**Project Report**

**Exploring Algorithms in Chess: A Journey into Artificial Intelligence and Game Theory**

# 1.Introduction

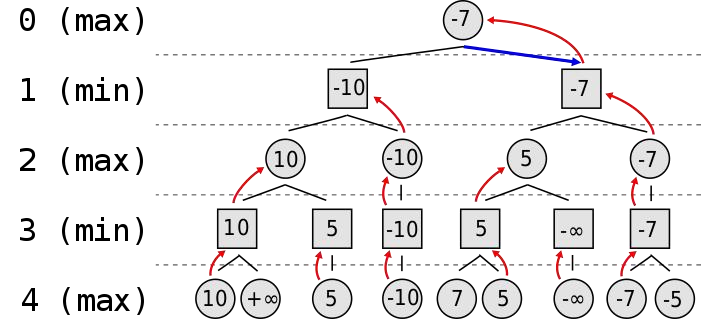
Chess, a timeless and intellectually demanding game, has served as a fertile ground for the convergence of artificial intelligence and game theory. Over the years, researchers and developers have crafted sophisticated algorithms to not only play chess at a competitive level but also to unravel the complexities inherent in strategic decision-making within the game. This report delves into the realm of chess-solving algorithms, examining the methodologies and strategies employed to tackle the chess game problem. From classic approaches like the Minimax Algorithm and Alpha-Beta Pruning to modern innovations such as Monte Carlo Tree Search (MCTS) and neural networks, the field has witnessed a fascinating evolution marked by computational prowess and creative problem-solving. As we embark on this exploration, we'll uncover the fundamental algorithms that form the backbone of chess engines, understand their underlying principles, and appreciate the intricate balance between computational efficiency and strategic insight. From heuristic evaluation functions that mimic human intuition to endgame tablebases that encode perfect play, each algorithm plays a crucial role in deciphering the complexities of the 64-square battlefield. Furthermore, the report will touch upon the interplay between traditional algorithmic techniques and the transformative power of neural networks. Deep learning, with its ability to discern intricate patterns and relationships, has redefined how chess engines approach position evaluation and move prediction. The fusion of classical algorithms with neural networks, as witnessed in engines like AlphaZero, showcases the synergy between age-old principles and cutting-edge technologies. As we navigate through the algorithms used to solve the chess game problem, we'll witness the chessboard transform into a playground for artificial intelligence, where computations rival strategic brilliance. Join us on this journey into the world of algorithms that not only play chess but also contribute to the broader

landscape of artificial intelligence and computational decision-making.

# 2.Algorithms

## 2.1 Minimax algorithm

Borrowing from Wikipedia's concise definition, the minimax algorithm is "a decision rule used ... for minimizing the possible loss for a worst case (maximum loss) scenario." With respect to chess, the player to act is the maximizer, whose move would be met with an adversarial response from the opponent (minimizer). The minimax algorithm assumes that the opponent is competent and would respond by minimizing the value (determined by some heuristic) of the maximizer.



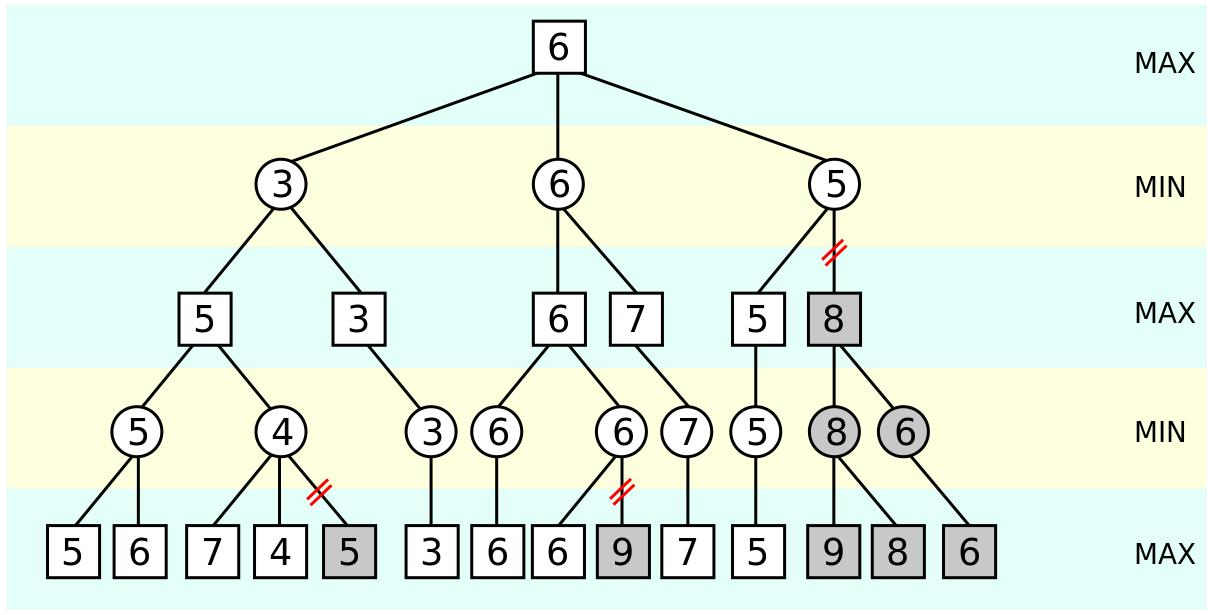
This simplified tree represents a game where each depth represents a player's turn. Starting at the bottom of the tree (the deeper into the tree, the further into the game), the leaf nodes' values are passed back up to determine the maximizer's current best move. In an actual chess game, the each depth would have many more branches with each branch representing a possible move by a chess piece.

### Alpha-Beta Pruning

Because of the number of board states possible in chess (estimated to be 10^120), minimax can be improved with a layer of alpha-beta pruning. By keeping track of alpha (the highest value guaranteed to the maximizer) and beta (the lowest value guaranteed to the minimizer), it is possible to avoid calculating the heuristics of

certain board states that cannot improve the situation for the current

player.



The grayed-out leaf node with a heuristic of 5 is never explored because the maximizer, at that point, is guaranteed a 5 by going left and can do no better than 4 by going right. That is, if the value of the grayed-out leaf node is greater than 4, the minimizer would choose the 4. If it were less than 4, the minimizer would choose it instead. From the maximizer's perspective, there is no reason to investigate that leaf

node. For more information on the history of chess, minimax, and alpha-beta

pruning, check out Patrick Winston's lecture.

