Using Depth First Search on Minimax Algorithm for Optimal Move Sequence in Chess Endgames

Ikhwan Al Hakim – 13522147[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
[1]13522147@std.stei.itb.ac.id

Abstract— This paper explores the application of Depth First Search (DFS) within the Minimax algorithm to determine optimal move sequences in chess endgames. The Minimax algorithm, traditionally used for decision-making in two-player games, evaluates moves by simulating possible future game states. By integrating DFS, we aim to enhance the efficiency of this algorithm in the context of chess endgames, where the search space is vast but finite. Our approach leverages the depth-first traversal capabilities of DFS to prioritize deeper exploration of promising moves, thereby improving the accuracy of optimal move identification.

Keywords—Depth First Search, Minimax Algorithm, Chess Endgames, Optimal Move Sequence

I. Introduction

Chess, a game of strategic complexity and depth, has long been a subject of interest for both human players and artificial intelligence researchers. The endgame phase, characterized by fewer pieces on the board and a higher reliance on precise calculations, presents unique challenges and opportunities for algorithmic analysis. Traditional methods for determining optimal moves in chess endgames involve exhaustive search techniques, which can be computationally intensive and time-consuming.

The Minimax algorithm is a fundamental approach used in game theory for decision-making in two-player games, including chess. It operates by simulating all possible moves and counter-moves to determine the best strategy for a player, assuming optimal play from both sides. However, as the complexity of the game increases, the efficiency of the Minimax algorithm becomes a critical concern.

To address this issue, we propose integrating Depth First Search (DFS) with the Minimax algorithm to enhance its performance in chess endgames. DFS, known for its ability to explore deep into search trees, allows for a more focused and efficient traversal of the game state space. By leveraging DFS, we aim to prioritize the examination of promising moves, thereby reducing the computational burden and improving the accuracy of the Minimax algorithm.

II. Theoretical Basis

A. Chess

Chess is a strategic two-player board game that has been played for centuries, characterized by its deep complexity and intellectual challenge. The game is played on an 8x8 square board, with each player controlling an army of 16 pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. The objective is to checkmate the opponent's king, putting it in a position where it cannot escape capture.

Figure 1. Chess Board
Source: https://images.chesscomfiles.com/uploads/v1/images_users/tiny_mce/lularobs/php6RFk0y.png

Chess involves a combination of tactics and strategy. Tactical play focuses on short-term moves and immediate threats, such as capturing opponent pieces or creating threats that must be responded to. Strategic play, on the other hand, involves long-term planning and positioning, aiming to control key areas of the board, improve piece activity, and create enduring advantages that can be leveraged later in the game.

1) Pieces

The game starts with a well-defined initial setup and proceeds through a series of turns, where players move

their pieces according to specific rules. Each type of piece has its own unique movement patterns.

a) King

The king can move one square horizontally, vertically, or diagonally. The king cannot move into a square that is under attack by an opponent's piece, as this would put it in check.

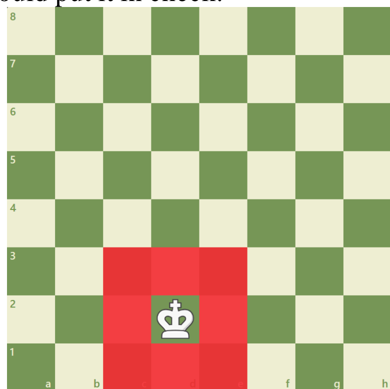

Figure 2. King's Movement in Chess
Source:
https://images.chesscomfiles.com/uploads/v1/images_users/tiny_mce/pdrpnht/phpmVRKYr.png

b) Queen

The queen can move any number of squares in any direction. It combines the powers of both the rook and the bishop, making it the most powerful piece in terms of mobility.

