



**University of  
Nottingham**

UK | CHINA | MALAYSIA

# **An Analysis of a new strategy for incorporating Opening Book in Monte-Carlo Tree Search with application to Chess**

Submitted September 2023, in partial fulfillment of  
the conditions for the award of the degree **MSc Computer Science**.

**Deepak Kumar Singh**  
**20493345**

**Supervised by Kristian Spoerer**

School of Computer Science  
University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text.

I hereby declare that I have all necessary rights and consents to publicly distribute this dissertation via the University of Nottingham's e-dissertation archive.



## **Abstract**

Monte Carlo Tree Search (MCTS) has emerged as a popular approach for developing competitive chess engines, transforming the landscape of artificial intelligence in board games. While MCTS has been lauded for its flexibility and effectiveness in various applications, it faces unique challenges during the opening phase of chess games. Traditional chess engines have leveraged opening books comprehensive databases of well-studied opening lines to navigate this complex phase. However, the stochastic nature of MCTS algorithms makes the direct application of these deterministic databases less effective. This study aims to bridge this gap by developing and evaluating three distinct MCTS engines that incorporate different strategies for utilizing opening books. Through a tournament of 200 games, the paper demonstrates that the strategic integration of opening books can significantly enhance the performance of MCTS algorithms in chess. The findings offer valuable insights into the challenges and opportunities of using opening databases to improve MCTS, thereby contributing to the advancement of artificial intelligence in board games as well as in non-gaming scenarios.



## Acknowledgements

I would like to acknowledge the help of my supervising professor Kristian Spoerer for his enormous help and guidance throughout the last term. My Master's degree would not have been a success without his feedback and ideas.



# Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background and Related Work</b>	<b>3</b>
2.1 Evolution of Game-Playing Algorithms . . . . .	3
2.2 Monte Carlo Tree Search(MCTS) . . . . .	5
2.3 Role of Opening Books . . . . .	7
2.4 Strategies to implement opening databases . . . . .	9
2.4.1 Move Pruning . . . . .	9
2.4.2 Move Biasing . . . . .	10
<b>3 Methodology</b>	<b>11</b>
3.1 MCTS for Tic-Tac-Toe . . . . .	12
3.2 Evaluation Function . . . . .	12
3.3 Policy for Simulation of MCTS . . . . .	14
3.4 Quiescence Search . . . . .	15
3.5 Search Opening Moves . . . . .	18
3.6 Three different engines . . . . .	21
3.6.1 Playing the Move Directly . . . . .	22
3.6.2 Feeding the Move to MCTS with Higher Exploration Constant . . . . .	23
<b>4 Results</b>	<b>26</b>

<b>5</b>	<b>Discussion</b>	<b>30</b>
5.1	Comparison Between Chess Engines . . . . .	30
5.2	Opening Book In MCTS With Applications Beyond Games . . . . .	31
<b>6</b>	<b>Conclusion</b>	<b>34</b>
	<b>Bibliography</b>	<b>35</b>



# List of Tables

4.1	MCTS Tic-Tac-Toe Results . . . . .	26
4.2	Comparison of MCTS and DM-MCTS in Chess . . . . .	28
4.3	Comparison of MCTS and IEC-MCTS in Chess . . . . .	28



# List of Figures

2.1	Four phases of MCTS . . . . .	7
3.1	Load NNUE file to create a Evaluation Function . . . . .	13
3.2	Policy for Simulation of MCTS . . . . .	14
3.3	Quiescence Search Code . . . . .	17
3.4	Opening Database Format . . . . .	18
3.5	Load Opening Database data into Trie . . . . .	20
3.6	Load Opening Database data into Trie . . . . .	22
3.7	Get Best Child for select node in MCTS . . . . .	24
4.1	MCTS Tic-Tac-Toe Time, Iterations Vs Win, Lose and Draw Percentage .	27



# Chapter 1

## Introduction

The application of artificial intelligence (AI) in board games has been a subject of scholarly interest and technological innovation for several decades. The complexity and strategic depth of board games like chess, Go, and shogi make them ideal testing grounds for AI algorithms. Among the algorithms that have gained significant attention in recent years is the Monte Carlo Tree Search (MCTS), an algorithm that has been lauded for its simplicity, flexibility, and effectiveness across a range of applications[25]. MCTS has been particularly successful in games with large state spaces and high branching factors, where traditional algorithms like minimax[27] and its variants often struggle. However, despite its many advantages, MCTS is not without its limitations. One of the most significant challenges facing MCTS algorithms is their performance in the opening phase of board games, particularly chess.

Chess is a game of immense complexity, with an estimated  $10^{40}$ [7] possible positions and a vast array of strategic and tactical considerations. The opening phase of the game is especially critical, as it sets the stage for the middle and endgame, often determining the trajectory and outcome of the game. Traditional chess engines have relied on extensive opening books—databases of well-studied opening lines—to navigate this complex phase. These opening books are the result of centuries of human expertise and analysis, encapsulating a wealth of strategic and tactical knowledge. However, the stochastic nature of MCTS algorithms, which rely on random simulations to evaluate positions, makes the direct application of these deterministic opening books less effective. This presents

a unique challenge and also an opportunity for innovation in the field of game-playing algorithms.

Despite the extensive research on MCTS and its application in various board games, there is a noticeable gap in the literature concerning the use of opening databases in chess to improve MCTS algorithms. While researchers have explored the concept of opening books in the context of other games like Go and Amazon[12, 16, 23], there is a lack of focused research on chess. This is surprising given the critical role that the opening phase plays in chess and the extensive body of human knowledge available in the form of opening databases. The absence of research in this area represents a significant opportunity for future work, particularly given the increasing computational power and advancements in machine learning techniques that could be leveraged to explore this topic further.

This paper aims to tackle the challenges and opportunities of using opening databases in chess to improve the MCTS algorithm. This paper will provide a comprehensive overview of the current state of research in this area. Importantly, it will highlight the glaring gap in research concerning the use of opening databases in chess to improve MCTS algorithms, setting the stage for future research that could fill this critical void. The review will explore the theoretical foundations of MCTS, the role of opening books in traditional chess engines, the unique challenges and opportunities presented by the integration of opening databases into MCTS algorithms, and potential avenues for future research.

# Chapter 2

## Background and Related Work

### 2.1 Evolution of Game-Playing Algorithms

The journey of game-playing algorithms is a fascinating tale that mirrors the broader evolution of computer science and artificial intelligence. In the early days, the focus was primarily on deterministic algorithms, such as minimax and its optimized variant, alpha-beta pruning[18]. These algorithms were straightforward but effective, relying on a tree-based search through the game's state space to find the best move. The minimax algorithm, in particular, was designed to minimize the possible loss for a worst-case scenario, making it a suitable choice for two-player zero-sum games like chess and tic-tac-toe. However, these algorithms had their limitations, especially when it came to handling games with large state spaces and high branching factors, such as Go and Shogi.

The introduction of evaluation functions[34] marked a significant advancement in game-playing algorithms. These functions allowed algorithms to assess the quality of a game state without exploring the entire tree, making it feasible to handle more complex games. However, crafting an effective evaluation function required extensive domain-specific knowledge and expertise, often encapsulated in opening books for games like chess. These opening books, which are databases of well-studied opening lines, provided a way for algorithms to navigate the complex opening phase of the game efficiently. However, the deterministic nature of these algorithms and their reliance on handcrafted evaluation functions made them less flexible and adaptable to new challenges.

The turn of the century saw the emergence of a new class of algorithms that would revolutionize the field of game-playing AI: Monte Carlo Tree Search (MCTS)[25]. Unlike its predecessors, MCTS did not require an evaluation function to assess the quality of a game state. Instead, it used statistical sampling to approximate the value of states, making it particularly useful for games where crafting an evaluation function was challenging. The stochastic nature of MCTS, which relies on random simulations to evaluate positions, offered a level of flexibility and adaptability that was previously unheard of. This made MCTS an ideal choice for a wide range of applications, from board games to real-world problems like robotics and optimization.

However, the advent of MCTS also brought new challenges, particularly concerning its performance in the opening phase of board games. While traditional algorithms could rely on opening books to navigate this phase, the stochastic nature of MCTS made the direct application of these deterministic databases less effective. This has led to a renewed interest in the role of opening databases in game-playing algorithms, but surprisingly, there is a noticeable gap in the literature concerning their use in chess to improve MCTS algorithms.

