks on the enemy's ranks are powerful, kings should be in the corners, so on. Scoring can be made even more sophisticated by accounting more subtle tactical and positional patterns like pawn structure, connected rooks, and king mobility.

From here we won't need to worry about what kind of evaluation function we use. For chess engines, evaluation and search are often independent of each other's implementations. This fact has recently been used in developing Stockfish NNUE, a Stockfish clone that uses a sparsely updated neural network as an evaluation function.

A search algorithm boils down to the way a chess engine compares possible moves. Thanks to the use of evaluation functions, search algorithms only need to search to a fixed depth to make reasonable decisions. How do they actually decide which moves to play?

Search algorithms involve a *search tree*, or a way of representing a board and all the possible moves to be made from that position on the board. The root node of the tree is the original board, and all of the nodes branching out from it are reachable positions.

The minimax algorithm takes advantage of the fact that chess is a zero-sum game. Maximizing your chances of winning is the same as minimizing the opponent's chances of winning. Each turn can be seen as a player making a move to maximize the evaluation function while the other tries to minimize it. In terms of a search tree, this means starting at a given node and choosing the children nodes with the best (or worst) scores.

Minimax is the "correct" way to do this, in the sense that if we were to let it go on indefinitely then it would play the game perfectly. That said, it is incredibly slow and impractical. This is thanks to the high branching factor of the search tree; any given position will have 10–20 possible moves, each of those new positions will have around the same amount, and so on. The number of nodes in a search tree increases exponentially with depth.

An early way to get around this was by using the "Shannon type B" variation of minimax. Instead of searching every possible move at a node (the type A approach), only a handful of the top moves are looked at. The candidate moves are decided either by the evaluation function, or some other heuristic like "checks-captures-threats". These types of algorithms keep the branching factor low on the search tree, so they are able to search to a greater depth. However, since they don't consider many moves they're prone to miss a lot of important tactics and fall into traps. By the 70's, an optimization of minimax known as alpha-beta pruning was discovered that made the Type A approach viable. Selecting candidate moves for a Type B algorithm ended up being too computationally expensive in comparison and it fell out of favor.

If you've ever played chess, you will know that some moves are just *bad*. These are typically moves that allow the other player to get a clear upper hand on the next few turns; such as blundering a piece. To standard minimax, these moves are just as important to think about as the others. In return, the algorithm gets weighed down by analyzing bad positions.

Alpha-beta pruning speeds up minimax by skipping the “irrelevant” nodes of a search tree. This can be accomplished by adding extra data to each node, an “alpha” and a “beta” value, which represent the worst outcome for each player from that node. Since the maximizing player knows that the minimizing player will pick a response that minimizes the evaluation, they also know that they can avoid thinking about moves that allow the minimizing player to make things worse than they already are. These moves are “pruned” from the search tree and skipped.

Alpha-beta pruning makes brute-force search possible. Even better, it yields exactly the same results as naive minimax would, so it’s also the “correct” algorithm for tree search. However, it is still limited.

In the best case, alpha-beta pruning reduces the computational time by a “square root”. For example, if minimax takes 100 seconds to determine the best move, then alpha-beta pruning will take only 10. In most cases, however, alpha-beta pruning will not preform as well and will take some intermediate amount of time to run.

The reason this happens is due to how this algorithm visits nodes. So far there’s no system for deciding which nodes to look at first, so the algorithm just looks through them at random. This can lead to a situation where the program looks at the worst move when it does not yet know that it’s the worst move, as opposed to skipping it when it knows that there’s already a better move to pick.

The trick to mimicking the ideal case for alpha-beta search comes through move ordering. The idea is to look at the most promising moves first so that bad moves can be quickly eliminated. Since it’s impossible to tell what moves are best without actually performing a search on them, this method has to be guided by heuristics. Thanks to this, there are several possible ways to implement this in a chess engine.

This technique is similar to the Type B approach to minimax. Instead of choosing a subset of moves, they are instead just preferentially ordered. This can be something simple like ordering captures first and considering everything else after all the captures have been analyzed. Some finer-scale heuristics may be used, such as first looking at capturing the last moved piece.

Non-capturing moves may also be ordered. The most common heuristic for these is the “killer” heuristic. Within the alpha-beta search tree, two sibling nodes (descended from the same parent node) are going to have very similar positions. This means that a move which causes an alpha-beta cutoff in one node will likely be just as important for its sibling nodes, so it will be analyzed first once the algorithm gets to those nodes.

Another heuristic method for move ordering is known as relative history. It’s actually a combination of the history and butterfly heuristics, which are inadequate by themselves. The history heuristic orders moves by the number of times they’ve cause an alpha-beta cutoff in the search tree. The butterfly heursitic orders moves by how many times they are played overall. Relative history orders moves by the ratio of these two scores, essentially picking out the moves that were most effective when played.

If we allow the search algorithm to keep information about its search tree from previous moves, then we can also employ a form of move-ordering based on each node’s preferred moves from the previous search. This way, a chess AI searching a position to a depth of 8 doesn’t need to throw away all of that analysis and start anew on the next turn.

Since this relies on storing nodes and information about them, there needs to be a memory-efficient way to store this data. For picking out principal variations, we can get away with storing the board as-is since we don’t need to store very many. This isn’t always the best way.

The speed of alpha-beta pruning is vastly improved by move ordering. Although it seems we’ve gone as far into optimizing this as possible, we can go even further through the use of transposition tables. A

transposition table stores data about nodes throughout the search algorithm, allowing it to skip nodes it has already seen before in the search.

In a game like chess, transposed nodes show up all across the search tree. With the use of transposition tables, a given transposition will only need to be analyzed once before the algorithm remembers it and knows not to waste time re-analyzing it elsewhere in the tree.

Keeping a transposition table requires a large amount of memory. Storing the entire position of a board becomes impractical here, so programmers got creative. Since the program only needs to check if two positions are the same, we can take an idea from cryptography and check their hashes instead.

The idea behind hashing two objects is that most of the information of the objects is obscured, but their equality can still be checked. In our case, “obscuring information” means simplifying the object to save memory. The next special property of hashing is that two similar objects should have completely dissimilar hashes; which means that the hashing function should be fairly random and chaotic.

Reference: https://medium.com/@SereneBiologist/the-anatomy-of-a-chess-ai-2087d0d565

(The Anatomy of a Chess AI)