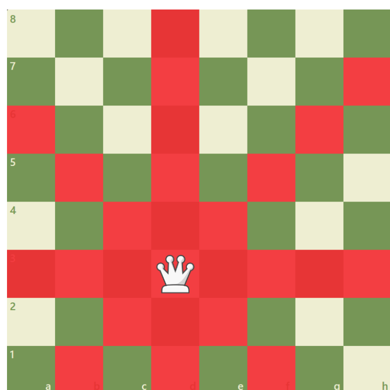

Figure 3. Queen's Movement in Chess
Source:
https://images.chesscomfiles.com/uploads/v1/images_users/tiny_mce/pdrpnht/phpCQgsYR.png

c) Rook

The rook can move any number of squares horizontally or vertically.

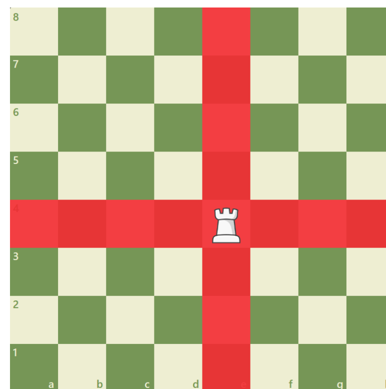

Figure 4. Rook's Movement in Chess
Source:
https://images.chesscomfiles.com/uploads/v1/images_users/tiny_mce/pdrpnht/phpfyINI1.png

d) Bishop

The bishop can move any number of squares diagonally. Each bishop is restricted to one color of squares (light or dark) for the entire game.

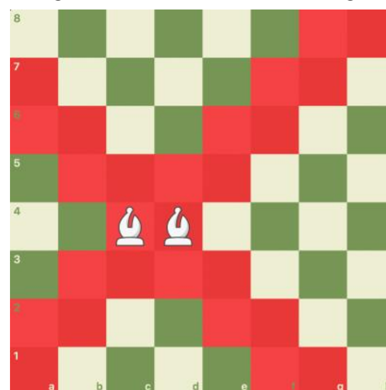

Figure 5. Bishop's Movement in Chess
Source:
https://images.chesscomfiles.com/uploads/v1/images_users/tiny_mce/PeterDoggers/php4dzIxh.png

e) Knight

The knight moves in an L-shape. The knight is the only piece that can "jump" over other pieces, meaning it can move to a square even if there are pieces in between.

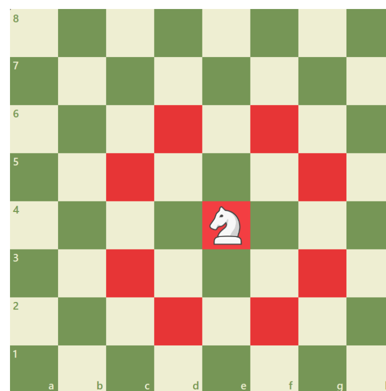

Figure 6. Knight's Movement in Chess

Source:
https://images.chesscomfiles.com/uploads/v1/images_users/tiny_mce/pdrpnht/phpVuLl4W.png

f) Pawn

Pawns move forward one square, but capture diagonally. On their first move, pawns have the option to move forward two squares.

Figure 7. Pawn's Movement in Chess
Source:
https://images.chesscomfiles.com/uploads/v1/images_users/tiny_mce/pdrpnht/phpEH1kWv.png

Pawns have two special moves, namely en passant and promotion.

a. En Passant

If a pawn moves two squares forward from its starting position and lands beside an opponent's pawn, the opponent's pawn can capture it as if it had moved only one square.

Figure 8. En Passant
Source:
https://images.chesscomfiles.com/uploads/v1/images_users/tiny_mce/pdrpnht/phpQ4CGRG.png;
https://images.chesscomfiles.com/uploads/v1/images_users/tiny_mce/pdrpnht/php3fpaGV.png

b. Promotion

When a pawn reaches the opponent's back rank (eighth rank for White, first rank for Black), it can be promoted to any other piece (queen, rook, bishop, or knight), except another king.

