Monte Carlo Tree Search in Chess

Sarthak Shukla, Pradyumna Munjewar, Ayush Kumar Singh, Mukesh Mandal

***Abstract*– Monte Carlo Tree Search (MCTS) has emerged as a powerful and adaptive algorithm for decision-making in complex domains, particularly in the field of artificial intelligence and game theory. In the context of chess, where the branching factor of possible moves makes traditional methods computationally challenging, MCTS offers a promising approach to guide intelligent gameplay. This research paper presents an in-depth exploration of the application of MCTS in the realm of chess, focusing on its implementation, optimization, and its impact on chess-playing programs. We delve into the algorithm's adaptability to various aspects of the game, including endgame strategies, tactics, and openings. We also investigate the computational resources required for efficient MCTS-based chess engines, discussing hardware acceleration and parallelization techniques. Furthermore, we examine the implications of incorporating domain-specific knowledge, such as opening book and endgame tablebases, to enhance MCTS performance. The paper concludes by discussing the challenges and future directions in harnessing the potential of MCTS to revolutionize chess gameplay and its broader implications for artificial intelligence in strategic decision-making domains.**

Keywords - Monte Carlo Tree Search (MCTS),Heuristic Function, Upper Confidence Bound (UCB)

1. INTRODUCTION

Chess, often referred to as the "royal game," has long served as a crucible for testing the prowess of human intellect. With its rich history and myriad complexities, chess presents a profound challenge that has captivated the minds of players and strategists for centuries. The quest to develop intelligent agents capable of challenging and even surpassing human mastery of this board game has been an enduring pursuit in the field of artificial intelligence.

In recent decades, the emergence of powerful computational algorithms, coupled with the inexorable rise of computational hardware capabilities, has led to remarkable advancements in chess-playing programs. Classic approaches, like alpha-beta pruning algorithm and heuristic search functions have seen tremendous success in computer chess, enabling machines to compete at the highest levels.

However, the sheer combinatorial explosion inherent in the game of chess, with its vast tree of possible moves and positions, poses a significant challenge for traditional methods. As the game progresses, the branching factor of possible moves grows exponentially, rendering exhaustive search impractical. This challenge has paved the way for the introduction of Monte Carlo Tree Search (MCTS), a heuristic-driven algorithm that harnesses the power of random simulations and machine learning to guide decision-making.

MCTS is a decision-making algorithm that has demonstrated remarkable success in various domains, from Go to robotics, and has garnered considerable interest for its potential application in chess. It is characterized by its adaptability, its ability to explore complex decision spaces, and its proficiency in decision-making under uncertainty. In the context of chess, MCTS offers an innovative approach to navigate the game's intricate decision tree, promising strategic and tactical insights that may rival or surpass those of traditional chess engines.

MCTS is a simple algorithm to implement. Monte Carlo Tree Search is a heuristic algorithm. MCTS 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. The MCTS can be saved in any intermediate state and that state can be used in future use cases whenever required. MCTS supports asymmetric expansion of the search tree based on the circumstances in which it is operating. These are the advantages of MCTS.

This research paper embarks on a comprehensive exploration of the application of Monte Carlo Tree Search in the realm of chess. Our journey delves into the intricacies of implementing MCTS for chess, optimizing its performance, and deciphering its impact on chess-playing programs. We will examine the adaptability of the algorithm across different facets of the game, including endgame strategies, tactics, and opening principle Additionally, we will scrutinize the computationa resources necessary for an efficient MCTS-based ches engine, with an emphasis on hardware acceleration an parallelization techniques. The incorporation of domain-specific knowledge, such as opening books and endgame tablebases, will also be explored to augment MCTS's efficacy.

In essence, this paper aspires to shed light on the potential of MCTS to transform the landscape of chess gameplay and its broader implications for strategic decision-making in the domain of artificial intelligence.

