A Decision Tree-Based Approach to Optimal Move Selection in Chess Middlegame Using Minimax Algorithm with Stockfish Engine

Ibrahim Ihsan Rasyid - 13522018[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13522018@std.stei.itb.ac.id

***Abstract*—The minimax algorithm in game theory serves as a decision-making rule to maximize the minimum advantage available. This is due to the nature of zero-sum games, where both players strive to maximize their own gains. In a zero-sum game like chess, when one player, say player A, seeks to maximize their advantage, its opponent, say player B, will consistently aim to minimize player A's advantage. This principle finds application in chess, where players often encounter challenges in determining the best moves, especially during the middlegame phase. Therefore, this paper delves into the application of the minimax algorithm in determining optimal moves in chess, discussing its implementation, and presenting the results of testing the implemented algorithm.**

***Keywords*—best move, middlegame, minimax algorithm, tree.**

I. Introduction

Chess is one of the most popular board games since its discovery around the 7th century, evolving into competitive play on the world stage for the first time in 1834 up to the present modern era. In chess, the term "middlegame" is recognized, referring to the phase where majority of player's pieces and pawns have been developed. Most of the times, players find it challenging to determine the best next move due to the multitude of possibilities that can unfold after each move. Poor move calculation can have long-term effects and lead to defeat in the end. It is not easy for humans to perform calculations to determine the best move.

Therefore, humans strive to develop chess engines to assist in analyzing previously completed games. Numerous chess engines have already been developed, so many that since 2010, the Top Chess Engine Championship (TCEC) has been held, which is a competition among chess engines. To date, TCEC has conducted 25 seasons, with one of the chess engines, Stockfish, emerging victorious in 15 seasons and currently reigning as the best chess engine.

The algorithm employed by Stockfish, eventually replaced by NNUE (Efficiently Updated Neural Networks) due to its widespread adoption across various chess engines, includes the Minimax Algorithm and Alpha-Beta Pruning. This paper focuses solely on the Minimax Algorithm and the application of tree concepts in constructing this algorithm.

The Minimax Algorithm is a decision-making rule designed to minimize the potential for the worst-case scenario. This algorithm is commonly applied to zero-sum games such as tic-tac-toe, checkers, shogi, and of course chess. In the context of these games, minimax entails maximizing the potential for the smallest advantages. Simply put, one player makes moves to gain maximum advantage for themselves, while the opposing player makes moves to diminish the advantage of their opponent. Hence, the evaluation of the board position in a game typically holds a positive value for one player and negative value for their opponent. In chess, the advantage of the white player is reflected in a positive evaluation, and vice versa. An infinite value signifies that one player can win the game within a few moves, usually written as M4, means there is a checkmate in 4 moves.

II. Theoretical Foundation

A. Graph

Graph G is defined as a pair of sets (V, E), expressed using the notation G = (V, E), where V is a non-empty set of vertices/nodes, and E is a set of edges connecting pairs of nodes. Geometrically, a graph is depicted as a collection of nodes in a two-dimensional plane connected by a set of lines

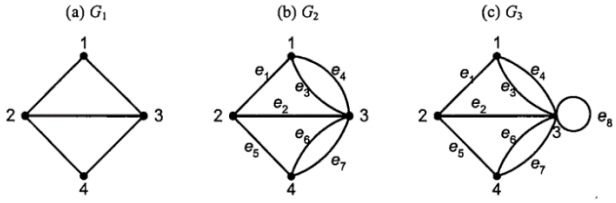

Fig. 2.1 (a) simple graph, (b) unsimple graph, (c) pseudo-graph (Source: Matematika Diskrit Edisi 3 2010 (Dr. Rinaldi Munir))

Based on the presence or absence of loops or multiple edges in a graph, graphs can generally be classified into two types:

a. Simple Graph
A simple graph is a graph that does not have loops or multiple edges. Examples can be seen in Fig. 2.1 (a)

b. Unsimple Graph
An unsimple graph is a graph that can have loops and/or multiple edges. Here we can classify an unsimple graph into:
   - Multi-graph
     Multigraph is a graph that contains multiple egdes, those are two or more edges that connect the same pair of nodes. Examples can be seen in Fig. 2.1 (b)

- Pseudo-graph
  Pseudo-graph is a graph that contains loops, that is an edge that connects a node to itself. Examples can be seen in Fig 2.1 (c)

Another way to categorize graphs is based on the orientation of their edges. Using this approach, graphs are generally classified into two types:

a. Undirected Graph
   A graph whose edges have no directional orientation is referred to as an undirected graph. In an undirected graph, the order of the pairs of nodes connected by an edge is not considered. Thus, for an edge connecting two nodes u and v in a graph, (u, v) is equivalent to (v, u). Examples can be seen in Fig 2.1.
b. Directed Graph
   A graph in which each edge is given a directional orientation is termed as a directed graph. In a directed graph, for an edge connecting two nodes u and v, (u, v) and (v, u) represent distinct pairs. In the case of the edge (u, v), node u is referred to as the origin node, and node v is termed the terminal node. Examples can be seen in Fig 2.2.