### Heuristics

There are many factors when calculating the heuristics of a chessboard. As we developed our heuristic formula to consider more factors, the computations required to calculate the best move increased exponentially. At the moment, the AI considers the following 4 aspects of a board in its heuristic function.

### Material

The material heuristic compares the value of one's pieces with the opponent's pieces.

It encourages the AI to capture pieces and make favorable trades. We used the standard values for each piece:

* Pawn: 1
* Bishop/Knight: 3
* Rook: 5
* Queen: 9

### Number of Moves

This heuristic calculates the number of legal moves a player can make. It encourages the AI to develop its pieces to exert more control over the board. It prioritizes controlling the center where pieces will have more options to influence the game. For example, a queen near the center of the board can move in 8 directions and thus control more squares, whereas a queen on a corner square can only move in 3

directions.

### Pawn Structure

The pawn structure heuristic gives a score based on the number of pawns supported by other pawns. It encourages the AI to develop its pawns in a way that allows pawns moving forward to be defended by other pawns from behind.

### Check Status

This heuristic checks if a player is in check or checkmate status. It encourages the AI

to make moves that would put the opponent in check while avoiding moves that would put itself in check. It also detects if a move would put opponent in checkmate, which would be prioritized over all other heuristics.

### Other Heuristics

The heuristics we used don't come close to representing all the depth involved in chess. Creating a heuristic that would encourage the AI to employ more complex

strategies and tactics is not only conceptually difficult. It is also computationally

demanding. As we optimize our engine, we would like to tweak our heuristics to

better match the complexities of chess.

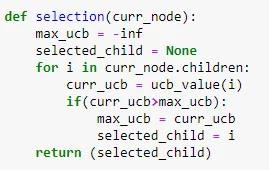
## 2.2 Monte Carlo tree search algorithm

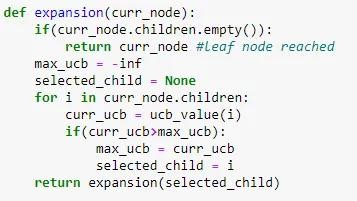
MCTS is a heuristic search algorithm that combines elements of random simulation and tree search. It uses statistical sampling to guide the search through the game tree. While the simulations are not based on strict heuristics, the selection and expansion phases involve heuristics to prioritize promising moves.

Monte Carlo Tree Search is a widely used algorithm in game theory which makes decision based on updated policy devised by the tree with each iteration. The main advantage of MCTS is that it can operate effectively without any knowledge in the particular domain, apart from the rules and end conditions, and can find its own moves and learn from them by playing random playouts. It consists of four stages Selecting → Expanding → Rollout → Backpropagating in each iteration. It is better than traditional tree search algorithm as it does not explore every possible state to make decision as while it is feasible to store all possible states of games like tic-tac-toe, it fails when we want to predict moves of games like chess and go as it has millions of trillions possible states which is impossible to store and process. In that case, MCTS provides us with best probabilistic move given the current state of the game.

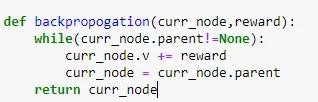
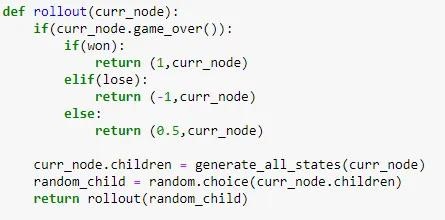
This algorithm has 4 main steps which are selection, expansion, simulation, backpropagation. In the context of chess, MCTS faces the challenge of dealing with the vast branching factor of the game tree. The number of possible moves in chess grows exponentially with the depth of the tree, making exhaustive search impractical. MCTS addresses this by using a combination of random simulations and careful

selection/expansion strategies.

 (Selection)

 (Expansion)

(Rollout)



(Backpropagation)

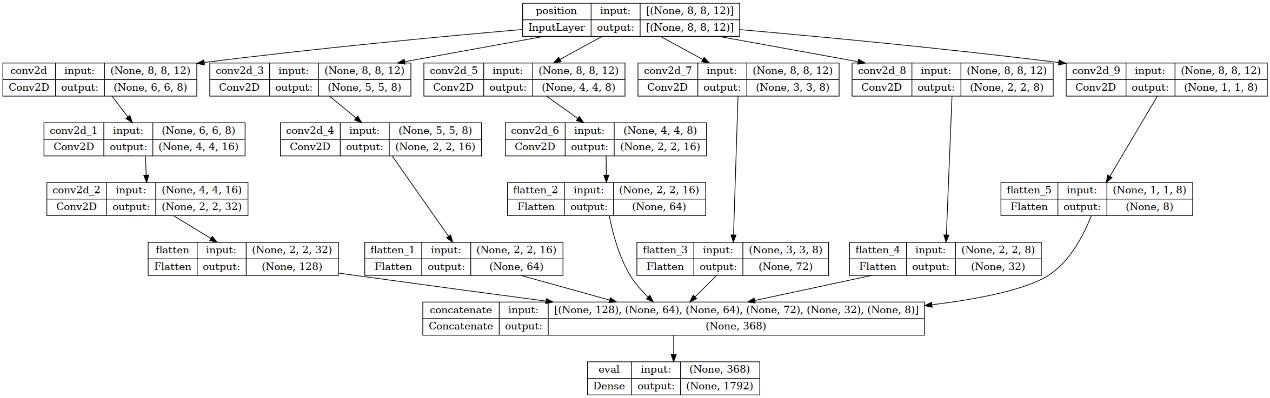
## 2.3 Neural Network

The concept of the program uses a neural network to evaluate the board, which is then fitted with the search algorithm, which checks all future position and finds the highest value, sort of like a min-max tree.

Deep learning techniques, including neural networks, are used in chess for various tasks such as move prediction and position evaluation. Neural networks can learn complex patterns and relationships from data, and their outputs are often used as heuristics to guide the search in game-playing algorithms.

The success of CNN-Go can be attributed to smooth arrangements of positions that are approximately continuous through and between games. Additionally, since each move in Go adds a single piece to the board, essentially flipping the value of one pixel, the difference in board representations before and after a move is smooth, constant, and almost always linked to the important patterns observed by the

network, which contributes to the consistency of Go classification algorithms.