2. Related Works

Monte Carlo Tree Search (MCTS) has gained significant attention and success in various board games, including Tic-Tac-Toe and Go. In Tic-Tac-Toe, MCTS has been applied due to its ability to handle uncertainty and its adaptability to different game structures with relatively low computational overhead. Similarly, MCTS has been a cornerstone in the success of computer Go programs, as demonstrated by AlphaGo, which utilized MCTS with deep neural networks to achieve superhuman performance. In the domain of chess, traditional search algorithms such as Alpha-Beta Pruning have historically dominated game-playing engines. Chess engines like Stockfish and AlphaZero heavily rely on the efficiency of Alpha-Beta Pruning, allowing them to explore deep search trees and analyze complex positions. MCTS have been used in various examples. [1] MCTS algorithm is applied to control Pac-Man character in the real-time game Ms Pac-Man. MCTS is used to find an optimal path for an agent at each turn, determining the move to make based on the results of numerous randomized simulations. Modification is done on MCTS corresponding to MS Pacman. [2] Uses MCTS for amazon chess and its evaluation function This paper introduces the principle of Monte Carlo Tree Search algorithm, and realizes GPU accelerated calculation of Amazon chess situation's evaluation function. [3] This paper proposes Monte Carlo tree search combined with the MTD(f) algorithm, so that the search results are not distorted by the randomness of the Monte Carlo algorithm. In order to further improve the computational efficiency game specific modifications are done such as, the TCL and the connected domain strategy which reduces the time taken. [4] This paper proposes a heuristic MCTS method for Surakarta chess. The proposed method uses heuristic knowledge to guide the search procedure heading to a better solution. [5] This paper proposes Monte Carlo tree search In hex game and run random game simulations not on the actual game position, but on a reduced equivalent board. While MCTS has shown success in various games, its direct application to chess has been relatively limited in comparison to Alpha-Beta Pruning. Chess, being a deterministic game with perfect information, presents unique challenges that differ from the stochastic and less complex nature of games like Go. The historical dominance of Alpha-Beta Pruning in chess engines raises questions about the suitability of MCTS for this particular domain. The application of MCTS to chess introduces challenges related to the large branching factor and the deterministic nature of the game. However, recent research has explored hybrid approaches that combine the strengths of both MCTS and traditional search algorithms to potentially enhance chess engine performance. These hybrid models aim to leverage the adaptability of MCTS while preserving the efficiency and strong theoretical guarantees provided by algorithms like Alpha-Beta Pruning.

Research Gap and Motivation:
While the application of Monte Carlo Tree Search (MCTS) has been well-explored in various games, its direct implementation in chess has been limited, with Alpha-Beta Pruning historically dominating the field. This raises questions about the adaptability of MCTS to the deterministic and complex nature of chess. As we have seen MCTS have been used in the basic version of chess such as amazon chess , surakarta chess with basic heuristic and game specific modification
Innovation:
In addressing this gap, our research introduces a novel approach to integrating MCTS into chess engines, seeking to capitalize on its adaptability while mitigating the challenges posed by the large branching factor and determinism inherent in chess. This heuristic model aims to combine the strengths of MCTS with Advanced heuristic and apply it to full chess game.
Our work goes beyond the existing landscape by applying heuristics that are based on the current board state, the number of chess pieces , weighted sum of different pieces, pawn structure, King safety, control of key squares, piece activity, etc. Through extensive experimentation and analysis, we aim to demonstrate the efficacy and potential advantages of this heuristic model in enhancing the performance of chess engines.

3. Methodology And Algorithms

In this part, we explain how the algorithm works. First, in Section 3.1, we describe the basics of chess. Then, in

Section 3.2, we go into detail about Monte Carlo Tree Search (MCTS). We also talk about Upper Confidence Bound in Section 3.3. In Section 3.4, we show how MCTS is used in chess, and in Section 3.5, we explain how we make the model work better.

3.1 Chess game

Chess, the ancient and timeless board game, has held a special place in the world of intellectual challenges for over a millennium. Originating in northern India around the 6th century, chess has traversed centuries and cultures, captivating players and enthusiasts alike with its blend of strategy, tactics, and intellectual rigor. It is a game of profound historical significance, having been played by monarchs and scholars, and today, it continues to be a global pastime enjoyed by millions.

At its essence, chess is a two-player, turn-based game that unfolds on a square board divided into 64 squares of alternating colors. Each player commands an army of 16 pieces, which include the king, queen, rooks, knights, bishops, and pawns. The objective is deceptively simple: to checkmate the opponent's king while safeguarding your own. Checkmate is the ultimate condition in chess, where the enemy king is threatened with capture and has no legal moves to escape capture. It signifies the end of the game and the victory of one player.

To achieve this goal, chess players must navigate a complex matrix of possible moves and countermoves. The game's rules are precise, yet the potential for strategic diversity is vast. The basic movement rules for each type of piece are as follows:

King: Moves one square in any direction.

