Application of Game Trees and Alpha-Beta Pruning in Optimizing Chess Engines

Henry Filberto Shenelo - 13523108[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523108@std.stei.itb.ac.id, [2]henryfilbertoshenelo@gmail.com

***Abstract*— The complexity of the game of chess makes it a standard benchmark for evaluating artificial intelligence (AI) algorithms. With over 10^{120} possible board states, efficient decision-making in chess requires advanced computational techniques. Game trees and alpha-beta pruning are widely used to optimize chess engines by reducing computational overhead and enhancing search efficiency. Game trees represent all possible game states, while alpha-beta pruning eliminates irrelevant branches, allowing deeper searches within a fixed timeframe. This paper explores the application of game trees and alpha-beta pruning in optimizing chess engines. The theoretical foundations of game trees and minimax principles are examined, alongside alpha-beta pruning as an optimization technique. The effectiveness of these methods is demonstrated through case studies, performance metrics, and implementation of a chess engine, showing significant improvements in search depth and computational efficiency. The results highlight the advantages of combining game tree exploration with alpha-beta pruning to optimize chess engines for real-time decision-making.**

***Keywords*— Chess engine, game tree, alpha-beta pruning, minimax algorithm, artificial intelligence.**

I. INTRODUCTION

The game of chess has long been a benchmark for evaluating artificial intelligence (AI) algorithms due to its complexity, strategic depth, and requirement for precise decision-making. With over 10^{120} possible board states, chess poses a formidable challenge for computational techniques, necessitating efficient algorithms for move selection and evaluation. Among the most widely used approaches for optimizing decision-making in chess engines are game tree exploration and alpha-beta pruning, which allow AI to simulate potential moves and counter-moves while minimizing computational overhead.

Game trees represent the foundation of chess AI, modelling the sequence of possible moves as a branching structure where nodes correspond to board states and edges denote potential moves. By traversing this structure, a chess engine can evaluate and compare various strategies to determine the optimal sequence of moves. However, the sheer size of the game tree often leads to combinatorial explosion, making it impractical to evaluate all possible moves within reasonable time constraints. To address this, alpha-beta pruning is employed to significantly reduce the number of nodes explored in the game tree without compromising the accuracy of the decision-making process.

Alpha-beta pruning enhances the performance of minimax-based search algorithms by eliminating branches of the game tree that cannot influence the final decision. This optimization reduces computational complexity, enabling chess engines to search deeper into the game tree within a fixed timeframe. The combination of game tree structures and alpha-beta pruning forms the backbone of many state-of-the-art chess engines, facilitating real-time decision-making and improved strategic play.

This paper explores the application of game trees and alpha-beta pruning in optimizing chess engines, highlighting their theoretical foundations, implementation methodologies, and practical effectiveness. By analysing case studies and performance metrics, we demonstrate how these techniques enhance move evaluation, search efficiency and reduce running time.

II. THEORETICAL BASIS

A. *Graph*

Graphs are commonly used to represent discrete objects and the relationships between them. A Graph G is defined as G = (*V*, *E*), where

1. *V* represents a non-empty set of vertices (nodes), denoted a $V = \{v_1, v_2, \ldots, v_n\}$. The set *V* must not be empty, ensuring that a graph must contain at least one vertex.
2. *E* represents a set of edges that connect pairs of vertices, denoted as $E = \{e_1, e_2, \ldots, e_m\}$. The set *E* can be empty, meaning a graph can exist without any edges.

Based on the orientation of edges, graphs are classified into two types:

1. Undirected graph, which is a graph where the edges have no specific direction
2. Directed graph (or digraph), which is a graph where each edge is assigned a specific direction.

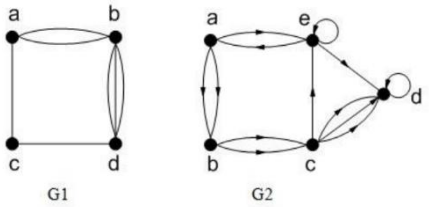

Figure 2. Illustration of an undirected graph (G1) and a directed graph (G2)
Source: [1]

A graph *G* is a *connected graph* if, for every pair of vertices v_i and v_j in the set *V*, there exists a path connecting v_i and v_j.