The most recent leap in game-playing algorithms came with the introduction of machine learning techniques, particularly deep learning. Algorithms like AlphaGo[36, 38] and AlphaZero[37, 30], which combine deep neural networks with MCTS, have achieved superhuman performance in a range of board games, setting new benchmarks for AI capabilities. These advancements have opened up exciting new avenues for research, particularly concerning the integration of machine learning techniques with traditional game-playing algorithms.

The evolution of game-playing algorithms has been a journey of continuous innovation and adaptation, from the deterministic algorithms of the past to the stochastic, machine learning-based algorithms of today. Each phase has brought its own set of challenges and opportunities, setting the stage for the next leap in AI capabilities.



## 2.2 Monte Carlo Tree Search(MCTS)

Monte Carlo Tree Search (MCTS) is a simulation-driven approach for gameplay and planning that has become popular in recent years. MCTS represented a paradigm shift from traditional game tree search methods like minimax. The key insight was using randomized simulation to evaluate positions rather than exhaustive calculation. This allowed MCTS to scale much better computationally. Since its inception, MCTS has been refined and applied to a diverse range of domains beyond two-player board games.

At a high level, MCTS repeatedly builds an asymmetric tree during the search process. Each node represents a state, with the root being the current position. On each iteration, four steps occur: selection, expansion, simulation, and backpropagation. Selection uses a tree policy like UCB1[20] to choose which node to explore next, balancing between exploitation and exploration.

$$UCB1 = \bar{X} + \sqrt{\frac{2 \ln n}{N}}$$

where  $\bar{X}$  is the empirical mean reward obtained from that node,  $N$  is the number of times that specific node has been visited, and  $n$  is the total number of simulations that have been run from the parent node. The first term,  $\bar{X}$ , represents the exploitation part, emphasizing nodes that have proven to be rewarding in past simulations. The second term introduces an element of exploration, encouraging the algorithm to also try nodes that haven't been visited frequently. The square root term ensures that nodes with fewer visits get a larger boost, thereby balancing exploration against exploitation. The constant 2 in the formula serves as an exploration constant, scaling the influence of the exploration term. The selected node is then expanded by generating one or more child nodes. With the new node added, a simulation or playout is run starting from that node, using a default policy to complete the game randomly. The simulation results are then backpropagated up the tree to update node statistics like visit counts and value estimates. Over many iterations, this grows the tree towards more promising areas in the search space. Fig. 2.1 illustrates the four phases of MCTS.

A major strength of MCTS is how it handles uncertainty. Traditional alpha-beta search is limited by horizon effects and inaccuracies in the position evaluation function. In contrast, MCTS accumulates robust statistics through many simulations, reducing variance and focusing on tactically critical parts of the tree. This also allows it to scale almost linearly in computation time and memory, unlike fixed-depth minimax. MCTS also prevents over-exploration by expanding the tree one node at a time. Additionally, the separation of policies for tree descent and simulation provides flexibility.

Since 2006, the most impactful improvements to MCTS have optimized its four steps. In selection, UCB1 was replaced by other bandit algorithms like UCT that scale better for games with vast branching factors like Go.

$$UCT = \frac{w_i}{n_i} + C \sqrt{\frac{\ln N_i}{n_i}}$$

The UCT[25] formula combines these two aspects. In the formula  $\frac{w_i}{n_i}$  represents the exploitation term, where  $w_i$  is the number of simulated wins and  $n_i$  is the number of times the move has been simulated. This term alone would make the algorithm always choose the move with the highest win rate so far. To balance this, the formula adds an exploration term  $C \sqrt{\frac{\ln N_i}{n_i}}$ , where  $C$  is a constant determining the level of exploration,  $N_i$  is the total number of simulations, and  $n_i$  is the number of times the specific move has been simulated. The square root and logarithm functions ensure that the exploration term diminishes over time but does so at a slower rate than the exploitation term, allowing for a balanced search. In expansion, adding domain knowledge guides growth towards more promising nodes. For simulation, better default policies have incorporated priors and machine learning. Finally, enhancements like virtual loss avoid re-exploring sub-optimal nodes during backpropagation.

MCTS powered a revolution in computer Go, defeating top human players for the first time in 2016. It has also been applied to perfect information games like Amazons[24] and Hex[10]. For imperfect information games[17, 19], MCTS integrates well with methods like information set Monte Carlo Tree Search. MCTS is now commonly combined with neural networks, using policy and value networks to guide the different phases. This has fueled

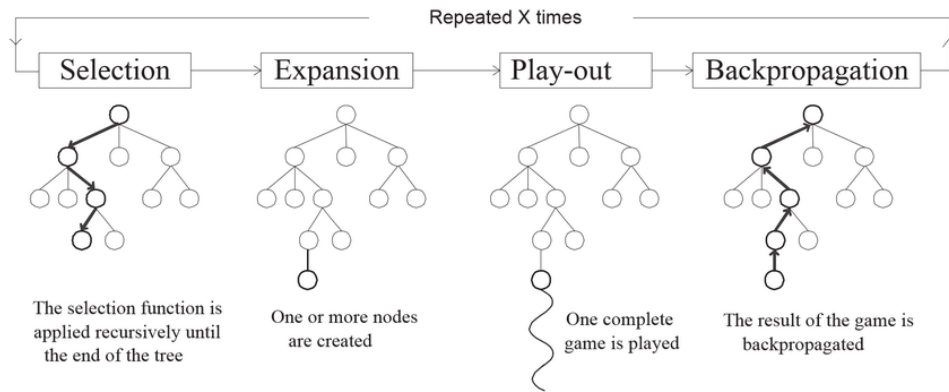


Figure 2.1: Four phases of MCTS

superhuman performance in poker, chess, shogi and more. Beyond games, MCTS has been used successfully for optimization[35, 32, 11, 29], planning and scheduling[8, 9, 28], robotics[14, 39] and more[40].

Monte Carlo tree search represents a landmark technique in AI and game theory. Its central ideas have proven widely effective and continue to drive advances augmented by deep reinforcement learning. MCTS provides a general framework for tackling decision-making under uncertainty.

## 2.3 Role of Opening Books

The opening phase in board games like chess is a critical juncture that can significantly influence the trajectory and outcome of the game. In chess, the opening moves are not just a series of arbitrary placements; they are the result of centuries of human expertise, strategic depth, and tactical nuance. The importance of the opening phase is so pronounced that it has led to the development of opening books—comprehensive databases of well-studied opening lines and sequences. These opening books serve as a reservoir of collective human wisdom, encapsulating a wealth of strategic and tactical knowledge that has been refined over generations.

In the realm of game-playing algorithms, opening books have traditionally played a pivotal role, especially in deterministic algorithms like minimax and alpha-beta pruning. These algorithms rely on a tree-based search to explore the game’s state space, and the

computational cost of this search can be prohibitively high, particularly in games with large state spaces and high branching factors. Opening books offer a computational shortcut in this context. By providing a set of pre-calculated, well-studied moves for the opening phase, opening books allow the algorithm to bypass the computationally expensive task of searching through the game tree. This not only speeds up the algorithm but also ensures that it adheres to well-established principles and strategies, thereby increasing its chances of success.

However, the deterministic nature of traditional algorithms made it relatively straightforward to integrate opening books into their architecture. These algorithms would simply query the opening book database for a given position and retrieve the corresponding best move or sequence of moves. This deterministic approach was effective but lacked the flexibility to adapt to new or unexpected situations. It also made the algorithms highly dependent on the quality and comprehensiveness of the opening book, limiting their performance to the extent of human expertise encapsulated in the database.

The advent of Monte Carlo Tree Search (MCTS) introduced a new set of challenges and opportunities concerning the role of opening books. Unlike deterministic algorithms, MCTS relies on statistical sampling and random simulations to evaluate game states. This stochastic nature makes the direct application of deterministic opening books less effective and less straightforward. The algorithm's inherent randomness can lead to situations where the moves suggested by the opening book are not adequately explored or are even ignored, thereby negating the advantage that the opening book is supposed to provide.

Despite these challenges, the integration of opening books into MCTS algorithms remains a topic of significant interest and potential[16, 23, 22, 12]. The stochastic nature of MCTS offers a level of flexibility and adaptability that deterministic algorithms lack, suggesting that a successful integration could yield a game-playing algorithm that combines the best of both worlds—the computational efficiency and strategic depth of opening books with the flexibility and adaptability of MCTS.

Opening books have played a critical role in the development and performance of game-playing algorithms. Their importance is not just a historical artefact but a com-

putational necessity, especially in complex games like chess. The advent of MCTS has complicated but also enriched the landscape, offering new challenges and opportunities for leveraging the computational advantage that opening books provides.