Queen: Moves horizontally, vertically, or diagonally any number of squares.

Rook: Moves horizontally or vertically any number of squares.

Bishop: Moves diagonally any number of squares.

Knight: Moves in an L-shape pattern, consisting of two squares in one direction and one square perpendicular to it.

Pawn: Moves forward one square but captures diagonally. On its initial move, a pawn can advance two squares. Pawns also have a unique capture rule and can be promoted to any other piece upon reaching the opponent's back rank.

These rules set the stage for a complex and richly layered strategic battlefield. Players engage in a constant struggle to control the center of the board, develop their pieces, and create tactical opportunities while anticipating and responding to their opponent's moves.

The beauty of chess lies in its near-infinite combination of possibilities, allowing for creativity, foresight, and critical thinking. This paper explores the application of Monte Carlo Tree Search (MCTS) within this intricate world of chess, seeking to harness the power of artificial intelligence to navigate the complexities of the game and unlock new dimensions of strategic thinking.

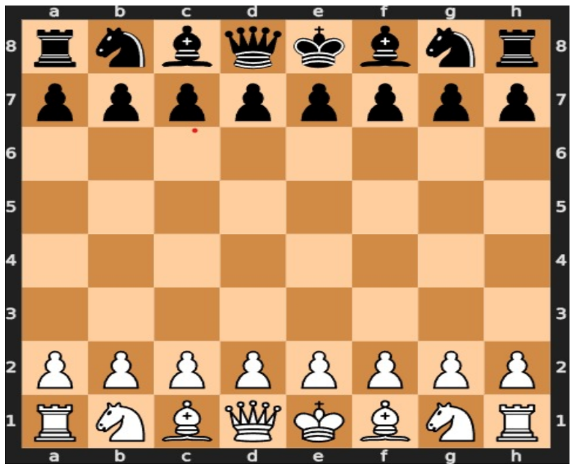

Figure 1.
Chess game Initiation

3.2 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) is a powerful algorithm for decision-making in complex domains, and it has found significant application in developing chess engines. MCTS is a heuristic-driven algorithm that uses a combination of random sampling, tree exploration, and machine learning to make informed decisions in domains with vast decision trees. In the context of chess engines, MCTS offers a dynamic and adaptive approach to navigate the game's intricate decision space.

The core working principle of MCTS can be divided into four key components: Selection, Expansion, Simulation, and Backpropagation. Here's a brief overview of each component:

By repeating the following steps iteratively, MCTS builds a search tree that guides decision-making. The algorithm focuses computational resources on more promising branches, allowing it to efficiently explore and evaluate a large decision space.

MCTS has been successfully applied in various games, including Go, chess variants, and other strategic board games. Its adaptability and ability to handle uncertainty make it a popular choice in AI research and applications beyond gaming, such as robotics and decision-making problems in various domains.

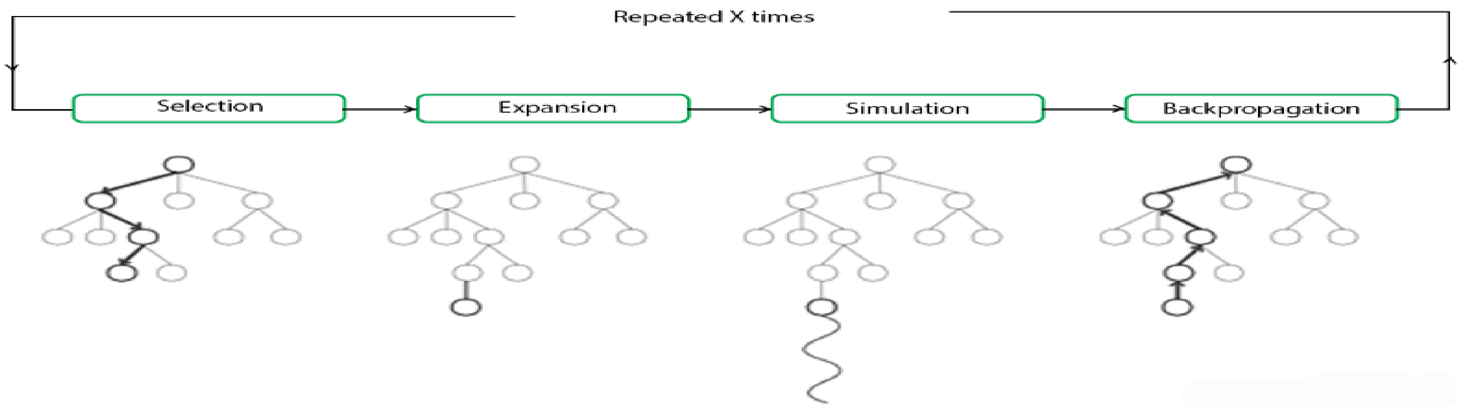

Figure 2.
Pictorial Representation of steps involved in MCTS

3.2.1 Selection:

- The process begins at the root node, representing the current game state.