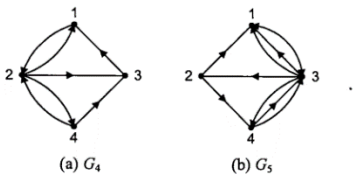

Fig. 2.2 Examples of directed graph
(Source: Matematika Diskrit Edisi 3 2010 (Dr. Rinaldi Munir))

There are several terminologies related to graphs. Below are some terms that will be used in this paper.

a. Path
   A path of length n from the initial vertex v_0 to the destination vertex v_n in the graph G is a sequence alternating between vertices and edges in the form v_0, e_1, v_1, e_2, …, v_{n-1}, e_n, v_n, such that $e_1 = (v_1, v_2)$, …, $e_n = (v_{n-1}, v_n)$ are the edges of the graph G.
   If the graph under consideration is a simple graph, then we only need to express the path as a sequence of vertices. For example, refer to Fig. 2.3. We can express the path for graph G_1 as 1, 2, 3, 4 instead of 1, 12, 2, 23, 3, 34.
b. Circuit
   A path that starts and ends at the same vertex is called a circuit. Refer to Fig 2.3 for an example. Note that graph G_1 has a path 1, 2, 4, 3, 1. Because the path starts and ends at the same vertex, graph G_1 is considered has a circuit. Similarly in graph G_2, we can see that G_2 has a path 1, e_2, 2, e_4, 3, e_3, 1 that starts and ends at the same vertex. It means that G2 has a circuit.

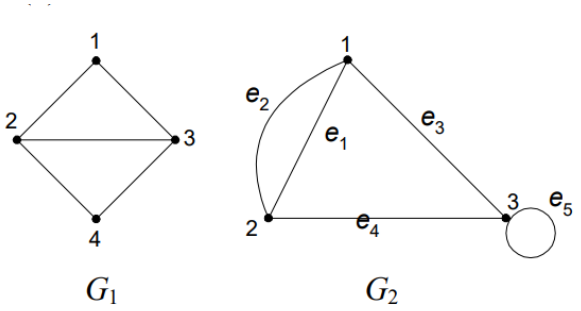

Fig. 2.3 Two graphs G_1 and G_2
(Source: Matematika Diskrit Edisi 3 2010 (Dr. Rinaldi Munir))

c. Connected
   An undirected graph G = (V, E) is called connected if, for every pair of vertices u and v within the set V, there exists a path from u to v. If not, G is referred to as a disconnected graph. For example, in Fig. 2.4 below, we can see that there is no path from vertex 1 to 5. It means that the graph below is a disconnected graph.

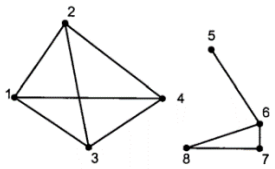

Fig. 2.4 Example of disconnected graph
(Source: Matematika Diskrit Edisi 3 2010 (Dr. Rinaldi Munir))

B. Tree

A tree is defined as a connected undirected graph that does not contain any circuits. In this definition, we can observe two essential properties of a tree: connectedness and absence of circuits. Notice the examples in Fig. 2.5 below. We can see that both graphs G_1 and G_2 are trees, as they are both connected graphs and do not contain any circuit. While graph G_3 is not a tree because it contains a circuit a, d, f, a, graph G_4 also not a tree because it is not a connected graph

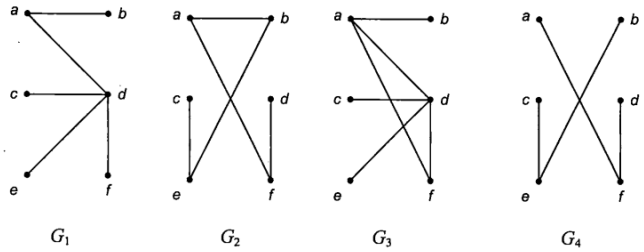

Fig. 2.5 Examples of tree (G_1 and G_2) and not-tree (G_3 and G_4)
(Source: Matematika Diskrit Edisi 3 2010 (Dr. Rinaldi Munir))

A tree in which one of its vertices is treated as the root, and its edges are directed away from the root, is called a rooted tree. In many applications of trees, a specific vertex is designated as the root, forming a rooted tree. Any arbitrary unrooted tree can be transformed into a rooted tree by selecting a vertex as the root. Refer to the example in Fig. 2.6 for illustration. Here we can transform the tree into a rooted tree by selecting vertex b or vertex e as the root.