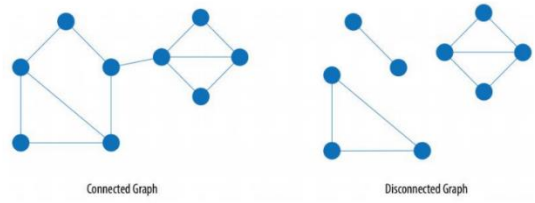

Figure 3. Illustration of connected graphs and disconnected graph
Source: [1]

In a graph G, A path that starts and ends at the same vertex is called a circuit or cycle.

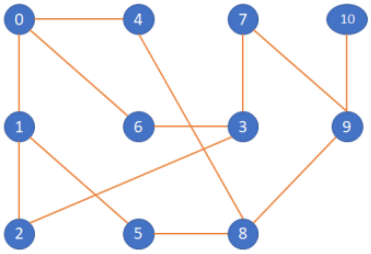

Figure 4. Path (0, 4, 8, 5, 1, 0) forms a circuit
Source: [1]

B. Tree

A tree is an undirected graph that is connected and contains no circuits (cycles). Thus, the three conditions for a graph to be classified as a tree are: it must be undirected, connected, and free of circuits (cycles). A rooted tree is a tree in which one of its vertices is designated as the root, and its edges are assigned directions to form a directed graph.

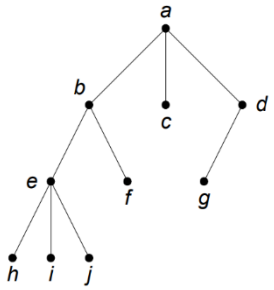

Figure 5. Illustration of a rooted tree
Source: [3]

Rooted trees have several fundamental terminologies to describe their structure and relationships:

1. Child and Parent
   A vertex v_1 is a *child* of vertex v_2 if there is a directed edge connecting v_2 to v_1. The originating vertex v_2 is the *parent* of v_1.
2. Path
   A *path* is a sequence of vertices and edges from a starting vertex to a destination vertex.
3. Ancestor
   A vertex v_1 is an *ancestor* of vertex v_2 if there is a path from v_1 to v_2. In other words, the ancestors of a vertex v_2 are all vertices along the path from the root to v_2.
4. Sibling
   Vertices v_1 and v_2 are called *siblings* if they share the same parent.
5. Subtree
   A *subtree* is a tree where its root is a child of another vertex in the main tree, essentially forming a subset of the original tree.
6. Degree
   The *degree* of a vertex is the number of children it has.
7. Leaf
   A *leaf* is a vertex that has no children, i.e., it has a degree of zero.
8. Internal node
   An *internal node* is a vertex that has at least one child.
9. Level
   The *level* of a vertex is the distance from the root to that vertex.
10. Height/Depth
    The *height/depth* of a tree is the maximum distance between the root and any leaf.

C. Game Tree

A game tree is a structure used to represent the dynamics of a two-player game. Each node *n* in this tree corresponds to a specific position or state in the game. For instance, Figure 1 illustrates a game tree that shows the first two levels of tic-tac-toe.

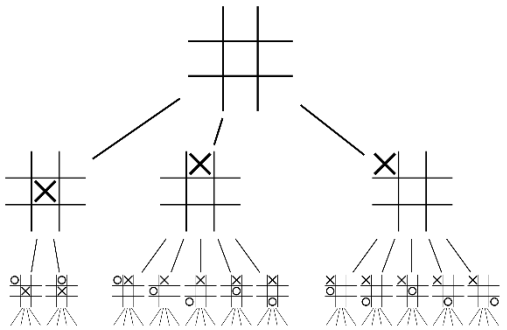

Figure 6. The first two plies of the game tree for tic-tac-toe.
Source: https://www.researchgate.net/figure/Part-of-the-game-tree-for-tic-tac-toe-ie-the-tree-that-follows-all-possible-moves-of_fig3_378437284

The two players involved are referred to as *Max* and *Min*. The value of a position, denoted by *f(p)*, represents the maximum guaranteed payoff that Max can secure, assuming both players play optimally. This value is determined using the minimax principle, which defines the game value based on the following:

1. If *n* is a Max node,
$$f(n) = max\{f(c) \mid c \text{ is a child of } n\}$$
2. If *n* is a Min node,
$$f(n) = min\{f(c) \mid c \text{ isa child of } n\}$$

This principle ensures that the game value at any node reflects the optimal strategy for the player whose turn it is to move. In essence, Max aims to maximize the game value, while Min strives to minimize it. These rules underpin the logic of algorithms designed to analyse and compute strategies for such games.