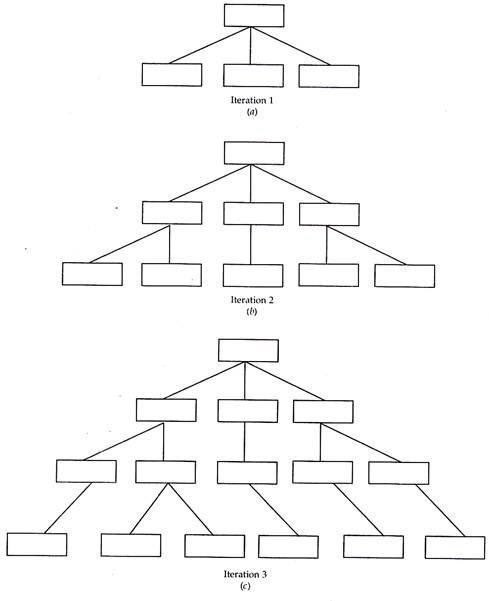


(Neural Network Structure)

## 2.4 Iterative Deepening Method

Ideas for searching in game have led to new algorithms in heuristic search. One such idea is Iterative Deepening, originally used in a program called chess 4.5. Rather than searching the game tree up to a fixed length, in this program the tree is generated up to a single ply, and static evaluation function is applied to the result of each of the possible moves. The minimax search is then initiated up to a depth of two plies and to more plies and so on. The name “iterative deepening” derives its name from the

fact that on each iteration, the tree is searched one level deeper. Fig. 5.18, illustrates the method. This method is also called progressive deepening.



At the first instance the procedure seems wasteful. Then why peruse? There are some justifiable reasons for this iteration, especially in game playing programs subjected to time constraints. (For example, chess game may be required to be completed in 2 hrs.). Since it is impossible to know in advance how long a fixed depth tree will take because of the variation in pruning efficiency and the need for selective search, a program may run of time.

With iterative deepening the current search can be aborted at any time and the best move found by previous iteration can provide invaluable move ordering constraints. If one move was judged to be superior to its siblings in a previous iteration, it can be searched first in the next interaction. With effective ordering, the α-β procedure can prune many more branches and the total search time can be decreased drastically. This allows more time for deeper iterations. Chess 4.5’s success with iterative deepening, can be applied to the single-agent search to solve the problems like that of 8 puzzle. An algorithm combining the salient features of depth-first and breadth first, is called Depth First Deepening (DFID).

**Advantages of Iterative Deepening Idea of Game Searching:**

1. Find the shortest path to the goal state. 2. The maximum amount of memory used by DFID is proportional to the number of nodes in that solution path. This algorithm has some disadvantages also. All the interactions except the final one are essentially wasted, which is not a serious problem because most of the activity during an given interaction occurs at the leaf node level. Assuming a complete tree, we see that there are as many leaf nodes at level n as there are total nodes in levels 1 to n. Thus, the work expended using the nth iteration is roughly equal to the work expended during all previous iterations. This means that DFID is only slower than the depth-first search by a consistent factor. In the case of Depth First (DF) there is no

way to know in advance how deep the solution lies in the search space. DFID

avoided the problem of choosing cut-off without sacrificing efficiency. In fact, DFID is the optimal algorithm, in terms of time and space, for blind search.

**Iterative Deepening A\*:**

Iterative deepening can also be used to improve the performance of heuristic informed search like the A\* search algorithm. The difficulty with A\* of requiring the average amount of memory, to maintain the search node lists can be obviated to a

great extent with iterative deepening.

**Algorithm Iterative Deepening A\* (IDA\*):**

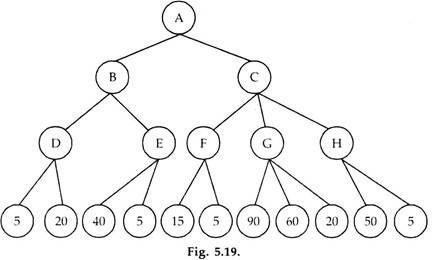
1. THRESHOLD = the heuristic evaluation of the start state.
2. 2. Correct a depth-first search, pruning any branch when its total function (g + h’)

exceeds THRESHOLD. If a solution path is found during the search, return it.

1. 3. Otherwise, increment THRESHOLD by minimum amount it was exceeded

during the first step. Go to step 2.

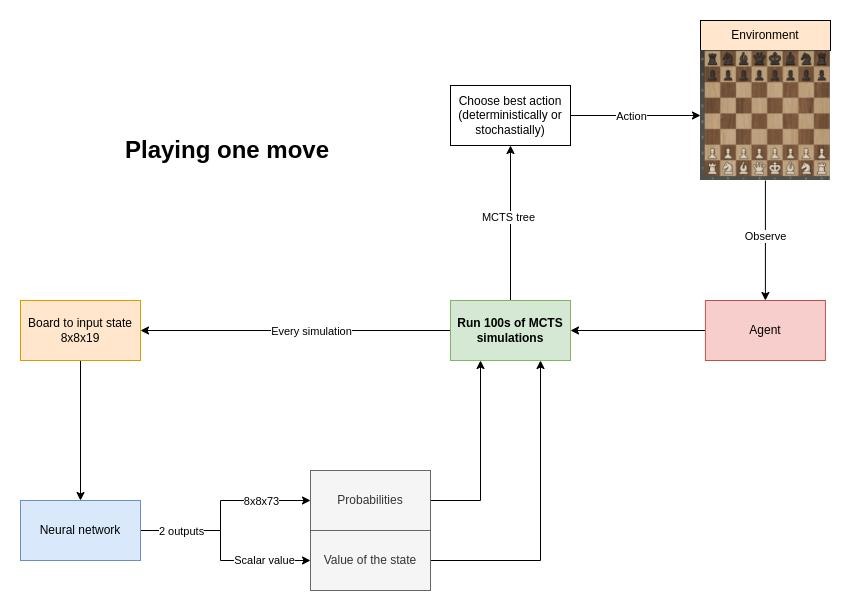
Like A\*, IDA\* is guaranteed to find an optimal solution provided h’ is an admissible heuristic. The IDA\* search technique is very efficient with respect of space, because of it being a depth-first search. IDA\* was the first heuristic algorithm to find the optimal solution paths for the 15 puzzle within reasonable time and space constraints. Moreover, game playing has been a fertile ground for experiments in machine learning. Machine learning and neural networks have been applied to the game GO which has led to the field for Pattern Matching.



## 2.5 Deep Reinforcement learning

During the research, there’s another method that combined both MCTS and neural network, normal chess engines work with the minimax algorithm: the engine tries to find the best move by creating a tree of all possible moves to a certain depth, and cutting down paths that lead to bad positions (alpha-beta pruning). It evaluates a position based on which pieces are on the board.