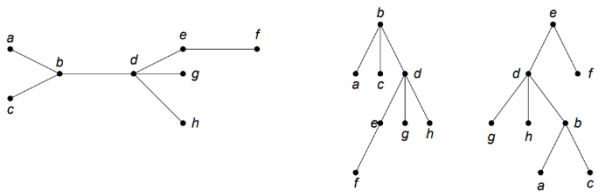

Fig. 2.6 A tree and its rooted tree produced by choosing a vertex as the tree root
(Source: Matematika Diskrit Edisi 3 2010 (Dr. Rinaldi Munir))

There are several terminologies related to rooted tree. We will discuss some terms that used in this paper.

a. Child and Parent
Let *x* be a node in a rooted tree. Node *y* is said to be the child of *x* if an edge from *x* to *y* exists. Therefore, *x* is parent of *y*. As an example, in Fig. 2.7, node d is a child of node a, and node a is the parent of node b, c, and d.

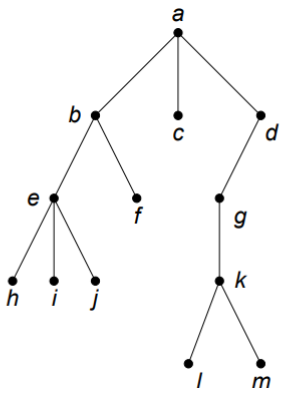

Fig. 2.7 A rooted tree
(Source: Matematika Diskrit Edisi 3 2010 (Dr. Rinaldi Munir))

b. Path
Path from vertex v_1 to vertex v_k is a series of vertices v_1, v_2, …, v_k such that v_i is the parent of v_{i+1} for $1 <= i < k$. From Fig. 2.7, the path from a to h is a, b, e, h.

c. Descendant and Ancestor
If there exists a path from vertex *x* to vertex *y*, then *x* is an ancestor of *y*, and *y* is a descendant of *x*. In Fig. 2.7, d is an ancestor of m and therefore m is a descendant of d.

d. Subtree
Let *x* be a node in a tree T. A subtree with x as the root is a subgraph *T' = (V', E')* such as *V'* contains *x* and its descendant and *E'* contains the edges of every path that starts from *x*. As an example, in Fig. 2.7, we can take node b as the root of our subtree so that we have subtree *T' = (V', E')* with *V'* = {*b, e, f, h, i, j*} and E' = {(*b, e*), (*b, f*), (*e, h*), (*e, i*), (*e, j*)}.

e. Degree
The degree of a node *x* in a rooted tree is the number of children of *x*. In Fig. 2.7, node d has a degree of 1, and node b has a degree of 3.

f. Leaf
A node that has a degree of zero, meaning it has no children, is called a leaf. In Fig. 2.7, node h, i, j, l, m are leaves.

g. Level and Height
The root of a tree is at level 0, while the level of any other node is defined as 1 plus the length of the path from the root to that node, while height of a tree is the maximum length of the path from the root to a leaf. In Fig. 2.7, the height of the tree is 4.

A rooted tree in which each node has at most n children is called an n-ary tree. A binary tree is an n-ary tree with n = 2. In a binary tree, the children of a node are referred to as the left child or the right child. A subtree that is the left child of a tree is called the left subtree, and a subtree that is the right child of a tree is called the right subtree.

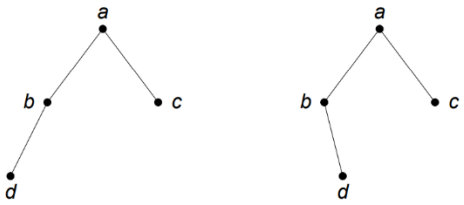

Fig. 2.8 A different binary tree
(Source: Matematika Diskrit Edisi 3 2010 (Dr. Rinaldi Munir))

C. Minimax Algorithm in Game Theory

A strategic game is a model of interactive decision-making, where each participant selects their course of action definitively, and these decisions occur simultaneously. The model includes a finite set *N* of players, and for each player *i*, there exists a set A_i of actions along with a preference relation on the set of action profiles. An outcome is referred to as an action profile $a = (a_j)_{j \in N}$, and the set of outcoumes $x_{j \in N} A_j$ is denoted as A.