D. Alpha-Beta Pruning

Alpha-beta pruning is an optimization technique designed to enhance the search efficiency of the minimax algorithm. This method works by pruning or eliminating nodes that are deemed irrelevant during the minimax search process. By doing so, the total number of nodes evaluated is significantly reduced compared to the number of nodes processed by the standard minimax algorithm without pruning.

Alpha-beta pruning is based on the concept that during a search, nodes can have two potential values: an optimistic value and a pessimistic value, represented as alpha (α), the maximum threshold, and beta (β), the minimum threshold. These bounds are used to ensure that certain state changes do not influence the final search results.

For example, suppose *F* is a function that seeks the maximum value, *G* is a function that seeks the minimum value, and P is a node in the search tree. If the minimax algorithm evaluates $F(p_1)$ = -10 and determines $G(p)$ >= 10, it is unnecessary to evaluate all sub nodes of another node $F(p_2)$.

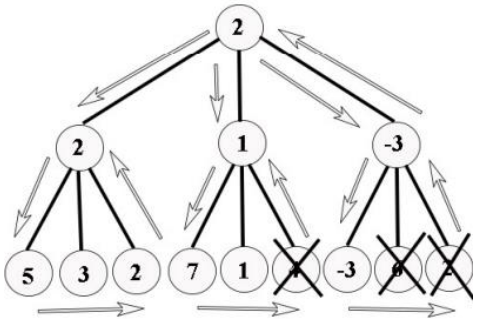

Figure 7. An Illustration of Alpha-Beta Pruning in the Minimax Algorithm

The pseudocode for Alpha-Beta Pruning is given below

```
function AlphaBeta(node, depth, α, β,
maximizingPlayer):
   if depth == 0 or node is a terminal node:
      return the heuristic value of node

   if maximizingPlayer:
      maxEval = -∞
      for each child of node:
         eval = AlphaBeta(child, depth - 1, α, β, false)
         maxEval = max(maxEval, eval)
         α = max(α, eval)
         if β <= α:
            break  // Beta cut-off
      return maxEval
   else:
      minEval = ∞
      for each child of node:
         eval = AlphaBeta(child, depth - 1, α, β, true)
         minEval = min(minEval, eval)
         β = min(β, eval)
         if β <= α:
            break  // Alpha cut-off
      return minEval
```

The algorithm recursively evaluates game positions, pruning branches where it can be determined that the current position is worse than a previously explored position (i.e., pruning occurs if $\alpha \geq \beta$). This pruning significantly reduces the number of nodes evaluated, making the search more efficient.

III. IMPLEMENTATION

To demonstrate how Alpha-Beta pruning is used to optimize chess engine, a simple chess engine is created based on the following steps: (source: https://lichess.org/@/JoaoTx/blog/making-a-simple-chess-engine/fGBIAfGB).

First, an approach to implement a board score involves assigning values to each piece and calculating the board's

value by subtracting the total value of the Black pieces from the total value of the White pieces. A pawn is worth 100, a bishop and a knight are worth 300 each, a rook is worth 500, a queen is worth 900, and the king is worth 1000.

```
def simple_board_score(board):
    if board.is_checkmate():
        if board.turn:
            score = -np.inf #white was checkmated
        else:
            score = np.inf #black was checkmated
    elif its_draw(board):
        score = 0.0
    else:
        score = 0.0
        for (piece, value) in [(chess.PAWN, 100),
                               (chess.BISHOP, 300),
                               (chess.KING, 1000),
                               (chess.QUEEN, 900),
                               (chess.KNIGHT, 300),
                               (chess.ROOK, 500)]:
            score = (score +
                     len(board.pieces(piece,
chess.WHITE))*
                     value)
            score = (score -
                     len(board.pieces(piece,
chess.BLACK))*
                     value)
    return score
```

Then, minimax is implemented without Alpha-Beta pruning to the position of the board up to a certain depth.

```
def minimax (board, depth):
    if (depth==0) or (board.is_game_over()) or
(its_draw(board)):
        return simple_board_score(board)
    if board.turn: # White's turn
        opt_value = -np.inf
        for move in board.legal_moves:
            board.push(move)
            ## White maximizes the board score
            opt_value =
np.max([opt_value,minimax(board, depth - 1)])
            board.pop()
        return opt_value
    else: # Black's turn
        opt_value = np.inf
        for move in board.legal_moves:
            board.push(move)
            ## Black minimizes the board score
            opt_value =
np.min([opt_value,minimax(board, depth - 1)])
            board.pop()
        return opt_value
```