## 2.4 Strategies to implement opening databases

The paper [12] explore the intricacies of enhancing the performance of Monte-Carlo Tree Search (MCTS) algorithms in the game of Go through the use of opening books. The paper identifies that MCTS, while powerful, often struggles in the opening phase of Go, a game characterized by a high branching factor and the delayed consequences of moves. To mitigate these challenges, the authors propose an "active opening book application," a methodology that integrates human expertise captured in databases of high-level game records into the MCTS framework. This approach aims to guide the search algorithm towards well-established moves while filtering out moves that may appear locally optimal but are globally problematic.

### 2.4.1 Move Pruning

One of the key techniques proposed in the paper is "move pruning"[12] which involves eliminating certain moves from consideration during the selection phase of the MCTS loop. The idea is to reduce the branching factor of the game tree, thereby allowing the algorithm to focus its computational resources on exploring more promising moves. In the context of Go, this is particularly important given the game's enormous state space and the computational limitations of MCTS in thoroughly exploring it. The authors suggest that move pruning can be guided by an opening book constructed from games played by professional or highly-ranked amateur players. By pruning moves that are not part of well-established opening sequences, the algorithm can avoid wasting computational resources on less promising lines of play.

### 2.4.2 Move Biasing

The second technique proposed is "move biasing" [\[12\]](#) which involves adjusting the selection policy of MCTS to favour moves that are part of established opening sequences. Unlike move pruning, which completely eliminates certain moves from consideration, move biasing simply adjusts the probabilities with which different moves are selected. This allows the algorithm to explore a broader range of moves while still favouring those that are more likely to lead to favourable outcomes based on human expertise. The authors suggest that move biasing can be particularly useful in situations where the opening book contains multiple moves that are roughly equally good. By biasing the selection towards these moves, the algorithm can explore multiple promising lines of play without committing to any single one.

# Chapter 3

## Methodology

The Monte Carlo Tree Search (MCTS) algorithm has traditionally used random simulations (also known as playouts) during the Simulation step. While this approach is computationally less demanding and easy to implement, it comes with its own set of challenges, especially in complex domains like chess. Chess, with its immense complexity, often has positions that are critical and require precise evaluation. Random simulations may not capture the nuances of such positions, leading to less accurate assessments. A random simulation requires playing out moves until a terminal state (checkmate, stalemate, etc.) is reached. In chess, this could take many moves and thus can be computationally expensive. Chess is not just about tactics but also about strategy. Positional nuances, pawn structures, open files, and other strategic elements are typically beyond the scope of random simulations.

While MCTS with random simulations can be effective for some games, the complexity and depth of chess make a knowledgeable evaluation function a far superior choice. The evaluation function is used to estimate the value or quality of a game state without requiring to explore all possible future moves[34]. The more accurate the evaluation function, the better the MCTS algorithm will perform. Integrating such an evaluation function into the MCTS framework significantly boost the playing strength and decision-making ability of a chess engine.

### 3.1 MCTS for Tic-Tac-Toe

To understand the performance and behaviour of the Monte Carlo Tree Search (MCTS) algorithm in a game of Tic-Tac-Toe, we implemented the algorithm under five different time controls. The time control specifies the maximum amount of time the algorithm has to make each move. The five chosen time controls were 1 second, 0.1 seconds, 0.01 seconds, 0.001 seconds, and 0.0001 seconds per move. The objective was to examine how varying time constraints affect the decision-making quality, computational efficiency, and overall success of the MCTS algorithm.

A standard 3x3 Tic-Tac-Toe board was initialized as the starting state. The MCTS algorithm was set to play from this initial position. The MCTS engine was designed to explore the game tree using the UCT (Upper Confidence Bound applied to Trees) formula for its selection policy.

The choice of Tic-Tac-Toe as the testbed was due to its simplicity and finite game space, allowing for a more controlled evaluation of the MCTS algorithm. The different time controls were selected to represent a range of computational budgets that might be encountered in practical applications.

By systematically studying the MCTS algorithm under different time constraints aims to provide a comprehensive understanding of the trade-offs between computational resources and decision-making quality. The insights gained can be useful for tuning MCTS in more complex games and applications.

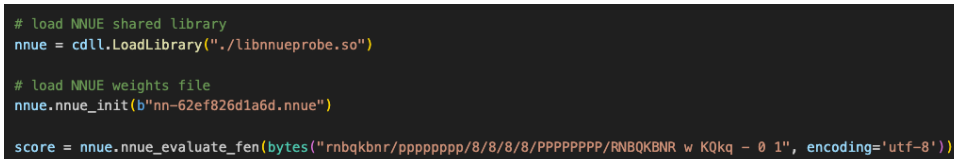
### 3.2 Evaluation Function

In this case, Stockfish’s NNUE(Efficiently Updateable Neural Network)[1, 2] is used to create an evaluation function. The term ”Efficiently Updateable Neural Network” refers to a specific architecture and design of neural networks that allows for quick and efficient updates of evaluations as the input state changes incrementally. The concept is particularly useful in applications where the environment changes step by step and you want to update the neural network’s output based on this new information without having to



perform a complete forward pass through the network each time. It evaluates new positions by incrementally updating evaluations from previous positions, which is crucial for a chess engine that needs to evaluate millions of positions.

Before integrating Stockfish NNUE into the MCTS, it is essential to pre-process the NNUE files to make them compatible with the existing engine. This includes parsing the files and extracting the neural network weights, biases, and other model parameters. Fig. 3.1 shows code to parse the NNUE file and extract model parameters.



```
# load NNUE shared library
nnue = cdll.LoadLibrary("./libnnueprobe.so")

# load NNUE weights file
nnue.nnue_init(b"nn-62ef826d1a6d.nnue")

score = nnue.nnue_evaluate_fen(bytes("rnbqkbnr/pppppppp/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1", encoding='utf-8'))
```

Figure 3.1: Load NNUE file to create a Evaluation Function

The provided Python code demonstrates how to interact with a shared library (in this case, **libnnueprobe.so**) using Python’s **ctypes** library. The shared library contains functions for handling NNUE (Efficiently Updateable Neural Network) functionality in a chess engine.

The first line imports all symbols from the **ctypes** library. The **ctypes** library allows you to call functions in dynamic link shared libraries and has support for calling functions in shared libraries written in C and other C-compatible languages.

Here, the **cdll.LoadLibrary** function is used to load the shared library (.so file) named **libnnueprobe.so** from the current directory. The variable **nnue** now holds a CDLL object, which can be used to call functions defined in the shared library.

In the last line, the **nnue.init** function from the **libnnueprobe.so** library is called to initialize the NNUE weights. The argument passed is a bytes object **nn-62ef826d1a6d.nnue**[3], which is the name of the file containing the pre-trained NNUE weights.

The function **nnue.evaluate\_fen** is called to evaluate a chess position given in FEN format. The FEN string is converted to a byte object with UTF-8 encoding, as the function expects a byte string. The function returns an evaluation score for the given position, which is stored in the **score** variable. **nnue.evaluate\_fen** function is used as the evaluation function but this evaluation function is not the simulation part of the MCTS

because there are other things which need to be taken care of. So, `nnue_evaluate_fen` is only a part of the simulation and not the whole policy.

### 3.3 Policy for Simulation of MCTS

Fig. 3.2 shows the whole policy used for the simulation step. The Python code defines a function, `nnue_policy(state)`, which performs a positional evaluation of a chess board. This function takes in a state object, which contains the current board state as one of its attributes, and returns a numerical evaluation score. The function uses the NNUE (Efficiently Updateable Neural Network) model to provide a more nuanced evaluation than traditional methods, but it also incorporates various game-specific conditions, such as checkmate, stalemate, and material balance, for a comprehensive assessment.

```
# NNUE evaluation
def nnue_policy(state):
    # case black is checkmated
    if state.board.is_checkmate(): return -10000

    # case stale mate
    elif state.board.is_stalemate(): return 0

    # case draw
    elif state.board.can_claim_draw(): return 0

    # case in insufficient material
    elif state.board.is_insufficient_material(): return 0

    # get NNUE evaluation score
    score = nnue.nnue_evaluate_fen(bytes(state.board.fen(), encoding='utf-8'))

    # on material imbalance
    if get_material_score(state):
        # get quiescence score
        quiescence_score = quiescence(state, -10000, 10000)

        # use either direct static evaluation score or quiescence score
        # depending on which one is greater
        return max(score, quiescence_score)

    # on material balance
    else:
        # use static evaluation score
        return score
```

Figure 3.2: Policy for Simulation of MCTS

In the first part of the function, several game-ending conditions are checked:

- If the board state represents a checkmate situation for black, the function immediately returns a score of -10,000, signifying an overwhelming advantage for white.
- In cases of stalemate or draw claims, a score of zero is returned, representing a neutral or equal position.