This chess engine is based on AlphaZero by Deepmind. It uses a neural network to predict the next best move. The neural network learns by playing against itself for a high amount of games, and using their results to train the network. The newly trained neural network is evaluated against the old network by playing many games against each other, and the best network is kept. This process is repeated for a long time.



**The neural network**

Input layer: 19 8x8 boards of booleans



20 hidden layers:

Convolutional hidden layer

19 residual blocks with skip-connections

2 outputs:

The win probabilities of each move (73 boards of 8x8 floats)

The value of the given board (scalar)



=> 30+ million parameters.

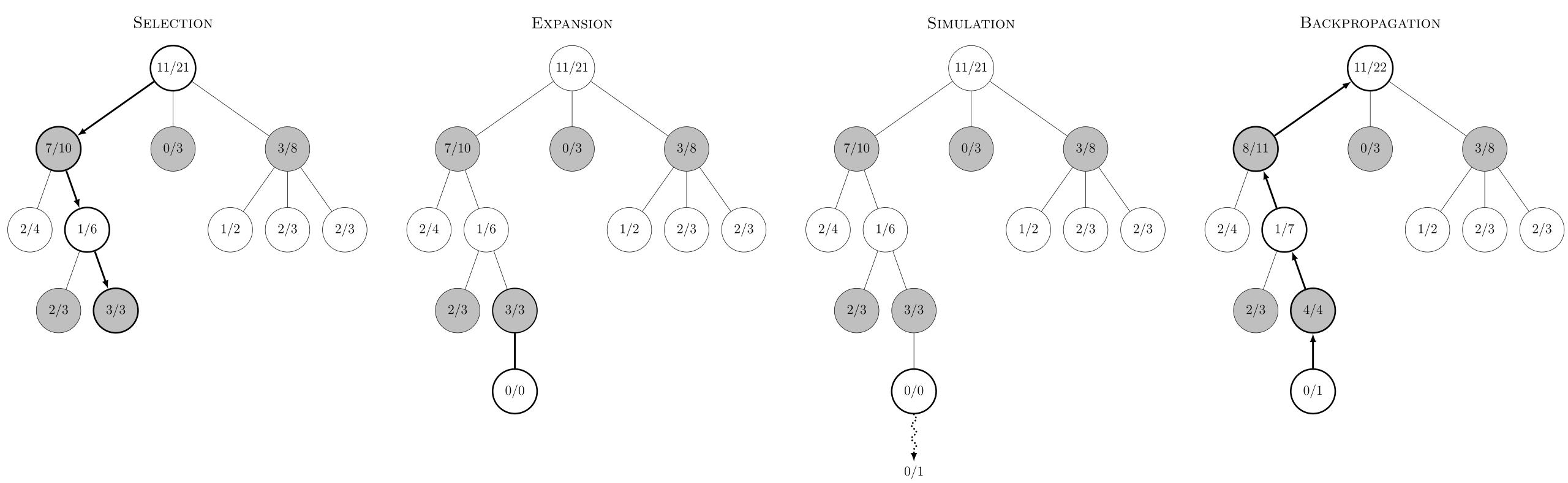
Every move, run a high number amount of MCTS simulations. AlphaZero uses an custom version of MCTS.

**Normal Monte Carlo Tree Search:**

1. **Selection**: Traverse the tree randomly until a leaf node is reached.
2. **Expansion**: expand the leaf node by creating a child for every possible action 3. **Simulation**: 'rollout' the game by randomly choosing moves until the end of the game.

4. **Backpropagation**: backpropagate the result of the rollout to the root node.

In chess, normal MCTS would be incredibly inefficient, because the amount of actions every position can have is too high (step 1), and the length of the game can be very long when choosing random moves (step 3).

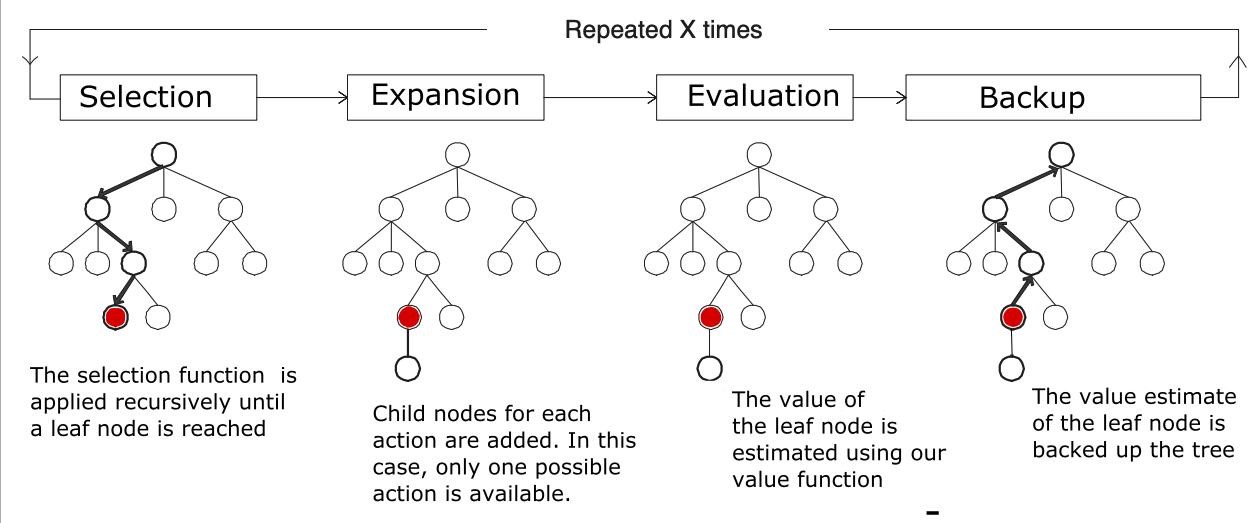


**AlphaZero's MCTS]**

AlphaZero uses a different kind of MCTS: step 1 (Selection) is not random, but based on neural network predictions and

upper confidence bound. step 3 (Simulation) is replaced by the value prediction received by the neural

network (Evaluation).



To run one MCTS simulation:

1. To traverse the tree, keep selecting the edges with maximum Q+U value.

Q = mean value of the state over all simulations.

U = upper confidence bound.

Do this until a leaf node is reached (= a node which has not been visited/expanded yet).

1. Expand the leaf node by adding a new edge for every possible action in the state.

Input the leaf node into the neural network.

The output:

* + 1. The probabilities.
    2. The value of the state.