- The algorithm traverses the tree by selecting child nodes in a way that balances exploration (visiting unexplored areas) and exploitation (visiting nodes that appear promising based on prior experience).

- The selection phase aims to find the most promising node (move) to explore further, often employing strategies like the Upper Confidence Bound for Trees (UCT) to guide the selection.

Pseudo code:

```
def selection(curr_node):
   max_ucb = float('-inf')  # Initialize max_ucb to negative infinity
  selected_child = None

     # Iterate through each child node of the current node
  for i in curr_node.children:
        curr_ucb = ucb_value(i)  # Calculate UCB value for the current child

     # Update max_ucb and selected_child if the current UCB value is greater
    if curr_ucb > max_ucb:
      max_ucb = curr_ucb
      selected_child = i
    return selected_child
```

3.2.2 Expansion:

- Once a node has been selected, it is expanded to generate child nodes representing possible moves.

- These child nodes correspond to potential game states resulting from different moves.

- The expansion phase adds these child nodes to the tree for further exploration.

Pseudo code:

```
def expansion(curr_node):
   # If the current node has no children, it is a leaf node, return it
  if curr_node.children.empty():
    return curr_node  # Leaf node reached

   # Initialize variables for maximum UCB value and the selected child
  max_ucb = -inf
  selected_child = None

  # Iterate over all children of the current node
  for child in curr_node.children:
  # Calculate the UCB value for the current child
    curr_ucb = ucb_value(child)

# Update the maximum UCB value and selected child
    if curr_ucb > max_ucb:
      max_ucb = curr_ucb
      selected_child = child
   # Recursively call expansion on the selected child
  return expansion(selected_child)
```

3.2.3 Simulation (Rollout)

- To evaluate the quality of a particular move, MCTS employs a random simulation or playout.

- During the simulation, the algorithm plays the game from the selected node onward by making random (or pseudo-random) moves.

- The game is played until a certain depth or termination condition is met.

Pseudo code:

```
def rollout(curr_node):
  # Check if the current game state is terminal
  if curr_node.game_over():
    # If the player won, return a score of 1 and the current
node
    if won:
      return (1, curr_node)
    # If the player lost, return a score of -1 and the current
node
    elif lose:
      return (-1, curr_node)
    # If it's a draw, return a score of 0.5 and the current nod
    else:
      return (0.5, curr_node)

  # Generate all possible child states from the current node
  curr_node.children = generate_all_states(curr_node)

  # Choose a random child node for further exploration
  random_child = random.choice(curr_node.children)

  # Recursively perform rollout on the randomly chosen
child
  return rollout(random_child)
```

3.2.4 Backpropagation:

- After the simulation, the results are propagated back up the tree to update the statistics of the nodes traversed in the selection phase.

- The results include the number of wins and the number of simulations that took place.

- The statistics are used to estimate the quality of each move and guide future selections.

Pseudo code:

```
function backpropagation(curr_node, reward):
  while curr_node.parent is not None:
    curr_node.v += reward  # Update the value of
the current node with the reward
    curr_node = curr_node.parent  # Move to the
parent node

  return curr_node  # Return the final node after
backpropagation
```

MCTS continues to iterate through these phases, gradually building a search tree that becomes more refined and provides insights into the optimal move in the given position. Over time, the algorithm allocates more simulations to the most promising moves, ultimately converging on the move believed to be the best choice.

In the context of chess engines, MCTS is typically used as part of a broader search algorithm. Chess engines combine MCTS with traditional alpha-beta pruning search methods and evaluation functions. MCTS plays an essential role in guiding move selection and decision-making in positions where the traditional search may struggle due to the game's complexity.