- If there is insufficient material to continue the game, another neutral score of zero is returned.

The second part of the function uses the NNUE model to evaluate the board state. The board's FEN (Forsyth-Edwards Notation) string is converted to a byte string and then passed to the function `nnue.nnue_evaluate_fen()`, which returns an evaluation score. This score quantifies the material and positional advantages on the board, giving more insight into which side stands better in a given position.

In the third section, the function considers material imbalance on the board using `get_material_score(state)`. If there is a material imbalance, the function proceeds to calculate a quiescence score. Quiescence search[13, 31, 21] is a technique used to evaluate positions at the end of tactical sequences more accurately. The function `quiescence(state, -10000, 10000)` presumably conducts this search and returns the quiescence score, which can vary drastically from the static evaluation if there are pending tactical threats or opportunities.

Lastly, the function returns the higher of the two scores—either the static evaluation score from the NNUE model or the dynamic quiescence score—when there is material imbalance. This implies that the function gives priority to tactical considerations over static position evaluations in such scenarios. If the material is balanced, the NNUE evaluation score is returned directly.

The `nnue_policy(state)` function is a sophisticated evaluation mechanism that integrates both machine learning-based static evaluations and traditional chess heuristics. By incorporating different layers of complexity, from game-ending conditions to material imbalances and tactical opportunities, it aims to provide a well-rounded evaluation of chess positions.

## 3.4 Quiescence Search

The quiescence search is a recursive function which is employed in chess engines to address the horizon effect, a phenomenon where the conventional minimax algorithm might prematurely stop its search just before a significant change in position evaluation, like a

capture or a check. When a typical search cuts off, there might be potential threats that are left unevaluated. Quiescence search focuses on these 'noisy' positions, diving deeper specifically in scenarios involving tactical sequences like captures and checks.

The primary aim of the quiescence search is to stabilize evaluations by ensuring that only 'quiet' positions, where no immediate tactics or threats exist, are evaluated for their static worth. This means that instead of evaluating a position where a piece is left under threat of capture, the search delves deeper until a stable position is reached, where no such immediate captures or checks are pending.

Despite adding depth to the search, quiescence search is computationally efficient. It doesn't assess every possible move but focuses on only those that might significantly alter the material or tactical balance, ensuring that the engine doesn't waste resources on inconsequential positions. In essence, quiescence search refines the evaluation process, allowing chess engines to make better-informed decisions in complex positions.

Fig. 3.3 shows the python code for quiescence search. The function **quiescence(state, alpha, beta)** takes three arguments: **state**, which encapsulates the current chess board state; and **alpha** and **beta**, which are used for alpha-beta pruning to optimize the search process. The function returns an evaluation score that takes into consideration the tactical implications of the board state.

The first part of the function performs a static evaluation using the NNUE model. This evaluation, stored in **stand\_pat**, serves as a baseline score for the current position. If this score is greater than or equal to **beta**, the function returns **beta**, effectively pruning this branch of the search tree as it won't affect the final result. Likewise, if **stand\_pat** is greater than **alpha**, the function updates **alpha** to be **stand\_pat**. These steps follow the principles of alpha-beta pruning to cut down on unnecessary computations.

Next, the function enters a loop that iterates through all legal moves from the current board state. It filters out moves that are not captured, focusing only on moves that could potentially change the material balance on the board. For each capturing move, the function simulates the move by updating the board (**state.board.push(chess.Move.from\_uci(str(move)))**)[5] and then makes a recursive call to **quiescence**, inverting the signs

```

# quiescence search
def quiescence(state, alpha, beta):
    # static evaluation score
    stand_pat = nnue.nnue_evaluate_fen(bytes(state.board.fen(), encoding='utf-8'))

    if stand_pat >= beta:
        return beta

    if alpha < stand_pat:
        alpha = stand_pat

    # loop over legal moves
    for move in state.board.legal_moves:
        # pick up only captures
        if state.board.is_capture(move):
            # make move on board
            state.board.push(chess.Move.from_uci(str(move)))

            # recursive quiescence call
            score = -quiescence(state, -beta, -alpha)

            # take move back (restore board position)
            state.board.pop()

            if score >= beta:
                return beta

            if score > alpha:
                alpha = score

    return alpha

```

Figure 3.3: Quiescence Search Code

of **alpha** and **beta** and negating the returned score (standard technique in minimax-based algorithms).

After the recursive call, the function restores the original board state (**state.board.pop()**) and continues the alpha-beta pruning process. If the returned score (**score**) is greater than or equal to **beta**, the function returns **beta**. If the score is greater than **alpha**, **alpha** is updated to be **score**. Finally, the function returns **alpha**, which represents the best score that the maximizing player is assured of in the considered moves.

In MCTS, the algorithm performs four main steps iteratively: selection, expansion, simulation, and backpropagation. Instead of simulation all other section works in the same way discussed in section 2. After running MCTS for a predefined time of 5 seconds it chooses a move to play. Selecting the move corresponding to the most robust child[15]—i.e., the one with the highest visit count—is the strategy used. The rationale is that if a particular move has been explored more frequently, the value estimate is likely to be more reliable. High visit counts generally imply that the move is part of a promising sequence of moves that lead to favourable outcomes more often than other moves.

### 3.5 Search Opening Moves

Searching for opening moves in databases often involves querying a structured set of data to retrieve sequences of moves that lead to specific positions or opening names. The type of database and the technology used can vary greatly, but the general methods for conducting such searches remain similar. Any position reached in chess needs to be searched in the database if there is an opening move available in the database. A database which has the list of all the well-known opening sequences is used in this experiment.

Fig. 3.4 shows the format of the opening databases[4]. It utilizes a Trie[6] data structure to efficiently store and search chess opening lines. The primary goal is to index various chess openings so that retrieval can be performed quickly. The code is broken down into several parts: the definition of a `TrieNode` class, reading multiple TSV (Tab-Separated Values) files containing chess opening data, combining these datasets, and finally building the Trie from this data. At the end, the Trie is serialized into a file using the `pickle` library for later use.

```

1 eco name pgn
2 A00 Amar Gambit 1. Nh3 d5 2. g3 e5 3. f4 Bxh3 4. Bxh3 exf4
3 A00 Amar Opening 1. Nh3
4 A00 Amar Opening: Gent Gambit 1. Nh3 d5 2. g3 e5 3. f4 Bxh3 4. Bxh3 exf4 5. 0-0 fxg3 6. hxg3
5 A00 Amar Opening: Paris Gambit 1. Nh3 d5 2. g3 e5 3. f4
6 A00 Amsterdam Attack 1. e3 e5 2. c4 d6 3. Nc3 Nc6 4. b3 Nf6
7 A00 Anderssen's Opening 1. a3
8 A00 Anderssen's Opening: Polish Gambit 1. a3 a5 2. b4
9 A00 Barnes Opening 1. f3
10 A00 Barnes Opening: Fool's Mate 1. f3 e5 2. g4 Qh4#
11 A00 Barnes Opening: Gedult Gambit 1. f3 d5 2. e4 g6 3. d4 dxe4 4. c3
12 A00 Barnes Opening: Gedult Gambit 1. f3 f5 2. e4 fxe4 3. Nc3
13 A00 Barnes Opening: Hammerschlag 1. f3 e5 2. Kf2
14 A00 Clemenz Opening 1. h3
15 A00 Clemenz Opening: Spike Lee Gambit 1. h3 h5 2. g4
16 A00 Crab Opening 1. a4 e5 2. h4
17 A00 Creepy Crawly Formation: Classical Defense 1. h3 d5 2. a3 e5
18 A00 Formation: Hippopotamus Attack 1. a3 e5 2. b3 d5 3. c3 Nf6 4. d3 Nc6 5. e3 Bd6 6. f3 0-0 7. g3
19 A00 Formation: Shy Attack 1. a3 e5 2. g3 d5 3. Bg2 Nf6 4. d3 Nc6 5. Nd2 Bd6 6. e3 0-0 7. h3
20 A00 Global Opening 1. h3 e5 2. a3
21 A00 Grob Opening 1. g4
22 A00 Grob Opening: Alessi Gambit 1. g4 f5
23 A00 Grob Opening: Double Grob 1. g4 g5

```

Figure 3.4: Opening Database Format

Firstly, the **TrieNode** class serves as the fundamental unit for building the Trie as shown in Fig. 3.5. Each `TrieNode` object contains a dictionary named **children**, which holds the subsequent moves from the current node. The **OpeningDepth** attribute denotes the depth of the opening line associated with this node. The **isEndOfOpening** attribute signifies whether the node represents the last move of an opening line.