Initialize the new edge's variables with these values:

N = 0

W = 0

Q = 0

P = p\_a (prior probability for that action) Add nodes (new states) for each action to the tree.

1. Backpropagation.

From the leaf node, backpropagate to the root node.

For every edge in the path, update the edge's variables.

N = N + 1

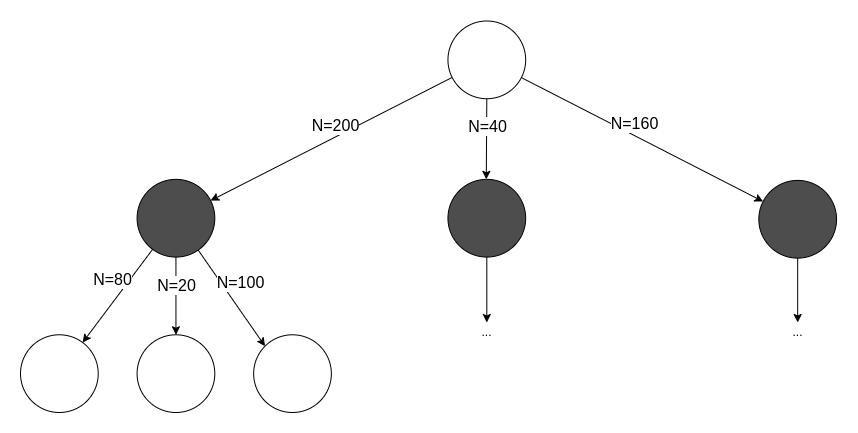
W = W + v, v is the value of the leaf node predicted by the NN in step 2.

Q = W / N

**After these simulations, the move can be chosen:**

The move with greatest (deterministically)

According to a distribution (stochastically): π ~ N



**Creating a training set**

To train the network, you need a lot of data.

You create this data through self-play: letting the AI play against a copy of itself for many games..

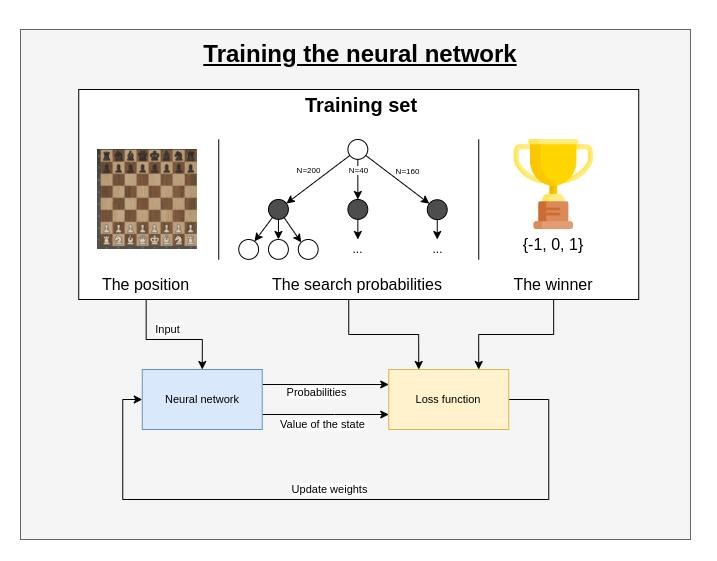
For every move, store: The state

The search probabilities

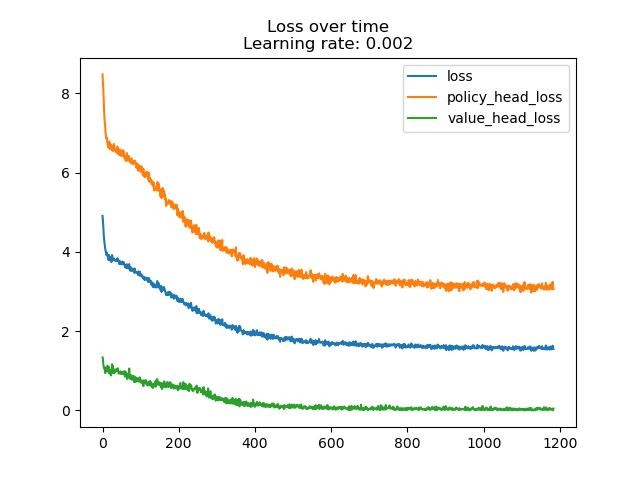
The winner, (added once the game is over)

**Training the network**

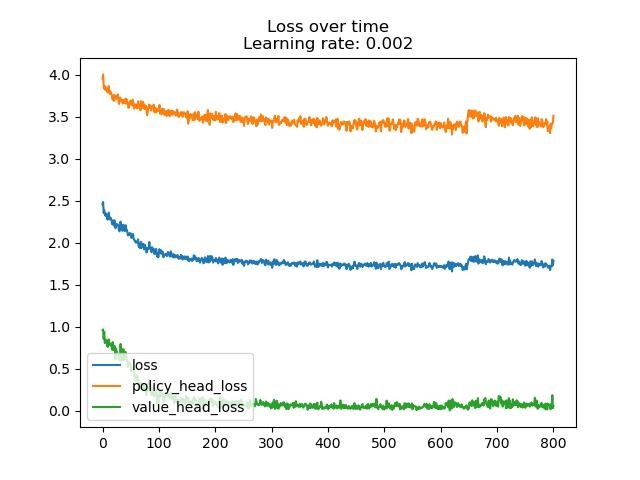
Sample a mini-batch from a high amount of positions (see training set) Train the network on the mini-batch



The first training session:

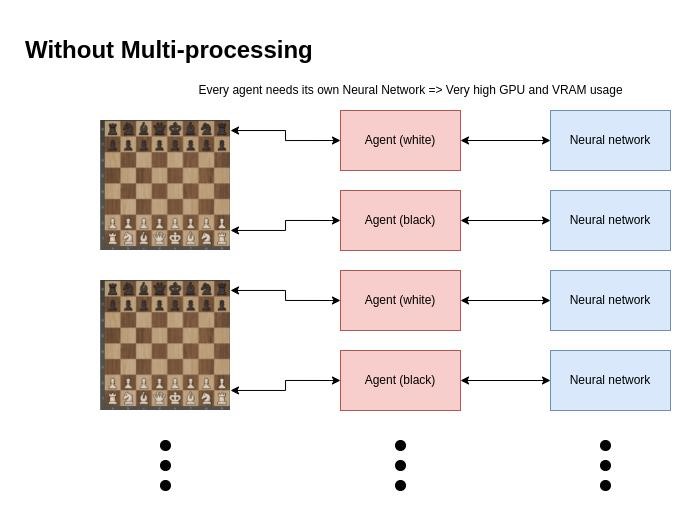


The second training session:

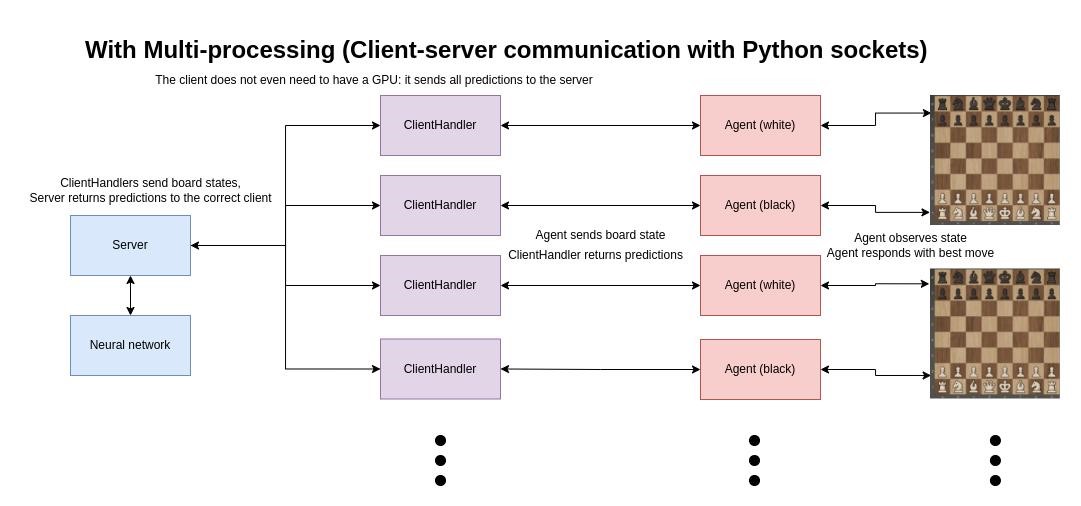


**Multi-processing improvements**

It is necessary to create a huge training set of positions by making the current best AI play against itself. To do that, I had the problem that playing multiple games in parallel was not possible because every agent needs access to the network:



To fix this, I created a server-client architecture with Python sockets: the server has access to the neural network, and the client sends predictions to the server. The server then sends the predictions back to the correct client. This is much more scalable and can be dockerized.



The result: much faster self-play. The other clients' GPUs do not get used, meaning any system with a good processor can run multiple self-play games in parallel when connected to a server. **Evaluate the network**

To know whether the new network is better than the previous one, let the new network play against the previous best for a high amount of games. Whoever wins the most games, is the new best network.

Use that network to self-play again. Repeat indefinitely.

# 3.Advantages and disadvantages

So far, we’ve managed to summarize four different heuristic search algorithms that can be used to solve chess-game problems, now let’s talk about both pro and con

about these algorithms.

First we’ve got Minimax Algorithm, which is a fundamental

decision-making algorithm used in two-player games like chess. It has both advantages and disadvantages in the context of chess. It shows three main

advantages which are:

1. **Optimality:**

The Minimax algorithm is guaranteed to find the optimal move in a game with

perfect information, assuming that both players play optimally. It explores the entire game tree and chooses the move that minimizes the maximum possible loss.

1. **Strategic Decision-Making:**

Minimax considers the strategic implications of each move. It evaluates positions based on potential outcomes, allowing it to make decisions that take into account the long-term consequences of moves.

1. **Widely applicable:**

Minimax is a general-purpose algorithm applicable to a broad range of games beyond chess. As long as a game has defined rules, Minimax can be adapted to find

optimal strategies.

And then let’s look at what are disadvantages this algorithm has.

1. **Exponential Growth of Game Tree:**

The primary disadvantage of Minimax in chess is the exponential growth of the game tree. The branching factor (number of possible moves) at each level of the tree is quite high in chess, leading to an enormous number of possible positions to

evaluate. This makes Minimax impractical without additional optimizations.

1. **Complete Search Impractical:**

Due to the vast number of possible moves in chess, a complete search using Minimax is often computationally infeasible. Even with optimizations like alpha-beta pruning, the search space can be too large for exhaustive exploration, especially in deep positions.

1. **Difficulty Handling Large State Spaces:**

Chess has an enormous state space, and Minimax struggles to handle the complexity of positions with a high number of possible moves and counter-moves. This

limitation can lead to shallow searches or suboptimal moves in practice.

1. **Assumes Perfect Information:**

Minimax assumes perfect information, meaning both players have complete

knowledge of the game state. In practice, this assumption may not hold, especially in situations where players make mistakes or when dealing with imperfect information. In summary, while Minimax provides a theoretically optimal solution for perfect information games, its practical application in chess requires careful consideration of computational constraints and the need for additional techniques to make it more

efficient and applicable to real-world scenarios.

Secondly, we’ve got Monte Carlo algorithms, particularly Monte

Carlo Tree Search (MCTS), have gained prominence in the field of artificial intelligence for playing complex games like chess. Here are the advantages and disadvantages of using Monte Carlo algorithms in the context of chess:

**Advantages:**

1. **Adaptability to Complex Games:**

MCTS is highly adaptable and effective in games with large and complex decision spaces, making it well-suited for chess, where the number of possible moves

grows exponentially.

1. **Effective in Decision-Making Under Uncertainty:**

MCTS can handle games with imperfect information or uncertainty, making it suitable for scenarios where players might not have complete knowledge of the

game state.

1. **Dynamic Tree Growth:**

MCTS dynamically grows the search tree based on promising branches,

allocating computational resources to areas that are more likely to yield valuable

information. This adaptability helps focus on relevant parts of the game tree.

1. **Parallelization:**

MCTS is amenable to parallelization, allowing multiple branches of the game tree to be explored simultaneously. This makes it more scalable, leveraging modern

multi-core processors and distributed computing.

1. **No Need for Heuristic Evaluation Functions:**

Unlike traditional minimax-based approaches, MCTS does not rely heavily on heuristic evaluation functions. Instead, it uses random simulations to evaluate

positions, allowing it to handle complex and dynamic positions.

**Disadvantages:**

1. **Computationally Intensive:**

MCTS can be computationally intensive, especially when simulating a large number of random games to evaluate positions accurately. This can make it

challenging to achieve real-time performance, especially in scenarios with limited

computational resources.