Next, as a comparison, Alpha-Beta pruning is used in minimax algorithm. This is the implementation of the Alpha-Beta pruning.

```
def alpha_beta(board, depth, alpha, beta):
    if (depth==0) or (board.is_game_over()) or
(its_draw(board)):
        return simple_board_score(board)
    if board.turn: # White's turn
        opt_value = -np.inf
        for move in board.legal_moves:
            board.push(move)
            ## White maximizes the board score
            opt_value = np.max([opt_value,
                    alpha_beta(board, depth - 1, alpha,
beta)])
            board.pop()
            if opt_value > beta:
                break # beta cutoff
            alpha = np.max([alpha, opt_value])
        return opt_value
    else: # Black's turn
        opt_value = np.inf
        for move in board.legal_moves:
            board.push(move)
            ## Black minimizes the board score
            opt_value = np.min([opt_value,
                    alpha_beta(board, depth - 1, alpha,
beta)])
            board.pop()
            if opt_value < alpha:
                break # alpha cutoff
            beta = min([beta, opt_value])
        return opt_value
```

Finally, we can get all of the evaluation scores for every possible move from the given position.

```
def get_legal_moves_scores(board,depth):
    score = []
    moves = []
    for move in board.legal_moves:
        board.push(move)
        score.append(alpha_beta(board, depth, -np.inf,
np.inf))
        moves.append(move)
        board.pop()
    ## end for
    legal_moves_scores = pd.DataFrame(
                        {'Score':score,'Move':moves}
                       ).sort_values('Score')
    return legal_moves_scores
depth = 4
legal_moves_scores =
get_legal_moves_scores(board, depth)

for index, row in legal_moves_scores.iterrows():
    print(f"Move: {row['Move']}, Score:
{row['Score']}")
```

IV. RESULT

Based on the implementation, we try to evaluate the board from Figure 8 and determine the best move for White. The board evaluation involves calculating the positional advantage using the assigned piece values and assessing the outcomes of potential moves. The evaluation helps identify the most favorable move for White at the given depth.

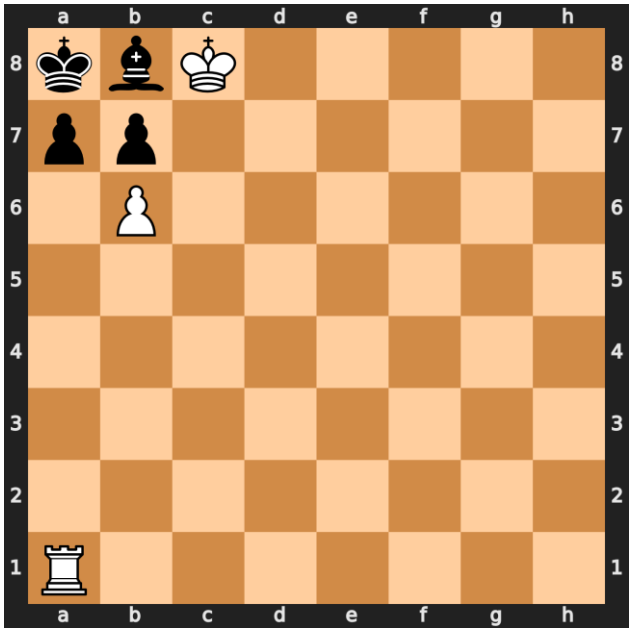

Figure 8. Chess Board to be Evaluated

The game tree for depth = 1 is shown in Figure 9, illustrating the initial set of possible moves for White.

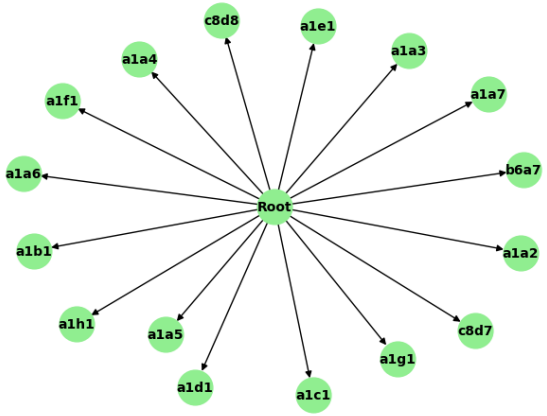

Figure 9. All possible moves for White from the position in Figure 8.