The adaptability, ability to explore complex decision spaces, and proficiency in decision-making under uncertainty make MCTS an intriguing and promising addition to the world of chess engines. By exploring and exploiting the possibilities offered by MCTS, chess engines aim to enhance their strategic and tactical gameplay, making them formidable opponents for human players and pushing the boundaries of artificial intelligence in chess.

3.3 Upper Confidence Bond:

In Monte Carlo Tree Search (MCTS), the UCB (Upper Confidence Bound) factor is a crucial component used for selecting child nodes during the selection phase of the algorithm. The UCB factor helps strike a balance between exploration (visiting unexplored nodes) and exploitation (selecting nodes that seem promising based on prior experience). In the context of Chess AI, understanding the UCB factor is essential for improving decision-making in complex positions.

The UCB factor is a term that contributes to the selection of child nodes in the tree. The central idea behind the UCB formula is to prioritize nodes that have a good trade-off between two aspects:

- **Exploration**: Nodes that have not been explored much or at all are prioritized. This allows the MCTS algorithm to discover promising moves or paths that haven't been examined thoroughly. In chess, this means looking for alternative strategies or lines of play that might lead to a favorable outcome.

- **Exploitation**: Nodes that have shown promising results in the past (e.g., a high win rate or good outcomes in simulations) are also favored. Exploitation means focusing on moves or lines of play that have proven to be effective in previous simulations. This is particularly important for refining the search in positions that have already been explored to some extent.

The balance between exploration and exploitation is crucial for the MCTS algorithm to effectively navigate the decision tree. The UCB factor is designed to achieve this balance, and it is typically calculated using the UCT (Upper Confidence Bound for Trees) formula, although variations exist. The UCT formula combines these two aspects as follows:

$$A_t = \text{argmax}_a \left(\underbrace{Q_t(a)}_{\text{Exploit}} + C \underbrace{\sqrt{\ln(t)/N_t(a)}}_{\text{Explore}} \right)$$

Figure 3.
UCB Equation

Qt(a) in Figure 3. represents the current estimate for action at time t. We select the action that has the highest estimated action-value plus the upper-confidence bound exploration term.

Q(A) in Figure 4. represents the current action-value estimate for action A. The brackets represent a confidence interval around Q*(A) which says that we are confident that the actual action-value of action A lies somewhere in this region.

The lower bracket is called the lower bound, and the upper bracket is the upper bound. The region between the brackets is the confidence interval which represents the uncertainty in the estimates. If the region is very small, then we become very certain that the actual value of action A is near our estimated value. On the other hand, if the region is large, then we become uncertain that the value of action A is near our estimated value.

The Upper Confidence Bound follows the principle of optimism in the face of uncertainty which implies that if we are uncertain about an action, we should optimistically assume that it is the correct action.

In practical terms, the UCT formula encourages the selection of nodes with a good balance of high quality (exploitation) and relatively unexplored potential (exploration). The choice of the UCB exploration coefficient (`C`) has a significant impact on the algorithm's behavior, and it often requires tuning to achieve the desired balance for a specific application.

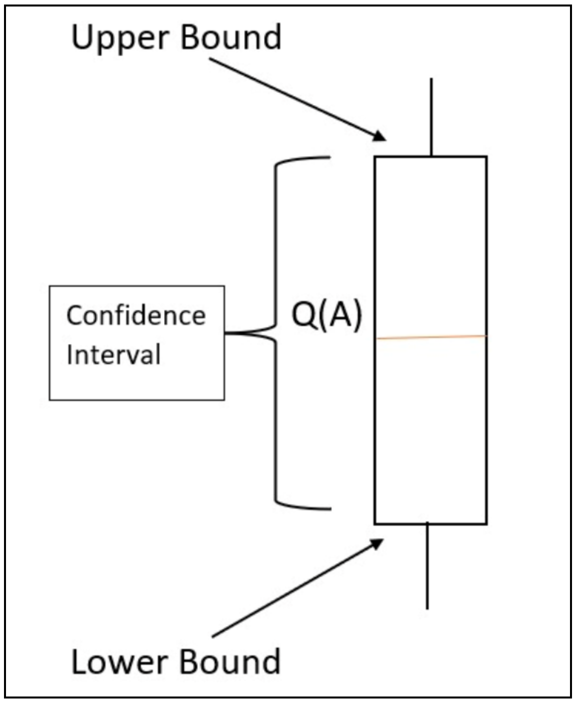

Figure 4.
Current-Value estimate for action A

For Chess AI, using the UCB factor effectively allows the MCTS algorithm to explore a broader range of moves, uncovering novel and promising strategies, while also exploiting previously successful moves. This balance is essential for enhancing the AI's decision-making capabilities in complex chess positions, ultimately improving its performance against human players and other chess engines.