1. **Sensitivity to Simulation Quality:**

The quality of MCTS heavily depends on the quality of the simulations. If the simulations are not representative of actual gameplay or if the simulation depth

is limited, the algorithm may produce suboptimal results.

1. **Tuning Exploration Parameters:**

MCTS involves parameters that control the balance between exploration and exploitation, such as the exploration constant in the UCT formula. Tuning these parameters for optimal performance can be challenging and may require

domain-specific knowledge.

1. **May Struggle with Deep Tactical Positions:**

In positions with deep tactical nuances, MCTS may struggle to perform well unless a large number of simulations are conducted. The exponential growth of the game tree in chess can lead to shallow searches, missing critical tactical

moves.

1. **Limited Understanding of Long-Term Strategies:**

MCTS primarily focuses on local tactical considerations rather than long-term strategic planning. In complex positions with deep strategic implications, traditional minimax-based approaches may have an advantage in understanding

overarching plans.

In summary, while MCTS has proven to be a powerful algorithm for game-playing AI, including chess, it comes with challenges related to computational intensity and the need for effective simulation strategies. Its adaptability and ability to handle uncertainty make it a valuable tool, but its performance can be influenced by the specific characteristics of the game and the quality of the simulations. Researchers

often explore hybrid approaches that combine MCTS with other techniques to

address these challenges effectively.

Moving on, we’ve got Neural Network Algorithm, Using neural

network algorithms in chess, particularly in the context of creating chess-playing engines, has both advantages and disadvantages. Let's explore these aspects:

**Advantages:**

1. **Pattern Recognition:**

Neural networks excel at recognizing complex patterns and relationships within data. In chess, where strategic patterns and positions play a crucial role, neural

networks can learn and generalize from vast datasets of high-level games.

1. **Position Evaluation:**

Neural networks can be trained to provide more accurate and nuanced

evaluations of chess positions compared to traditional heuristic evaluation functions. They can capture subtleties that are challenging to express explicitly

through handcrafted rules.

1. **Adaptability to Various Playing Styles:**

Neural networks can adapt to various playing styles and strategies. By training on diverse datasets, they can learn to emulate the playing style of different

grandmasters or adapt to dynamic changes in the game.

1. **Integration with Monte Carlo Tree Search (MCTS):**

Neural networks are often integrated with Monte Carlo Tree Search (MCTS) algorithms. This combination, as seen in engines like AlphaZero, allows the neural network to guide the search process, providing a more informed

exploration of the game tree.

1. **Learning from Experience:**

Neural networks can learn from experience and continuously improve their performance over time. This adaptability allows them to refine their evaluations

based on exposure to new positions and gameplay scenarios.

**Disadvantages**:

1. **Computational Intensity:**

Training and running neural networks for chess applications can be

computationally intensive, requiring substantial resources. Training on large datasets and conducting evaluations during gameplay can be time-consuming,

limiting real-time applicability in some scenarios.

1. **Interpretability:**

Neural networks are often considered as "black-box" models, meaning their decision-making processes are not easily interpretable by humans. Understanding why a neural network made a particular move in a chess game

can be challenging, which may limit their application in critical or high-stakes

scenarios.

1. **Overfitting to Training Data:**

Neural networks are susceptible to overfitting, where they memorize patterns from the training data rather than generalizing to new, unseen positions. This can lead to suboptimal performance in real-world situations or against

opponents employing unexpected strategies.

1. **Limited Understanding of Chess Concepts:**

Neural networks might lack an intrinsic understanding of chess concepts. While

they can learn to imitate strong moves from training data, they may not comprehend the underlying strategic principles in the same way a human

grandmaster does.

1. **Dependency on Training Data Quality:**

The performance of a neural network in chess is heavily dependent on the

quality and diversity of the training data. Incomplete or biased datasets may lead

to the network learning suboptimal strategies or patterns.

In summary, the use of neural network algorithms in chess engines introduces a powerful tool for position evaluation and move prediction. However, challenges

related to computational intensity, interpretability, and the risk of overfitting must be carefully addressed to fully harness their potential in creating strong and reliable chess-playing programs. Researchers continue to explore ways to optimize neural network architectures and training methodologies for improved chess AI

performance.

Last but not least, we have Iterative Deepening, Iterative Deepening is a search strategy used in chess algorithms to balance the trade-off between

computational resources and the depth of the search. It involves performing multiple searches with increasing depth until a specified limit is reached. Here are the advantages and disadvantages of the Iterative Deepening Method in the context of

chess.

**Advantages:**

1. **Completeness:**

Iterative Deepening is a complete search strategy, meaning it guarantees finding the optimal solution if enough time is given. This property is crucial in games like chess, where finding the best move is essential for optimal play.

1. **Optimality in Limited Time:**

In scenarios where there is a time constraint on move calculation (as is common in chess tournaments), Iterative Deepening can provide reasonably good moves even if the search is terminated before reaching the deepest level.

1. **Better Move Ordering:**

By exploring shallower depths first, Iterative Deepening can improve move ordering for the deeper searches. This can be particularly beneficial when combined with techniques like alpha-beta pruning, reducing the number of nodes evaluated.

1. **Dynamic Adaptation to Time Constraints:**

Iterative Deepening allows the algorithm to dynamically adapt to time

constraints. If there is a limited amount of time available for move calculation, the algorithm can still return a move based on the deepest search completed within that time frame.

1. **Memory Efficiency:**

Iterative Deepening typically requires less memory compared to fixed-depth searches because it discards the previous search results at each iteration, storing only the results of the current depth.

**Disadvantages**:

**1.Redundant Work:**

Iterative Deepening can lead to redundant work, as the same positions are often evaluated multiple times at different depths. This redundancy can slow down the search process, especially in positions where the branching factor is high.

**2.Inefficiency in Certain Positions:**

In positions where the most critical moves are located deeper in the search tree, Iterative Deepening may not be as efficient as fixed-depth searches. This is because it explores deeper depths only after completing shallower searches.

**3.May Miss Short-Term Tactics:**

Due to its focus on exploring deeper depths gradually, Iterative Deepening might prioritize long-term strategies over short-term tactics. In positions where immediate tactics are crucial, this approach may not be as effective.

**4.Dependence on Evaluation Function:**

The effectiveness of Iterative Deepening is closely tied to the quality of the evaluation function used. If the evaluation function is not accurate or is biased, the algorithm may not identify the most promising lines of play.

In conclusion, Iterative Deepening is a practical and widely used search strategy in chess algorithms, balancing the need for optimality with the constraints of computational resources. While it provides advantages such as completeness and adaptability to time constraints, its potential for redundant work and inefficiency in

certain positions should be carefully considered in the overall design of a

chess-playing engine.