The **insert** method adds a sequence of moves into the Trie, starting from the root

node. It iterates through each move, checks if the move already exists as a child, and either creates a new node or updates the existing one. The depth of the most extended opening line containing this move is stored in **OpeningDepth**.

The code then reads chess opening lines from five TSV files named "a.tsv", "b.tsv", "c.tsv", "d.tsv", and "e.tsv". These files are loaded into pandas DataFrames (**df\_1** to **df\_5**). Each DataFrame is reset to ensure proper indexing. After this, all the DataFrames are concatenated into a single DataFrame **df**, which is also reset to fix its index.

A root TrieNode is instantiated, and the code iterates through each row in the combined DataFrame **df**. Each row contains a chess opening sequence in the fifth column (indexed as **row[4]**). This sequence is first split into individual moves. The code then passes this list of moves to the **insert** method of the root TrieNode to populate the Trie. Each sequence of moves thereby becomes a path in the Trie, starting from the root and ending at a leaf node.

Finally, the populated Trie, rooted at **root**, is serialized into a binary file named 'opening.pkl' using the **pickle** library. This allows for quick and easy loading of the Trie structure for future queries, thereby saving the time that would be needed to rebuild the Trie from scratch.

By using a Trie for storing chess openings, the code enables efficient storage and quick retrieval. Each path in the Trie corresponds to a unique chess opening, making it a suitable data structure for this purpose. The Trie's branching factor is limited by the number of possible legal moves in a position, making the Trie both space and time-efficient. Moreover, by storing the maximum depth of any opening line passing through each node, additional information is readily available for further analysis or for aiding decision-making in a chess engine. All this information is then saved into a binary file for future use, making the system both efficient and portable.

Now MCTS class has a member function **searchOpening** which is called in the constructor at the time of object creation. The **searchOpening** function aims to find the particular sequence of chess moves within a Trie data structure that serves as a database of chess openings. This Trie is rooted at the variable **opening\_root**. The function is

```

class TrieNode:

    # Trie node class
    def __init__(self, depth=0, isLast=False):
        self.children = {}

        # isEndOfWord is True if node represent the end of the word
        self.OpeningDepth = depth
        self.isEndOfOpening = isLast

    def insert(self, move):
        ro = self
        for i in move:
            if i not in ro.children:
                ro.children[i] = TrieNode(len(move))
            elif len(move) > ro.children[i].OpeningDepth:
                ro.children[i].OpeningDepth = len(move)
            ro = ro.children[i]

df_1 = pd.read_csv("a.tsv", sep='\t')
df_1 = df_1.reset_index()

df_2 = pd.read_csv("b.tsv", sep='\t')
df_2 = df_2.reset_index()

df_3 = pd.read_csv("c.tsv", sep='\t')
df_3 = df_3.reset_index()

df_4 = pd.read_csv("d.tsv", sep='\t')
df_4 = df_4.reset_index()

df_5 = pd.read_csv("e.tsv", sep='\t')
df_5 = df_5.reset_index()

df = pd.concat([df_1, df_2, df_3, df_4, df_5])
df = df.reset_index()

root = TrieNode()
for x, row in df.iterrows():
    s = row[4].split(" ")
    k = []
    for y in s[1:]:
        y = y.split(" ")
        k.append(y[0])
        if len(y) > 1:
            k.append(y[1])

    root.insert(k)

```

Figure 3.5: Load Opening Database data into Trie

designed to work within a class, as indicated by the **self** parameter, which allows it to access and potentially modify the object's state.

The function **searchOpening** as shown in Fig. 3.6 begins by initializing an empty



chess board using the **chess.Board()** constructor from the Python chess library. It also initializes an empty list, **san\_list**, which is intended to hold the Standard Algebraic Notation (SAN) form of each move. The moves passed to the function are expected to be in Universal Chess Interface (UCI) format, and are contained in the **moves** parameter.

The first **for** loop iterates over each move in **moves**. For each move, it converts the UCI format to a move object using **chess.Move.from\_uci(x)** and then translates this move object to its SAN equivalent using the **board.san(move)**. This SAN representation of the move is appended to **san**, and the board state is updated with **board.push(move)** to reflect the new move.

Once all moves are converted to their SAN forms and stored in **san\_list**, the function sets **root** to point to **opening\_root**, essentially starting at the top of the Trie. The second **for** loop goes through each move in **san\_list** to traverse the Trie. Inside the loop, a try-except block is used to navigate through the Trie. If the move exists as a child of the current node (**root.children[y]**), **root** is updated to point to this child node, effectively descending one level down the Trie. If the move does not exist, the try-except block will catch this, set a **flag** attribute of the object to 1 (which is used to avoid searching the subsequent positions in the trie because it cannot be found in trie anymore), and return **None**.

By the end of this traversal, if all moves in **san\_list** are found in the Trie, the **root** will point to the Trie node that corresponds to the last move in the input sequence. This node is then returned, thereby providing information related to this specific sequence of opening moves. This is essentially a look-up operation and is highly efficient, taking advantage of the Trie data structure to quickly locate a sequence of moves in the database of chess openings.

## 3.6 Three different engines

In the context of a chess engine, encountering a move that is part of a well-known opening database creates an interesting fork in the road: the engine can either play the move directly or feed it into a Monte Carlo Tree Search (MCTS) algorithm for further evaluation.

```
def searchOpening(self, moves):
    board = chess.Board()
    san_list = []
    # print(moves)
    for x in moves:
        move = chess.Move.from_uci(x)
        san_list.append(board.san(move))
        board.push(move)

    root = opening_root

    for y in san_list:
        try:
            if root.children[y]:
                root = root.children[y]
        except:
            self.flag=1
            return None

    return root
```

Figure 3.6: Load Opening Database data into Trie

Each approach has its merits and demerits, and their relative effectiveness may depend on various factors like computational resources, time constraints, and the engine’s overall design philosophy.

### 3.6.1 Playing the Move Directly

The most straightforward approach is to play the recognized move immediately. Doing so has several advantages. First, it is computationally inexpensive. Well-known openings have been analyzed extensively, both manually by grandmasters and automatically by powerful engines. By playing a move from the database, the engine effectively leverages centuries of accumulated chess knowledge without having to spend any computational resources. This can be particularly advantageous in rapid or blitz time controls, where saving time on the clock is crucial.

Second, playing a recognized move often leads to familiar positions that have been analyzed deeply. This helps the engine navigate the complexities of the opening phase

without making critical errors that could jeopardize its position. Established opening moves often aim to control the centre of the board, develop pieces efficiently, and ensure the safety of the king, among other strategic objectives.

However, this approach has its drawbacks too. It may lack creativity and surprise, making the engine's play predictable. More advanced opponents with their own extensive databases may quickly recognize the sequence and deploy counter-strategies, potentially gaining the upper hand.

### **3.6.2 Feeding the Move to MCTS with Higher Exploration Constant**

The alternative is to feed the recognized move into an MCTS algorithm but modify the exploration constant to favour this particular move. In essence, this approach hybridizes database knowledge and computational creativity. The MCTS algorithm generally balances exploration (trying new moves) and exploitation (playing known good moves). By increasing the exploration constant for the recognized move, the algorithm is biased to explore variations emanating from this move more deeply, while still considering other options.

The main advantage here is adaptability. Although the recognized move is given preference, MCTS still evaluates other moves in the position. This can be particularly useful if the opponent's moves indicate a departure from established lines or an attempt to set a trap. By exploring other options, the engine retains the flexibility to adapt its strategy based on the evolving board state.

Moreover, the adjusted MCTS can discover nuances in a position that are not captured by the opening database. The MCTS algorithm simulates games for each possible move, accumulating statistics that represent the estimated value of those moves. Even a well-known move may turn out to be less effective in a specific board state once this deeper level of analysis is conducted.

However, this approach is computationally expensive. Even with a higher exploration constant for the known move, MCTS will still explore other moves, consuming more

computational resources and taking more time. This may be less suitable for faster time controls or if computational resources are limited.