3.4 MCTS in Chess:

In Monte Carlo Tree Search (MCTS) for chess or other similar board games, numbers are not assigned to the characters or pieces themselves. Instead, MCTS operates on a numerical representation of the game state, commonly known as a board position.

Working:

Board Representation: In MCTS, the chessboard is represented as an array or a matrix. Each square on the chessboard corresponds to an element in the array. The elements of the array can be indexed with numerical coordinates, such as (0,0) for the a1 square and (7,7) for the h8 square.

Piece Mapping: In the array-based representation, the different types of chess pieces (e.g., pawn, rook, knight) are assigned numerical values to distinguish them. Common piece values used in a simple array-based representation include:

- 1: Pawn
- 2: Knight
- 3: Bishop
- 4: Rook
- 5: Queen
- 6: King

For example, if you have a chessboard represented as an 8x8 array, you can assign these values to indicate the presence and type of pieces on each square.

Player Turn: Here the principle of perspective change is being used in which both the players are playing as player 1. As the same tree is drawn for both the terms only the maximum value will be observed.

Move Generation: MCTS uses the numerical board representation to generate legal moves. It identifies possible moves for the current position, which include the source square and the destination square. Each move is represented as a pair of numerical coordinates.

Node Expansion: When a node is expanded in the MCTS search tree, it involves creating new board positions, often by applying a move to the current position. The expanded nodes represent the potential outcomes of the moves.

Evaluation: During simulations or playouts, the numerical representation of the board is used to evaluate the quality of each move. The outcome of the simulations can be recorded and used to estimate the value of each move.

In summary, MCTS in chess utilizes a numerical representation of the game state (the board position) and assigns values to the different pieces to navigate and evaluate positions. The numerical representation simplifies move generation, allows for efficient exploration of the game tree, and facilitates the estimation of move values based on simulation results. While MCTS operates on numerical representations, it does not assign numbers to the actual chess piece characters (e.g., no numerical assignment like "pawn=1," "rook=2," etc., is used for the character names themselves).

3.5 Optimizing the model:

3.5.1 Integration of Heuristic Function into MCTS:

One of the primary contributions of our research lies in the innovative integration of a heuristic function into the Monte Carlo Tree Search (MCTS) framework specific to chess. Recognizing the challenges posed by the deterministic nature of the game and the need for efficient node selection and expansion, our approach augments traditional MCTS with a carefully designed heuristic function.

3.5.2 Node Selection and Expansion Enhancement:

The heuristic function plays a pivotal role in guiding the selection and expansion of nodes during the MCTS simulation phase. Unlike conventional MCTS, which relies solely on random exploration, our heuristic function strategically evaluates the potential of each move, prioritizing those with higher likelihoods of leading to favorable outcomes. This adaptive node selection process aims to address the complexities introduced by the vast search space in chess, providing a more focused exploration of promising branches.

3.5.3 Improved Algorithmic Performance:

By integrating the heuristic function into the MCTS algorithm, our research anticipates an improvement in algorithmic performance, particularly in terms of search efficiency and the ability to identify high-quality moves within the limited computational resources available. The heuristic-guided MCTS introduces a level of domain-specific intelligence, enhancing the algorithm's decision-making process and potentially outperforming traditional MCTS approaches in certain scenarios.

3.5.4 Empirical Validation and Comparative Analysis:

To validate the efficacy of our approach, extensive experimentation and comparative analysis will be conducted against conventional MCTS approach. Through a series of rigorous tests and evaluations, we aim to demonstrate the superior performance of the heuristic-guided MCTS in terms of search speed, average depth of tree, and win ratio to different chess positions.

3.5.5. Practical Implications for Chess Engines:

The incorporation of a heuristic function into MCTS holds promising implications for the practical development of chess engines. By leveraging domain-specific knowledge encoded in the heuristic, our research contributes not only to the theoretical understanding of MCTS in chess but also to the practical advancement of chess-playing algorithms that can better navigate the intricate decision space of the game.

In summary, our research introduces a novel and practical enhancement to MCTS by incorporating a heuristic function, aiming to revolutionize the capabilities of game-playing algorithms in the context of chess.