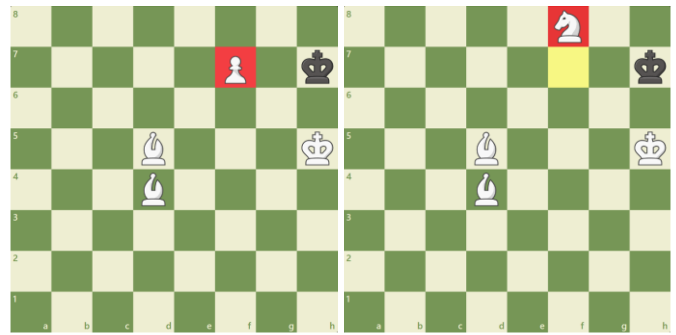

Figure 9. Promotion
Source:
https://images.chesscomfiles.com/uploads/v1/images_users/tiny_mce/pdrpnht/phpIlmnmf.png;
https://images.chesscomfiles.com/uploads/v1/images_users/tiny_mce/pdrpnht/phpZeuI6e.png

2) Phases

a) Opening

The opening phase of chess involves the initial moves where players aim to develop their pieces to active squares, control the center of the board, and ensure the safety of their king, typically by castling. Players often follow established opening principles and theories to achieve these objectives, balancing development, control, and king safety. Efficient and effective piece development during the opening can set the stage for a strong middlegame, while errors can lead to early disadvantages. Common opening moves include advancing the central pawns (e.g., e4, d4) and developing knights and bishops to prepare for castling and connecting the rooks.

b) Middlegame

The middlegame begins once both players have completed their development and the board becomes a battlefield of tactics and strategy. This phase is characterized by intricate maneuvers, combinations, and attacks as players seek to gain material or positional advantages. Key elements of the middlegame include planning and executing tactics like forks, pins, and skewers, as well as formulating strategies that exploit weaknesses in the opponent's position. Effective coordination of pieces, dynamic play, and the ability to foresee and counter the opponent's plans are crucial for success in the middlegame.

c) Endgame

The endgame occurs when most pieces have been traded off and the focus shifts to converting any advantages into a win, often through precise and methodical play. In this phase, the activity of the king becomes paramount, often acting as a strong piece in both offense and defense. Pawn promotion is a critical aspect, where pawns advance to the back rank to be promoted, typically to a queen, to gain decisive material superiority. Endgame knowledge, including key

concepts like opposition, zugzwang, and the principle of the square, is essential for converting small advantages and achieving checkmate or securing a draw in disadvantageous positions.

B. Depth First Search

Depth First Search (DFS) is a fundamental algorithm in graph theory used for traversing or searching tree or graph data structures. It starts at a chosen root node and explores as far as possible along each branch before backtracking. This approach is achieved by utilizing a stack data structure, either through a recursive function call stack or an explicit stack. The algorithm marks nodes as visited and proceeds to visit an unvisited adjacent node, pushing each onto the stack. If it reaches a node with no unvisited adjacent nodes, it backtracks to the previous node, continuing this process until all nodes reachable from the initial node have been visited. DFS is known for its ability to provide a systematic way of exploring all possible paths in a graph, making it useful for solving problems like finding connected components, topological sorting, and detecting cycles in directed graphs. However, it may not always find the shortest path in unweighted graphs, as it explores deeply along each branch before considering sibling nodes.