The choice between these two strategies can be seen as a trade-off between computational efficiency and strategic depth. Playing the recognized move directly is fast and leverages well-established chess wisdom but may lack flexibility. On the other hand, using MCTS with an increased exploration constant for the recognized move combines the best of both worlds but at a computational cost. The ideal strategy may involve dynamically choosing between these options based on the specific circumstances of the game, such as the time available and the observed behaviour of the opponent.

```
def getBestChild(self, node, explorationValue, root=None):
    bestValue = float("-inf")
    bestNodes = []
    for child in node.children.values():
        k = 1
        if self.flag==0:
            try:
                board = node.state.board
                move = chess.Move.from_uci(str(child.state.board.peek()))

                if board.san(move) in root.children:
                    k = root.children[board.san(move)].OpeningDepth
            except:
                pass
        nodeValue = child.totalReward / (1 + child.numVisits) + explorationValue * k * math.sqrt((1 + node.numVisits) / (1 + child.numVisits))
        if explorationValue==0:
            nodeValue = child.numVisits

        if nodeValue > bestValue:
            bestValue = nodeValue
            bestNodes = [child]
        elif nodeValue == bestValue:
            bestNodes.append(child)
    return random.choice(bestNodes)
```

Figure 3.7: Get Best Child for select node in MCTS

Image 6 shows the code for how the nodes are selected when they are found in the opening databases. The function **getBestChild** as shown in Fig. 3.7 is designed to find the most promising child node of a given node in the context of the Monte Carlo Tree Search (MCTS) algorithm and it uses the given formula. This function is crucial for deciding the next move in a game of chess. It takes three parameters: **node**, which is the parent node whose children are to be evaluated; **explorationValue**, a parameter that balances exploitation and exploration in MCTS; and **root**, is the root of the Trie containing opening moves.

$$UCT = \frac{w_i}{1 + n_i} + C * k * \sqrt{\frac{1 + N_i}{1 + n_i}}$$

The function starts by initializing **bestValue** to negative infinity and **bestNodes** to an empty list. These will ultimately hold the highest value calculated for any child node and the corresponding best child nodes.

The variable **k** is initialized as 1 and is used as a multiplier for the exploration constant. If a flag (**self.flag**) is set to 0, the code attempts to check if the move associated with the current child node is found in the opening database (**root.children**). If the move is found, **k** is updated to the depth of that opening in the database (**root.children[board.san(move)].Opening**). This essentially increases the exploration constant for this particular move, as **k** will be greater than 1.

The value of each child node is then calculated using the formula for the Upper Confidence Bound applied to Trees (UCT). This formula comprises two parts: the average reward (**child.totalReward / (1 + child.numVisits)**) represents the exploitation term, and the remaining part of the expression represents the exploration term. The exploration term is multiplied by **k**, which means that the exploration constant is effectively increased for moves found in the opening database. This encourages the MCTS algorithm to explore these moves more.

If **explorationValue** is set to 0, **nodeValue** is set as the number of visits to the child node, essentially disabling the exploration term. This is useful in the final stages of the MCTS where exploitation is favored over exploration.

Finally, the function checks whether the calculated **nodeValue** for each child is higher than the current **bestValue**. If so, **bestValue** is updated and **bestNodes** is reset to contain just this child. If **nodeValue** is equal to **bestValue**, the child is appended to **bestNodes**. In the end, one of the best nodes is returned at random.

This function not only serves its typical role in MCTS of determining the best child node to explore next but also adapts its behaviour when a move is found in the opening database by effectively increasing the exploration constant for that move. This results in a potentially more effective search strategy.

# Chapter 4

## Results

The findings emerged from the experiments conducted to evaluate the performance of Monte Carlo Tree Search (MCTS) in two board games: Tic-Tac-Toe and chess. The primary focus of this section is to understand the influence of opening databases on the performance of MCTS algorithms. The results obtained from these experiments are instrumental in shedding light on the efficacy of different strategies for utilizing opening books in MCTS algorithms, thereby providing a comprehensive understanding of how these strategies can be used in non-gaming scenario which can be solved by MCTS.

Let's first turn our attention to the results for MCTS in the game of Tic-Tac-Toe. The experiment was meticulously designed to assess how varying time constraints impact the performance of the MCTS algorithm. The metrics used for evaluation were the average iteration count (Avg It Count), the number of wins (W), the number of draws (D), and the number of losses (L). The results are intriguing and offer an understanding of the algorithm's behaviour under different conditions.

Time	Avg It Count	Win	Draw	Lose
1	78530	0	100.0%	0
0.1	7051	30.0%	70.0%	0
0.01	539	45.0%	40.0%	15.0%
0.001	12	60.0%	30.0%	10.0%
0.0001	1	100.0%	0.0%	0.0%

Table 4.1: MCTS Tic-Tac-Toe Results

At a time constraint of 1 second per move, the MCTS algorithm performed remarkably

well. With an average iteration count of 78,530, the algorithm achieved a 100% draw rate, with no wins or losses recorded. This suggests that given sufficient time to explore the game tree, the MCTS algorithm is capable of making optimal decisions that lead to draws. It's worth noting that the high average iteration count indicates that the algorithm had the opportunity to explore multiple branches of the game tree, thereby increasing the likelihood of finding the optimal move.

However, as the time constraint was reduced to 0.1 seconds, the performance of the algorithm began to show signs of deterioration. The average iteration count dropped significantly to 7,051, and the draw rate decreased to 70%. Interestingly, the algorithm still managed to avoid any losses and had a 30% win rate. This suggests that while the algorithm's performance declined due to the reduced time for exploration, it is giving an advantage to the player making the first move.

Further reducing the time constraint to 0.01 seconds led to more pronounced changes in the algorithm's performance. The average iteration count plummeted to a mere 539, and the win rate dropped to 45%. Additionally, the algorithm recorded a 40% draw rate and a 15% loss rate. These results indicate that the algorithm's ability to make optimal decisions was severely compromised due to the limited time available for exploration and evaluation.

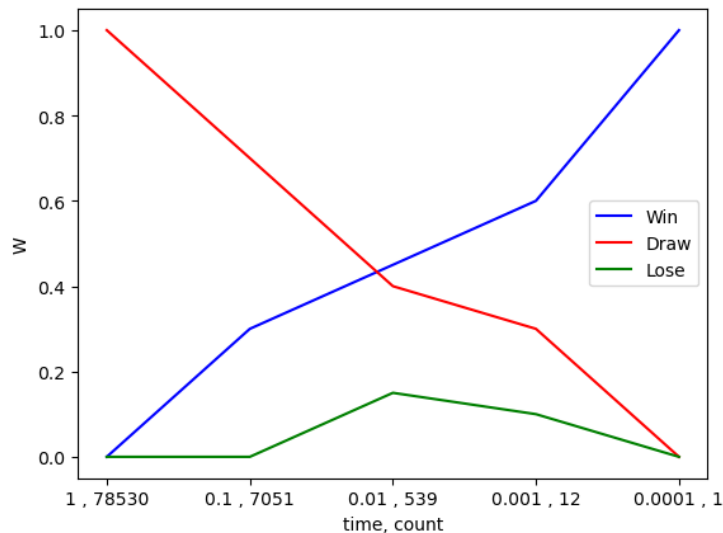


Figure 4.1: MCTS Tic-Tac-Toe Time, Iterations Vs Win, Lose and Draw Percentage

At an even more stringent time constraint of 0.001 seconds, the average iteration

count dropped drastically to just 12. Surprisingly, however, the win rate increased to 60%, accompanied by a 30% draw rate and a 10% loss rate. This counterintuitive result suggests that the algorithm favours the player who moves first and has a natural advantage because they get to control the board from the outset. Also, With limited time, the second MCTS bot has less opportunity to "learn" from the first bot's moves. The first bot, on the other hand, starts with a blank slate and doesn't have to adapt its strategy based on an opponent's previous moves. This also explains the results at an extremely low time constraint of 0.0001 seconds where the algorithm had an average iteration count of just 1 and managed to secure a 100% win rate

White Vs Black	W	D	L
MCTS v DM-MCTS	34.00%	21.00%	45.00%
DM-MCTS v MCTS	56.00%	26.00%	18.00%

Table 4.2: Comparison of MCTS and DM-MCTS in Chess

Next, we transition to the more complex domain of Chess, where the MCTS algorithm was pitted against modified versions of itself, designed to test the impact of different strategies for utilizing opening databases. The first variant, referred to as DM-MCTS, employed a direct move from the opening database. When MCTS with white pieces was pitted against DM-MCTS with black pieces, the former won 34% of the games, drew 21%, and lost 45%. On the flip side, DM-MCTS with white pieces won 56% of the games, drew 26%, and lost 18%. This indicates that the incorporation of a direct move from the opening database can significantly enhance the performance of the MCTS algorithm in Chess, particularly in the critical opening phase. This is a pivotal finding as it demonstrates the potential benefits of integrating domain-specific knowledge into MCTS algorithms, thereby expanding their applicability and effectiveness.