Using minimax, with or without alpha-beta pruning, does not affect the outcome of the position evaluation. At a depth of 3, we obtain the following results.

Table 1. Evaluation of every possible move for White with depth = 3

Move	Score
a1a7	-100.0
a1f1	0.0
a1c1	0.0
a1d1	0.0
a1e1	0.0
a1h1	0.0
a1g1	0.0
a1b1	100.0
a1a5	100.0
a1a2	100.0
a1a4	100.0
a1a3	100.0
c8d8	100.0
c8d7	100.0
b6a7	100.0
a1a6	Inf

Table 1 illustrates that the move Rook a1 to a6 is the optimal move, yielding an evaluation score of infinity, signifying a forced checkmate following the move. One of the lines leading to checkmate can be seen on Table 2.

Table 2. Evaluation of the Best Moves from the Initial Position (Figure 8) to the End of the Game

Move	Score
a1a6 (White)	inf
b7a6 (Black)	inf
B6b7 (White) (Checkmate)	inf

This result is consistent with the evaluation from Stockfish 16 NNUE on Lichess, where the same move, Rook a1 to a6, leads to a forced mate in 2 moves, at a depth of 99. Interestingly, this comparison shows that achieving the best move does not always require an extensive search depth, especially in situations where there are limited possible moves within the game tree. In such cases, reducing the search depth does not compromise the accuracy of the evaluation, thus allowing for significant computational savings.

This highlights the importance of selective pruning in reducing unnecessary calculations, particularly when the game tree's branching factor is small, or the position is already simplified. In this scenario, the search depth required to find the best move can be relatively shallow, and the algorithm can efficiently converge on the best decision without an exhaustive exploration of all possibilities.

To compare the running time and the number of nodes evaluated on different depths, we run the program using minimax without Alpha-Beta pruning implemented and then compare it with the one using Alpha-Beta pruning.