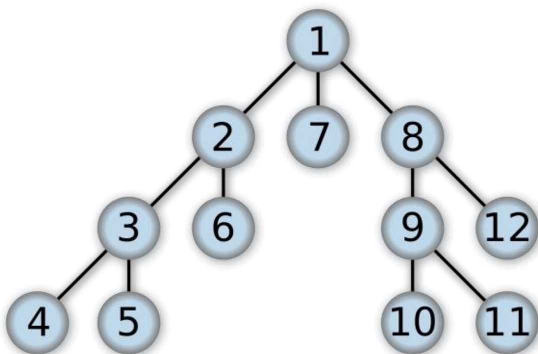

Figure 10. Depth First Search Tree
Source: https://upload.wikimedia.org/wikipedia/commons/thumb/1/1f/Depth-first-tree.svg/1280px-Depth-first-tree.svg.png

C. Minimax Algorithm

The Minimax algorithm is a decision-making tool used primarily in two-player zero-sum games like chess and tic-tac-toe. It aims to minimize the possible loss for a worst-case scenario, effectively maximizing the minimum gain. In Minimax, one player is considered the maximizer, seeking to maximize their score, while the other is the minimizer, aiming to minimize the maximizer's score. The algorithm recursively evaluates the game tree, where each node represents a game state and edges represent possible moves. Starting from the current state, it simulates all possible moves, generating a tree of potential future states. At each level, it alternates between the maximizer and minimizer until it reaches terminal nodes, which are evaluated using a utility function that assigns a numerical value to the outcome. The minimax value of a node is determined by its children: the maximizer chooses the move with the highest value, while the minimizer selects the one with the lowest value. By backtracking from the terminal nodes to the root, the algorithm identifies the optimal move for the current player, ensuring the best achievable outcome against a rational opponent.

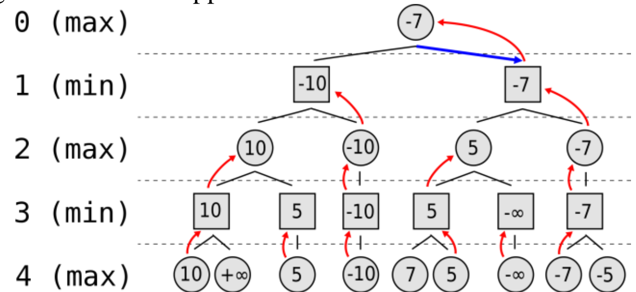

Figure 11. Minimax Tree
Source:
https://upload.wikimedia.org/wikipedia/commons/thumb/6/6f/Minimax.svg/1920px-Minimax.svg.png

III. Methodology

Before the author explain any of the method used in this research, it should be noted that the experiment is done in the following hardware specifications:

- Processor: AMD Ryzen 7 5800HS 3.2GHz
- RAM: 16 GB

A. Defining the Endgame Position

Before we start to make a move sequence for a particular endgame position, we have to determine which endgame position we want to use. For this experiment, the author will use one of the most famous endgame position called “The Lucena Position”.

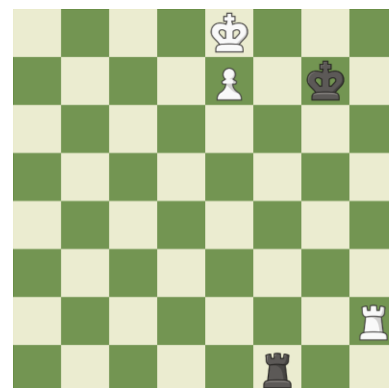

Figure 12. The Lucena Position
Source: chess.com

Black is to play in this position and we will generate the most optimal move sequence for white to win the game by assuming both sides played the best move in each turn. Note that the “best” move here is the best move generated by the minimax tree and it can differ from the actual best move as this paper is made in a small environment and a not-so-good hardware.