	W	D	L
MCTS v IEC-MCTS	35.0%	50.0%	15.0%
IEC-MCTS v MCTS	15.0%	75.0%	10.0%

Table 4.3: Comparison of MCTS and IEC-MCTS in Chess

The second variant of MCTS, known as IEC-MCTS, was designed to have an increased exploration constant. When MCTS with white pieces competed against IEC-MCTS with



---

black pieces, the former won 35% of the games, drew 50%, and lost 15%. Conversely, IEC-MCTS with white pieces won 15% of the games, drew 75%, and lost 10%. These results suggest that increasing the exploration constant for the book move does not guarantee enhanced performance.

The results offer compelling evidence that modifications to the MCTS algorithm, such as the incorporation of opening databases and adjustments to the exploration constant, can have a profound impact on its performance. These findings not only contribute to the existing body of knowledge but also open up intriguing avenues for future research, particularly in the realm of game-specific optimizations for MCTS. The results serve as a foundation for further studies, offering both a validation of the algorithm's potential and a roadmap for future investigations aimed at refining and expanding its capabilities.

# Chapter 5

## Discussion

### 5.1 Comparison Between Chess Engines

The comparison between MCTS (Monte Carlo Tree Search), DM-MCTS (Direct Move-Monte Carlo Tree Search), and IEC-MCTS (Increase Exploration Constant-Monte Carlo Tree Search) chess engines offers a fascinating look into the role of opening databases and different strategies for leveraging them. The results indicate that DM-MCTS, which directly plays moves found in opening databases, shows a clear advantage over both MCTS and IEC-MCTS. This outcome provides valuable insights into the effectiveness of different approaches to incorporating opening databases in MCTS algorithms for chess.

The DM-MCTS engine’s strategy of directly playing moves from the opening database appears to be highly effective. This is in line with traditional expectations, as opening books in chess are the culmination of centuries of human expertise and analysis. By directly playing these well-studied moves, DM-MCTS gains a strong initial position that likely contributes to its overall success. This approach seems to bypass the complexities and uncertainties introduced by the stochastic nature of MCTS, providing a more deterministic pathway through the opening phase of the game. The advantage gained in the opening phase can have a cascading effect, setting the stage for a stronger middle and endgame.

On the other hand, the IEC-MCTS engine, despite its sophisticated approach of increasing the exploration constant for moves found in the opening database, does not show

a similar advantage. This is particularly intriguing because, theoretically, increasing the exploration constant should allow the engine to consider a broader range of potentially advantageous moves. However, the results suggest that this increased exploration does not necessarily translate into better performance. One possibility is that the increased exploration constant might lead the engine down less promising paths, negating the benefits of the opening database. This could be indicative of a deeper issue related to how MCTS algorithms balance exploration and exploitation, especially in the context of a complex game like chess.

Opening databases are used by both DM-MCTS and IEC-MCTS, however there is a variation in how each method processes the provided position. While IEC-MCTS searches the position and then carries the Trie node throughout the different phases of MCTS like Selection, Expansion, Simulation, and Backpropagation because it examines whether the selected moves in the Selection phase are present in Trie, DM-MCTS checks whether the position is found in Trie or not. This may increase the computational cost of the selection phase of the IEC-MCTS, leaving less time for the other phases, which may lead to fewer iterations and poor performance.

The contrasting performances of DM-MCTS and IEC-MCTS have significant implications for future research. They suggest that the method of incorporating opening databases into MCTS algorithms can drastically affect performance. While DM-MCTS shows that a direct, deterministic approach can be highly effective, the underperformance of IEC-MCTS indicates that more research is needed to understand the optimal way to balance exploration and exploitation in MCTS, especially when opening databases are involved.

## 5.2 Opening Book In MCTS With Applications Beyond Games

The Monte Carlo Tree Search (MCTS) algorithm is widely known for its applications in game-playing, but its utility extends to various other domains, including retrosynthesis<sup>[33]</sup>

in chemistry and vehicle routing problems[26, 41].

The application of Monte Carlo Tree Search (MCTS) in retrosynthesis represents a fusion of computational chemistry and artificial intelligence to solve complex problems in organic synthesis. Retrosynthesis is the process of breaking down a target molecule into simpler, readily available building blocks, essentially working backward from a complex molecule to simpler precursors. MCTS, on the other hand, is an algorithmic approach used for making optimal decisions in a given computational model. The integration of an opening database in MCTS can significantly enhance the efficiency and accuracy of retrosynthetic analysis.

In the context of retrosynthesis, an opening database can be thought of as a repository of pre-computed strategies for breaking down commonly encountered molecular structures. These strategies are generated through extensive simulations and are stored for quick retrieval. When a new target molecule is presented, the algorithm first checks the opening database to see if the molecule or a similar structure has already been analyzed. If so, the pre-computed strategy is used as the starting point, saving computational time and resources.

While the opening database approach offers numerous advantages, it is not without limitations. The quality of the database is crucial; poor-quality data can lead to suboptimal synthetic routes. Moreover, the database needs to be sufficiently large to be useful, requiring significant computational resources for its creation and maintenance. Future work could focus on optimizing the database and integrating it with other computational tools for a more holistic approach to retrosynthesis.

The integration of the opening database approach in Monte Carlo Tree Search (MCTS) can significantly enhance the performance of vehicle routing problems (VRP). The opening database is essentially a pre-computed set of solutions for the initial states of a problem, which can be quickly accessed to provide a strong starting point for further exploration. This approach is particularly useful in complex optimization problems like VRP, where finding an optimal or near-optimal solution can be computationally expensive. Speeding Up the Initial Phase

In traditional MCTS, the algorithm starts from a root node and explores the state space by simulating random playouts. This can be time-consuming, especially in the initial phase where the algorithm has little information about the state space. By using an opening database, the MCTS algorithm can bypass this initial phase, starting from a more informed position. This is particularly beneficial in VRP, where the initial routing decisions can significantly impact the overall efficiency of the solution.

The opening database is usually constructed using high-quality solutions from previous runs or other optimization methods. Therefore, the initial solutions provided by the opening database are often of high quality, which can guide the MCTS to explore more promising branches of the tree. In the context of VRP, this means that the routes generated will likely be more efficient, reducing the overall distance travelled and time taken.

# Chapter 6

## Conclusion

This work presented a comprehensive investigation into the integration of opening books with Monte Carlo Tree Search (MCTS) algorithms for game-playing. The aim was to evaluate different techniques to leverage opening books to improve the performance of MCTS engines in the critical opening phase of games like chess.

The background research revealed that opening books play an indispensable role in traditional chess engines, but their integration with MCTS poses unique challenges due to the algorithm's stochastic nature. Nonetheless, opening books offer substantial value by providing strong starting positions and directing the engine towards well-trodden theoretical lines. An extensive review of the literature established that despite the potential of opening books, there is a glaring gap in focused research on using opening books to improve MCTS.

Three distinct MCTS engines were developed to test different integration strategies - Direct Move MCTS, Increase Exploration Constant MCTS, and MCTS. The Direct Move engine instantly played recognized book moves, while the IEC engine fed book moves into MCTS with an increased exploration constant. These were evaluated over 200 games against MCTS.

The results demonstrated the clear superiority of Direct Move MCTS, which won over 50% of games against MCTS. This underscores the value of directly utilizing opening knowledge rather than treating it stochastically within MCTS. In contrast, increasing the exploration had an unclear benefit, suggesting that more research is needed.

---

Beyond games, opening books also offer assurance in guiding MCTS for vehicle routing problems and retrosynthesis. The integration of an opening database in MCTS offers a powerful tool for solving vehicle routing problems. The incorporation of an opening database in MCTS for retrosynthesis offers a robust and efficient method for navigating the complex landscape of organic synthesis. As computational resources continue to improve, the combination of MCTS and opening databases promises to be a highly effective approach for solving complex search problems.