# Future work

Since people use heuristic algorithm to solve chess-game for a long period of time, it is pretty mature nowadays, the field of heuristics in chess is continuously evolving, and there are several potential avenues for future work to enhance the application of heuristics in chess games. Here are some areas that researchers and developers may

explore:

1. **Deep Reinforcement Learning (DRL)**: Integrating deep reinforcement learning techniques with heuristic evaluation functions could lead to more adaptive and dynamic evaluations. DRL allows systems to learn from experience, potentially improving their understanding of chess positions and strategies over time.
2. **Neuroevolution and Genetic Algorithms**: Further exploration of neuroevolution and genetic algorithms for evolving heuristic evaluation functions could be a promising avenue. Optimizing heuristics using evolutionary techniques might lead to more effective strategies and evaluations.
3. **Ensemble Methods**: Combining the strengths of multiple heuristics through ensemble methods could result in a more robust and versatile evaluation function. Ensemble methods have been successful in various machine learning applications and might contribute to improved chess engines.
4. **Explainable AI**: As AI systems, including chess engines, become more complex, there is a growing need for explainability. Future work could focus on developing heuristic models that provide more transparent explanations for their decisions, making them more interpretable to users.
5. **Hierarchical Heuristics**: Developing hierarchical heuristics that operate at different levels of abstraction could be a fruitful direction. This approach could involve high-level heuristics for strategic planning and low-level heuristics for tactical considerations, creating a more organized decision-making process.
6. **Quantum Computing**: With the emergence of quantum computing, exploring

the potential of quantum algorithms for heuristic evaluation in chess is an area that could bring new perspectives to the field. Quantum algorithms might provide

advantages in solving certain types of chess-related problems.

1. **Experiments with New Heuristic Features**: Researchers may experiment with new heuristic features and domain-specific knowledge. Innovations in feature engineering could lead to more effective and discriminating heuristics. As technology and research in artificial intelligence advance, the application of heuristics in chess games will likely continue to evolve, contributing to the development of more sophisticated and capable chess-playing engines.

# Conclusion

In this paper, I have managed to organized several heuristic algorithms that people being using for solving chess-game, it is quite a ride to explore all the possibilities and different approaches to solve this topic, and it is also fascinating to actually putting the computer science theory and logic to solve the real word’s problem, this topic specifically, Chess has existed for 2000 years in the history of mankind, and through all these years it has been recognized as a game of strategy, a game of games, a game that mankind has been refining for 2000 years, with countless different tactics, ways of playing, and variations. With the development of technology and the advancement of artificial intelligence, mankind has tried to use the power of computers to create an opponent that can defeat all of mankind. It's incredible that before Alpha Dog, no one could imagine that computers could be powerful enough to defeat mankind, but that's the incredible thing about computers, if there is a powerful algorithm coupled with an algorithm that can continually learn and evolve itself, it's not going to be possible. If there is a strong enough algorithm and a model that can constantly learn and evolve, then it is only a matter of time before a computer beats humans. And this problem with the progress of science and technology, hardware development, optimization of algorithms, and ultimately achieve the victory over humans, in this paper, I collected different kinds of algorithms, are these years people through the analysis of the underlying logic of chess, coupled with heuristic algorithms of the characteristics of the actual use of the solution to the specific subject of chess, these algorithms I collected with a different point of view These algorithms I have collected use different perspectives and logics, and each has its own unique properties, advantages and disadvantages. In this thesis I have tried to explain them, to analyze them, to run them, to study the differences between the different algorithms, and after this round of study I understand better the power of heuristic algorithms, and the infinite possibilities of applying them to real problems in the real world.

**References**:

1. <https://en.wikipedia.org/wiki/Chess>
2. <https://www.chess.com/>
3. "Deep reinforcement learning," Wikipedia. Jan. 29, 2022. Accessed: Feb. 01, 2022.

[Online]. Available:

[https://en.wikipedia.org/w/index.php?title=Deep\_reinforcement\_learning&oldid=10](https://en.wikipedia.org/w/index.php?title=Deep_reinforcement_learning&oldid=1068657803)

68657803

1. “Reinforcement learning,” Wikipedia. Jan. 15, 2022. Accessed: Feb. 01, 2022.

[Online]. Available:

[https://en.wikipedia.org/w/index.php?title=Reinforcement\_learning&oldid=1065862](https://en.wikipedia.org/w/index.php?title=Reinforcement_learning&oldid=1065862559)

559

1. “AlphaZero,” Wikipedia. Jan. 15, 2022. Accessed: Feb. 01, 2022. [Online]. Available:

<https://en.wikipedia.org/w/index.php?title=AlphaZero&oldid=1065791194>

1. “AlphaGo,” Wikipedia. Jan. 25, 2022. Accessed: Feb. 01, 2022. [Online]. Available:

<https://en.wikipedia.org/w/index.php?title=AlphaGo&oldid=1067772956>7. “AlphaGo Zero,” Wikipedia. Oct. 14, 2021. Accessed: Feb. 01, 2022. [Online].

Available: <https://en.wikipedia.org/w/index.php?title=AlphaGo_Zero&oldid=1049954309>8. “Monte Carlo tree search,” Wikipedia. Jan. 23, 2022. Accessed: Feb. 01, 2022.

[Online]. Available:

[https://en.wikipedia.org/w/index.php?title=Monte\_Carlo\_tree\_search&oldid=106739](https://en.wikipedia.org/w/index.php?title=Monte_Carlo_tree_search&oldid=1067396622)

6622

1. “Minimax,” Wikipedia. Jan. 18, 2022. Accessed: Feb. 01, 2022. [Online]. Available:

<https://en.wikipedia.org/w/index.php?title=Minimax&oldid=1066446492>

1. “Alpha–beta pruning,” Wikipedia. Jan. 30, 2022. Accessed: Feb. 01, 2022. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta\_pruning&oldid=10](https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=1068746141)

68746141

1. <http://vision.stanford.edu/teaching/cs231n/reports/2015/pdfs/ConvChess.pdf>
2. Erik Bernhardsson. Deep learning for... chess. 2014
3. Christopher Clark and Amos Storkey. Teaching deep convolutional neural networks to play go. arXiv preprint arXiv:1412.3409, 2014. 14. <https://en.wikipedia.org/wiki/Monte_Carlo_tree_search>

15. https://towardsdatascience.com/train-your-own-chess-ai-66b9ca8d71e4