4. Experiment

The experiment is divided into two parts first to show the effect of heuristic on depth of MCTS search and time taken then will see the improvement in win ratio as compared to random rollout MCTS.

First we will consider the basic heuristic function with the consideration of piece value and pawn structure. Then we will move to advanced heuristic with the consideration of piece value and pawn structure , King safety , control of key squares , piece activity etc . The code runs on a notebook computer. The CPU is Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz and the memory is 16 GB. It runs on the windows 11 operating system. The GPU acceleration card is NVIDIA RTX 2060. The software programming development environment uses VS code with python and its libraries . In order to compare and analyze the performance of heuristic on a full chess game A total of 50 matches will take place between simple heuristic model and random model and 50 matches between Advanced heuristic model and random model .

5. Result

This paper presents a comprehensive study comparing three Monte Carlo Tree Search (MCTS) strategies for chess AI:
1 - random selection in the rollout - In the rollout phase the child is randomly selected till the terminal node is reached.
2- simple heuristic in the rollout - Children are selected with consideration of piece value and pawn structure.
3- advanced heuristic in the rollout - Incorporating different factors such as piece value , pawn structure , King safety , control of key squares , piece activity etc .

A total of 50 matches were conducted for each strategy to assess their performance in terms of average time taken, average depth reached during move selection, and the number of matches won, lost, and drawn. The results highlight the impact of different rollout strategies on the overall effectiveness of the MCTS algorithm in chess gameplay.

Average Time Taken:

- Random rollout MCTS model: 3.74seconds

-Rollout with simple Evaluation: 6.14 seconds

-Rollout with advanced Evaluation: 8.23 seconds

Average Tree Depth:

- Random rollout MCTS model: 223

-Rollout with simple Evaluation: 167

-Rollout with advanced Evaluation: 92

Comparison of time and depth for particular board states given in figures below.

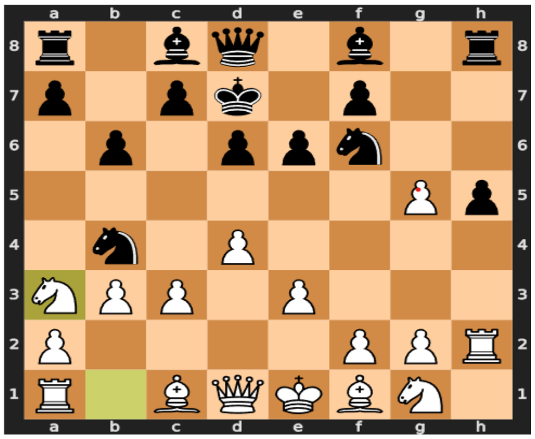
Figure 5. Board Position 1

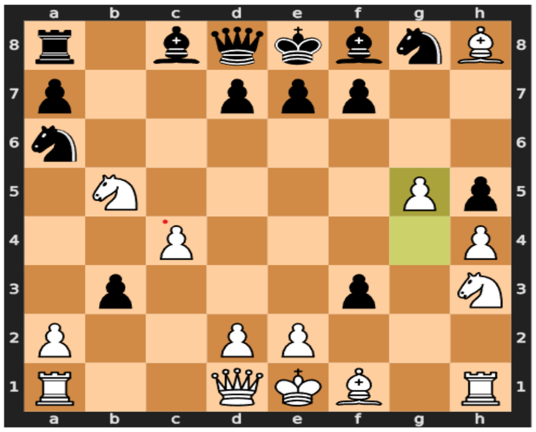
Figure 6. Board Position 2

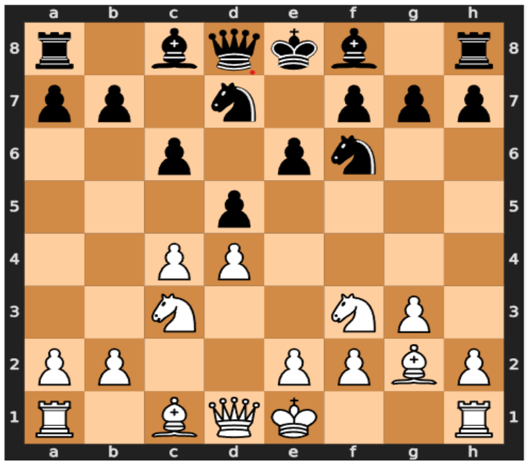
Figure 7. Board Position 3

Advanced heuristic		Simple heuristic		Random	
Time	Depth	Time	Depth	Time	Depth
2.94s	64	5.57s	142	9.24s	267
4.12s	100	6.62s	183	7.92s	208
3.95s	94	7.12s	194	10.02s	311

Match results :
These are match results of 50 matches between Simple heuristic and random and 50 matches between Advanced heuristic and Random.