Table 2. Running time without Alpha-Beta Pruning

Depth	Running Time (s)
1	0.027
2	0.189
3	2.4
4	33
5	557.9

Table 3. Running time with Alpha-Beta Pruning

Depth	Running Time (s)
1	0.017
2	0.189
3	1.2
4	15.4
5	162.5

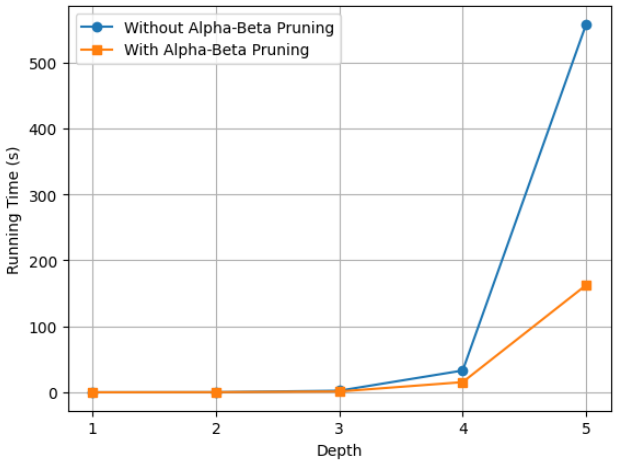

Figure 10. Comparison of Running Time with and without Alpha-Beta Pruning

Table 4. Total Nodes Visited without Alpha-Beta Pruning

Depth	Total Nodes
1	17
2	144
3	988
4	23152
5	353292

Table 5. Total Nodes Visited with Alpha-Beta Pruning

Depth	Total Nodes
1	17
2	130
3	2096
4	2883
5	30903

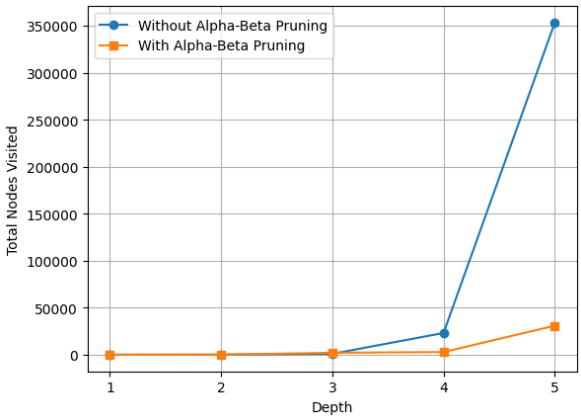

Figure 10. Comparison of Total Nodes Visited with and without Alpha-Beta Pruning

The results demonstrate the significant performance improvements brought by Alpha-Beta Pruning in the context of search depth and node evaluation. Table 3 reveals a consistent reduction in running time with the use of Alpha-Beta Pruning compared to the version without pruning. For instance, at depth 5, the running time decreased from 557.9 seconds without pruning to 162.5 seconds with pruning, indicating a notable reduction in computational cost. The comparison of running times, illustrated in Figure 10, further shows the efficiency of Alpha-Beta Pruning, especially as the depth increases, highlighting the exponential reduction in time complexity.

In terms of node evaluation, the tables clearly show the effect of pruning in reducing the number of nodes visited during the search process. Without Alpha-Beta Pruning (Table 4), the number of nodes grows exponentially with the depth, reaching a staggering 353,292 nodes at depth 5. However, with Alpha-Beta Pruning (Table 5), the number of nodes visited is significantly reduced—decreasing to just 30,903 nodes at depth 5. This reduction not only accelerates the search process but also makes the algorithm more feasible for deeper searches, demonstrating the pruning technique's effectiveness in optimizing the exploration of the search space.

V. Conclusion

The integration of game trees, minimax, and alpha-beta pruning significantly enhances the performance of chess engines by optimizing the decision-making process. Game trees form the foundational structure, modeling the sequence of moves and counter-moves, while the minimax algorithm provides a framework for evaluating the best possible moves under the assumption that both players act optimally. Alpha-beta pruning optimizes minimax by reducing the number of nodes that need to be evaluated, thus eliminating irrelevant branches and improving efficiency. This combination allows the chess engine to explore deeper levels of the game tree, enhancing search

depth and reducing processing time without compromising the accuracy of move evaluation. The findings confirm the effectiveness of using alpha-beta pruning to optimize the minimax algorithm, making it an essential tool for real-time decision-making and strategic play in computational chess.

VI. APPENDIX

To provide a clearer understanding of the concepts and methods discussed in this paper, a supplementary video has been prepared. Link: https://youtu.be/P6OwINVf6OU

VII. ACKNOWLEDGMENT

The Author would like to express gratitude to Mr. Rinaldi Munir and Mr. Rila Mandala, the lecturers for Discrete Math at Bandung Institute of Technology, for his guidance and comprehensive teaching of the subject, which provided the foundation for understanding the concepts applied in this paper.

The Author also acknowledges the various sources of information that greatly contributed to the completion of this paper. These include academic journals, articles, online resources, and publicly available code repositories, all of which offered invaluable insights and practical tools necessary for the development of this work.

.

REFERENCES

[1] R. Munir, "Graf Bagian 1," Department of Informatics, Institut Teknologi Bandung, 2024-2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf.

[2] R. Munir, "Pohon Bagian 1," Department of Informatics, Institut Teknologi Bandung, 2024-2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf.

[3] R. Munir, "Pohon Bagian 2," Department of Informatics, Institut Teknologi Bandung, 2024-2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/24-Pohon-Bag2-2024.pdf.

[4] W. Pijls and A. de Bruin, "Game tree algorithms and solution trees," *Journal of Artificial Intelligence*, vol. 32, no. 4, pp. 305-312, 2021.

[5] Y. R. S. Kurniawan and M. M. Putra, "Perbandingan metode optimasi algoritma minimax pada permainan catur," *Alu, Jurnal Sistem Informasi dan Teknologi*, vol. 5, no. 1, 2024. [Online]. Available: https://journal.ubm.ac.id/index.php/alu.

[6] J. Tx, "Making a simple chess engine," *Lichess.org*, [Online]. Available: https://lichess.org/@/JoaoTx/blog/making-a-simple-chess-engine/fGBIAfGB.

STATEMENT

Hereby, I declare that this paper I have written is my own work, not a reproduction or translation of someone else's paper, and not plagiarized.

Bandung, 04 January 2024

Henry Filberto Shenelo 13523108