B. Generating the Movement Tree

By using the position from figure 12, we can then generate the movement tree that consists of the movements possible from that position. The author is using Stockfish chess engine to help generate the moves. Below is the code implementation to generate the minimax tree from a certain position:

```
from stockfish import Stockfish

class Node:
    def __init__(self, sf, move):
        self.child = []
        self.sf = sf
        self.move = move
        self.eval = sf.get_evaluation()

    def add_child(self, n):
        self.child.append(n)

        def make_tree(self, depth, child_count):
        if (depth == 0):
            return

        moves = self.sf.get_top_moves(child_count)
        for move in moves:
            new_sf = 
Stockfish(path="./stockfish/stockfish-windows-x86-64-
avx2.exe")

new_sf.set_fen_position(self.sf.get_fen_position())

new_sf.make_moves_from_current_position([move['Move']])
            move_temp = self.move.copy()
            move_temp.append(move['Move'])
            child = Node(new_sf, move_temp)
            self.add_child(child)
            child.make_tree(depth-1, child_count)
            del move_temp
```

For the sake of calculation simplicity, the author will create the tree with a depth of only 3 and each node will only have 2 children.

```
fen = "4K3/4P1k1/8/8/8/8/7R/5r2 b - - 0 1"
stockfish = Stockfish(path="./stockfish/stockfish-
windows-x86-64-avx2.exe")
stockfish.set_fen_position(fen)
root = Node(stockfish, [])
root.make_tree(3, 2)
```

C. Calculating the Minimax Value

This function will generate the minimax value by using depth first search. The minimax value will differ depending on which side is currently playing. Because of that, the author will add a distinguishing variable to distinguish the minimax function result according to whose turn it is.

```
def minimax(node, depth, maximizing):
    if depth == 0 or len(node.child) == 0:
        return node

    if maximizing: # white turn
        if node.eval['type'] == 'cp' or 
(node.eval['type'] == 'mate' and node.eval['value'] <= 
0):
            extreme_node = minimax(node.child[0], depth-
1, False)
            for i in range (1, len(node.child)):
                minimax_node = minimax(node.child[i], 
depth-1, False)
                if (minimax_node.eval['value'] > 
extreme_node.eval['value']):
                    extreme_node = minimax_node
        else:
            extreme_node = minimax(node.child[0], depth-
1, False)
            for i in range (1, len(node.child)):
                minimax_node = minimax(node.child[i], 
depth-1, False)
                if (minimax_node.eval['value'] < 
extreme_node.eval['value']):
                    extreme_node = minimax_node
    else: # black turn
        if node.eval['type'] == 'cp' or 
(node.eval['type'] == 'mate' and node.eval['value'] >= 
0):
            extreme_node = minimax(node.child[0], depth-
1, True)
            for i in range (1, len(node.child)):
                minimax_node = minimax(node.child[i], 
depth-1, True)
                if (minimax_node.eval['value'] < 
extreme_node.eval['value']):
                    extreme_node = minimax_node
        else:
            extreme_node = minimax(node.child[0], depth-
1, True)
            for i in range (1, len(node.child)):
                minimax_node = minimax(node.child[i], 
depth-1, True)
                if (minimax_node.eval['value'] > 
extreme_node.eval['value']):
                    extreme_node = minimax_node
    return extreme_node
```

If white is currently playing, the maximizing variable will be valued True and the function will return a value that'll favor the white side accordingly. In the contrary, if black is currently playing, the maximizing variable will be valued False.

IV. Result

By using the minimax function mentioned earliier, we can then generate the minimax tree from the initial Lucena Position.

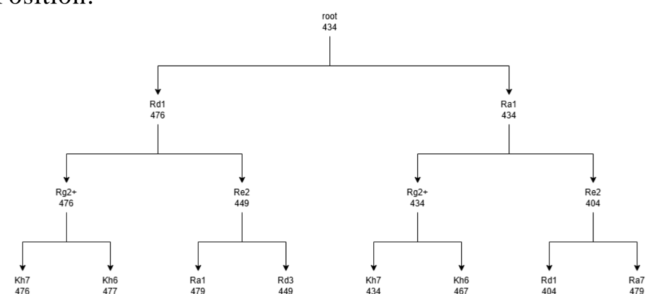

Figure 13. Minimax Tree on Initial Position
Source: author's documentation

The move sequence returned by this function on this position is Ra1, Rg2+, Kh7. Then we play Ra1 as black on this position. Below is the board after we play the move:

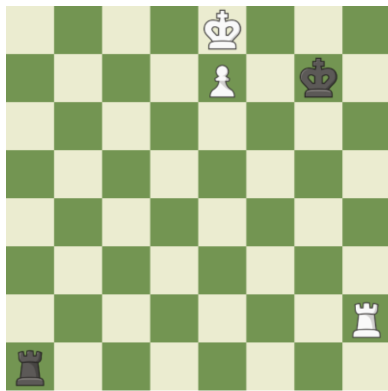

Figure 14. The Board after Ra1
Source: chess.com

Then we repeat the same process. Generate the minimax tree using the minimax function then play the move on the board.

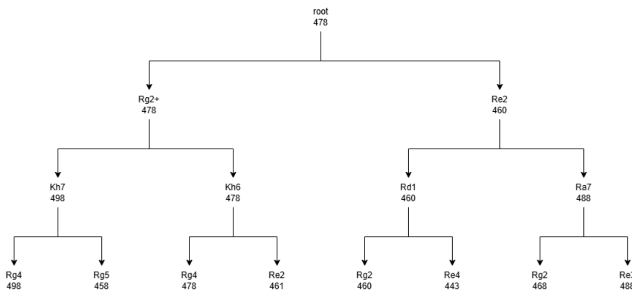

Figure 15. Minimax Tree on the Position After Ra1
Source: author's documentation

Because it is white's turn, we change the orientation from finding the minimum value to finding the maximum value. Thus the move sequence generated from this tree is Rg2+, Kh6, Rg4. Below is the board after we played Rg2+:

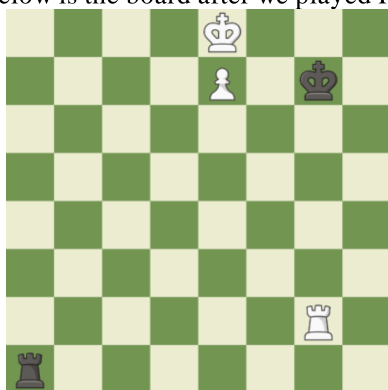

Figure 16. The Board after Rg2+
Source: chess.com

Because this whole process will take a lot of time if we do it manually, the author will automate all of the steps into a code below:

```
def is_game_over(sf):
    info = sf.get_evaluation()
    return (info['type'] == 'mate' and info['value'] == 0)

def print_board(sf):
    fen_to_ascii = {
    'R': '♜', 'N': '♞', 'B': '♝', 'Q': '♛', 'K': '♚', 'P': '♟',
    'r': '♖', 'n': '♘', 'b': '♗', 'q': '♕', 'k': '♔', 'p': '♙',
    }
    board = sf.get_board_visual()
    ascii_board = ""
    for i in range (len(board)):
        if i + 31 >= len(board):
            ascii_board += board[i]
        else:
            ascii_board += fen_to_ascii.get(board[i], board[i])
    return ascii_board

fen = "4K3/4P1k1/8/8/8/8/7R/5r2 b - - 0 1"
white = False
while True:
    stockfish = Stockfish(path="./stockfish/stockfish-windows-x86-64-avx2.exe")
    stockfish.set_fen_position(fen)
    root = Node(stockfish, [])
    root.make_tree(3, 2)

stockfish.make_moves_from_current_position([minimax(root, 3, white).move[0]])
    print(print_board(stockfish))
    fen = stockfish.get_fen_position()
    print(fen)

    if white:
        white = False
    else:
        white = True

    if (is_game_over(stockfish)):
        print("game over")
        break
```