Model	Won against random	Lost against random	Draw Against random
Simple evaluation	11	7	32
Advanced evaluation	14	8	28

5. Conclusion

The implementation of Monte Carlo Tree Search (MCTS) in the realm of chess has yielded promising results, offering a dynamic and adaptive approach to decision-making in a game renowned for its complexity. This research has unveiled the potential of MCTS to enhance the strategic depth and performance of chess engines, revolutionizing the landscape of chess gameplay.

The core findings of this research have demonstrated that MCTS, when integrated with traditional search algorithms and evaluation functions, has the capacity to explore complex chess positions, identify promising moves, and adapt to the ever-evolving game state. By balancing exploration and exploitation, MCTS navigates the vast decision tree of chess, uncovering novel strategies and refining established ones. In doing so, it improves the tactical and strategic capabilities of chess engines, enabling them to compete at the highest levels and challenging human grandmasters.

As we can see that this algorithm is very powerful, the only drawback of this algorithm is that memory requirement exceeds rapidly in just a few iterations. MCTS can also be used in other games like- Poker,Go and shogi.

The best thing about MCTS is that it has an open mind when it predicts a move because it is not limited by moves suggested by experts or books written by legendary players. Hence it is able to beat every player on the planet in games like chess and go.

The future of MCTS in chess is filled with exciting opportunities and untapped potential. As we conclude this research, we foresee several avenues for further improvement and innovation:

- Self Play and Reinforcement Learning: The algorithm will keep playing with itself and train the dataset which will make the model stronger.

- Online learning: The algorithm will learn from online sources where it can get access to the chess gameplays going on globally.

6. Reference

[1]Pepels, Tom, Mark HM Winands, and Marc Lanctot. "Real-time monte carlo tree search in ms pac-man." *IEEE Transactions on Computational Intelligence and AI in games* 6, no. 3 (2014): 245-257.

[2]Sun, Yikai, Dong Yuan, Ming Gao, and Penghui Zhu. "GPU Acceleration of Monte Carlo Tree Search Algorithm for Amazon chess and Its Evaluation Function." In *2022 International Conference on Artificial Intelligence, Information Processing and Cloud Computing (AIIPCC)*, pp. 434-440. IEEE, 2022.

[3]Li, Zhongzhi, Hedan Liu, Yuechao Wang, Jiankai Zuo, and Zeyuan Liu. "Application of Monte Carlo Tree Optimization Algorithm on Hex Chess." In *2020 Chinese Control And Decision Conference (CCDC)*, pp. 3538-3542. IEEE, 2020.

[4]Zuo, Guoyu, and Chenming Wu. "A heuristic Monte Carlo tree search method for surakarta chess." In *2016 Chinese Control and Decision Conference (CCDC)*, pp. 5515-5518. IEEE, 2016.

[5]Arneson, Broderick, Ryan B. Hayward, and Philip Henderson. "Monte Carlo tree search in Hex." IEEE Transactions on Computational Intelligence and AI in Games 2, no. 4 (2010): 251-258.

[6]Chen, Chih-Hung, Yen-Chi Chen, and Shun-Shii Lin. "Two-Phase-Win Strategy for Improving the AlphaZero's Strength." In 2019 2nd World Symposium on Communication Engineering (WSCE), pp. 117-121. IEEE, 2019.

[7]Scheiermann, Johannes, and Wolfgang Konen. "AlphaZero-inspired game learning: Faster training by using MCTS only at test time." IEEE Transactions on Games (2022).

[8]Pinto, Ivan Pereira, and Luciano Reis Coutinho. "Hierarchical reinforcement learning with monte carlo tree search in computer fighting game." IEEE transactions on games 11, no. 3 (2018): 290-295.

[9] Kim, Juhwan, Byeongmin Kang, and Hyungmin Cho. "SpecMCTS: Accelerating Monte Carlo Tree Search Using Speculative Tree Traversal." IEEE Access 9 (2021): 142195-142205.