In conclusion, this work provided valuable insights into the challenges of using opening books to improve MCTS algorithms. While showing clear benefits for games, findings also shed light on the open challenges of this technique for complex planning and optimization problems. This creates a foundation to pursue exciting research directions in both game-playing and non-gaming artificial intelligence.

# Bibliography

- [1] Efficiently updatable neural network. *Efficiently updatable neural network*. [https://en.wikipedia.org/wiki/Efficiently\\_updatable\\_neural\\_network](https://en.wikipedia.org/wiki/Efficiently_updatable_neural_network).
- [2] Introducing nnue evaluation. *Introducing NNUE Evaluation*. <https://stockfishchess.org/blog/2020/introducing-nnue-evaluation/>.
- [3] Neural network repository. *Stockfish testing framework*. [https://tests.stockfishchess.org/nns?network\\_name=nn-62ef826d1a6d.nnue](https://tests.stockfishchess.org/nns?network_name=nn-62ef826d1a6d.nnue).
- [4] *Opening lichess.org* — *lichess.org*. <https://lichess.org/opening/tree>.
- [5] python-chess: a chess library for Python x2014; python-chess 1.10.0 documentation — python-chess.readthedocs.io. <https://python-chess.readthedocs.io/en/latest/#>.
- [6] AHO, A. V., AND HOPCROFT, J. E. *The design and analysis of computer algorithms*. Pearson Education India, 1974.
- [7] ALLIS, L. V. *Searching for solutions in games and artificial intelligence*. 1994.
- [8] ANAND, A., GROVER, A., MAUSAM, AND SINGLA, P. Asap-uct: Abstraction of state-action pairs in uct. In *International Joint Conference on Artificial Intelligence* (2015).
- [9] ANAND, A., NOOTHIGATTU, R., MAUSAM, AND SINGLA, P. Oga-uct: On-the-go abstractions in uct. In *Proceedings of the Twenty-Sixth International Conference on International Conference on Automated Planning and Scheduling* (2016), ICAPS’16, AAAI Press, p. 29–37.



- [10] ARNESON, B., HAYWARD, R. B., AND HENDERSON, P. Monte carlo tree search in hex. *IEEE Transactions on Computational Intelligence and AI in Games* 2, 4 (2010), 251–258.
- [11] AWAIS, M. U., GHASEMZADEH, H., AND PLATZNER, M. An mcts-based framework for synthesis of approximate circuits. *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)* (2018), 219–224.
- [12] BAIER, H., AND WINANDS, M. H. M. Active opening book application for monte-carlo tree search in  $19 \times 19$  go.
- [13] BEAL, D. A generalised quiescence search algorithm. *Artificial Intelligence* 43 (1990), 85–98.
- [14] BEST, G., CLIFF, O. M., PATTEN, T., METTU, R. R., AND FITCH, R. Dec-mcts: Decentralized planning for multi-robot active perception. *The International Journal of Robotics Research* 38, 2-3 (2019), 316–337.
- [15] CHASLOT, G., WINANDS, M., HERIK, H., UITERWIJK, J., AND BOUZY, B. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation* 04 (11 2008), 343–357.
- [16] CHEN, K.-H., AND ZHANG, P. Building opening books for  $9 \times 9$  go without relying on human go expertise. *Journal of Computer Science* 8 (2012), 1594–1600.
- [17] COWLING, P., POWLEY, E., AND WHITEHOUSE, D. Information set monte carlo tree search. *IEEE Transactions on Computational Intelligence and Ai in Games* 4 (06 2012), 120–143.
- [18] DONALD E. KNUTH, R. W. M. *An analysis of alpha-beta pruning*. 2003.
- [19] FURTAK, T., AND BURO, M. Recursive monte carlo search for imperfect information games. pp. 1–8.

- [20] GELLY, S., AND WANG, Y. Exploration exploitation in Go: UCT for Monte-Carlo Go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop* (Canada, Dec. 2006).
- [21] HSU, F.-H. Ibm’s deep blue chess grandmaster chips. *IEEE Micro* 19, 2 (1999), 70–81.
- [22] J-B, G., HOOCK, J.-B., PEREZ, J., RIMMEL, A., TEYTAUD, O., AND WINANDS, M. Meta monte-carlo tree search for automatic opening book generation.
- [23] KLOETZER, J. *Monte-Carlo Opening Books for Amazons*. 2010.
- [24] KLOETZER, J., IIDA, H., AND BOUZY, B. The monte-carlo approach in amazons.
- [25] KOCSIS, L., AND SZEPESVARI, C. *Bandit Based Monte-Carlo Planning*. 2006.
- [26] MADZIUK, J., AND WIECHOWSKI, M. Uct in capacitated vehicle routing problem with traffic jams. *Information Sciences* 406-407 (2017), 42–56.
- [27] NEUMANN, J. v. Zur theorie der gesellschaftsspiele. *Mathematische Annalen* 100 (1928), 295–320.
- [28] PAINTER, M., LACERDA, B., AND HAWES, N. Convex hull monte-carlo tree search. *ArXiv abs/2003.04445* (2020).
- [29] SABAR, N. R., AND KENDALL, G. Population based monte carlo tree search hyper-heuristic for combinatorial optimization problems. *Inf. Sci.* 314 (2015), 225–239.
- [30] SCHRITTWIESER, J., ANTONOGLU, I., HUBERT, T., SIMONYAN, K., SIFRE, L., SCHMITT, S., GUEZ, A., LOCKHART, E., HASSABIS, D., GRAEPEL, T., LILLICRAP, T., AND SILVER, D. Mastering atari, go, chess and shogi by planning with a learned model. *Nature* 588, 7839 (dec 2020), 604–609.
- [31] SCHRÜFER, G. A strategic quiescence search. *ICGA Journal* 12, 1 (1989), 3–9.
- [32] SEGLER, M., PREUSS, M., AND WALLER, M. Planning chemical syntheses with deep neural networks and symbolic ai. *Nature* 555 (03 2018), 604–610.

- [33] SEGLER, M. H. S., PREUSS, M., AND WALLER, M. P. Planning chemical syntheses with deep neural networks and symbolic ai. *Nature* 555 (2018), 604–610.
- [34] SHANNON, C. E., AND TELEPHONE, B. Xxii. programming a computer for playing chess 1. *Philosophical Magazine Series 1* 41 (1950), 256–275.
- [35] SHI, F., K SOMAN, R., HAN, J., AND WHYTE, J. Addressing adjacency constraints in rectangular floor plans using monte-carlo tree search. *Automation in Construction* 115 (03 2020).
- [36] SILVER, D., HUANG, A., MADDISON, C., GUEZ, A., SIFRE, L., DRIESSCHE, G., SCHRITTWIESER, J., ANTONOGLOU, I., PANNEERSHELVAM, V., LANCTOT, M., DIELEMAN, S., GREWE, D., NHAM, J., KALCHBRENNER, N., SUTSKEVER, I., LILLICRAP, T., LEACH, M., KAVUKCUOGLU, K., GRAEPEL, T., AND HASSABIS, D. Mastering the game of go with deep neural networks and tree search. *Nature* 529 (01 2016), 484–489.
- [37] SILVER, D., HUBERT, T., SCHRITTWIESER, J., ANTONOGLOU, I., LAI, M., GUEZ, A., LANCTOT, M., SIFRE, L., KUMARAN, D., GRAEPEL, T., LILLICRAP, T., SIMONYAN, K., AND HASSABIS, D. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science* 362, 6419 (2018), 1140–1144.
- [38] SILVER, D., SCHRITTWIESER, J., SIMONYAN, K., ANTONOGLOU, I., HUANG, A., GUEZ, A., HUBERT, T., BAKER, L., LAI, M., BOLTON, A., CHEN, Y., LILLICRAP, T. P., HUI, F., SIFRE, L., VAN DEN DRIESSCHE, G., GRAEPEL, T., AND HASSABIS, D. Mastering the game of go without human knowledge. *Nature* 550 (2017), 354–359.
- [39] WU, F., RAMCHURN, S., JIANG, W., FISCHER, J., RODDEN, T., AND JENNINGS, N. Agile planning for real-world disaster response.

- [40] ŚWIECHOWSKI, M., GODLEWSKI, K., SAWICKI, B., AND MAŃDZIUK, J. Monte carlo tree search: a review of recent modifications and applications. *Artificial Intelligence Review* 56 (2022), 2497–2562.
- [41] ŚWIECHOWSKI, M., AND MAŃDZIUK, J. Simulation-based approach to vehicle routing problem with traffic jams.