Then the code will generate these move sequence:

```
1. Ra1 2. Rg2+ 3. Kf6 4. Kf8 5. Ra8+ 6. e8=R 7. Ra4 8. Rf2 9. Kg6 10. Re1 11. Ra8+ 12. Ke7 13. Kh5 14. Rg2 15. Ra4 16. Rh1 17. Rh4 18. Rxh4 19. Kxh4 20. Rg6 21. Kh5 22. Kf6 23. Kh4 24. Kg7 25. Kh3 26. Kf7 27. Kh2 28. Rg7 29. Kh3 30. Ke7 31. Kh4 32. Kf6 33. Kh5 34. Rg6 35. Kh4 36. Ke6 37. Kh5 38. Kf6 39. Kh4 40. Ke6 41. Kh3 42. Ke5 43. Kh2 44. Rg7 45. Kh3 46. Kf4 47. Kh2 48. Kf3 49. Kh1 50. Kf2 51. Kh2 52. Rh7#
```

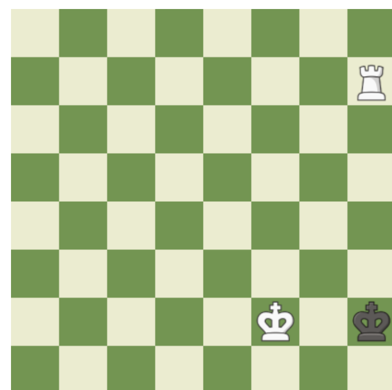

Figure 17. White Winning the Game
Source: chess.com

The length of the winning sequence for white is 52, which is pretty normal for a king and rook endgame considering both sides is guaranteed to do the best move in each turn.

V. Conclusion

From the conducted experiment, it is safe to say that the combination of depth first search and minimax algorithm can be used to generate the most optimal move sequence in chess endgames. However, the minimax movement tree in this experiment is so limited in terms of size and depth. Due to the constraint, the author is certain that the result of this experiment can still be improved with more complex tree in a better and more complex environment.

VI. Suggestion

The author's suggestion for future researchers that want to continue this experiment is to do it in a more high-end hardware so that the computer can handle a bigger and deeper movement tree, thus be able to generate a better and shorter move sequence to obliterate the opposing side faster.

Repository and Video Link

https://github.com/Nerggg/Stockfish_Minimax_Simulation/

Acknowledgment

The author extends heartfelt gratitude to Allah, the Most Merciful and Compassionate, for providing the strength, wisdom, and perseverance to undertake this scholarly journey. Acknowledgment is due to the esteemed lecturer, Dr. Ir. Rinaldi Munir, M.T., Ir. Rila Mandala, M.Eng., Ph.D., and Monterico Adrian, S.T., M.T., whose invaluable guidance, insightful feedback, and unwavering support have profoundly shaped the quality of this work. The author is profoundly grateful to their family for the unwavering love and encouragement that served as pillars of strength, and to friends, whose inspiration and camaraderie enriched the entire process. The author recognizes and appreciates the collective contributions, big and small, from everyone involved, and hopes that this work, guided by divine grace, contributes positively to the broader realm of knowledge.

References

[1] Munir, Rinaldi. (2020). "Breadth/Depth First Search (BFS/DFS) (Bagian 2)" . https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf (accessed on 9th June 2024)

[2] (2023). CHESS AI: Machine learning and Minimax based Chess Engine. doi: 10.1109/iconat57137.2023.10080746 (accessed on 9th June 2024)

[3] Yang, Hu., Guoyu, Zuo. (2019). Expectation Minimax Algorithm for the Two-Player Military Chess Game. doi: 10.1109/CCDC.2019.8833085 (accessed on 11th June 2024)

[4] Mitsuru, Shibayama. (2015). Minimax approach to the n-body problem. 221-228. doi: 10.2969/ASPM/06410221 (accessed on 11th June 2024)

STATEMENT

I, the individual signing below, affirm that the content presented in this document is an original creation authored by me. It is not a derivative work, translation of another document, or a product of plagiarism.

Bandung, 12th June 2024

Ikhwan Al Hakim, 13522147