A strategic game $(\{1, 2\}, (A_i), (\succcurlyeq_A))$ is strictly competitive if for any $a \in A$ and $b \in A$ we have $a \succcurlyeq_1$ if and only if $b \succcurlyeq_2$. A strictly competitive game is sometimes called zero-sum because if player 1's preference relation $\succcurlyeq_1$ is represented by the payoff function u_1 then player 2's preference relation is represented by u_2 with $u_1+u_2 = 0$.

A player *i* is said to maxminimizes if he chooses an action that is best for him on the assumption that whatever he does, player *j* will choose her action to hurt him as much as possible.

Let $(\{1, 2\}, (A_i), (u_i))$ be a strictly competitive game. The action $x^* \in A_1$ is a maxminimizer for player 1 if

$$\min u1(x*, y) \geq \min u1(x, y)$$

for all $x \in A_1$. Similarly, the action $y^* \in A_2$ is a maxminimizer for player 2 if

$$\min u2(x, y*) \geq \min u2(x, y)$$

for all $y \in A_2$. In words, a maxminimizer for player *i* is an action that maximizes the payoff that player *i* can guarantee. Next on we will refer this as minimax algorithm.

Below is an illustration of a simple example of the minimax algorithm depicted in a decision-tree

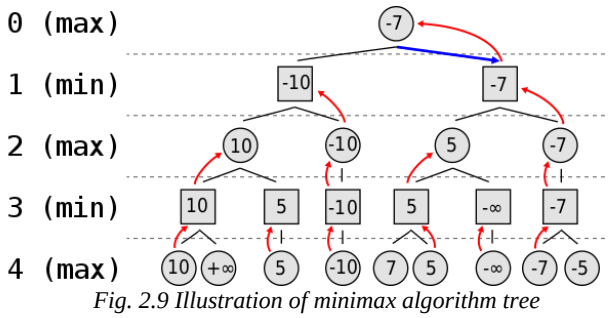

Fig. 2.9 Illustration of minimax algorithm tree

(Source: https://en.wikipedia.org/wiki/Minimax)

D. Chess

Chess is a board game which is played between two opponents (light-colored/white and dark-colored/black) who move their pieces alternately, with the objective is to place the opponent's king in "under attack" in such a way that the opponent does not have any legal move. The player who achieves this goal is said to have "checkmated" the opponent's king and to have won the game, while the opponent king whose king has been checkmated has lost the game.

The middlegame in chess is the phase between the opening and the endgame. Typically, the middlegame begins when players have developed a significant portion of their pieces, brought their king to safety, and concludes when only a few pieces remain on the board. Middlegame theory is considered less developed compared to opening and endgame theories. This is because the positions of pieces in the middlegame often vary significantly from one to another, making it much more challenging to memorize the various variations compared to openings and endgames.

III. IMPLEMENTATION METHOD

In this section, I will demonstrate the approach I took to implement the minimax algorithm to determine the best move in chess middlegames. The implementation is done with Python, utilizing chess module available in Python

A. A View on Python Chess Module

Python language provides a module for chess computations called the chess module. Documentation is available on the following link. This module offers various features utilized in this paper, such as Board, Move, Portable Game Notation (PGN) parsing and writing, Game model, and engine analysis

1. Board
   Board is a Python class that, upon initialization, generates the starting position of a standard chess game. An object of the Board class can be manipulated by making moves or by copying it to another variable. Player is represented as a boolean, meaning that white is true and black is false. Fig.3.1 shows the output when we print a board object. Uppercase letters represents white's pieces, vice versa.

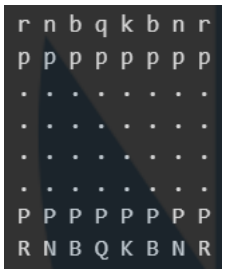

Fig. 3.1 Example of Board print output

2. Move
   The Move class represents a move from one square to another using the Universal Chess Interface (UCI) notation. For example, "e3e5" denotes a move of a piece from square e3 to e5.
3. PGN
   PGN (Portable Game Notation) is a standard text format for representing a chess game. Parsing a PGN file can be performed, and it will return a Game object representing the chess game.
4. Game
   A game is represented as a tree of various moves, with each node being a state or position on the board. The GameNode is a class that represents a node in the game tree.
5. Engine analysis
   In this module, there is a function available to analyze a position on the board and return a numerical representation of the analysis result. The analysis is performed by an external chess engine such as Stockfish, Torch, and others.

B. Implementation of Minimax Algorithm

In the implementation of the minimax algorithm, a Python class named MinimaxEngine is created, featuring a method called minimax. The minimax method takes inputs such as a Board object, a boolean value, an integer, and engine.SimpleEngine, then returns a list of integers and Move objects. The following is an excerpt of the code.

```
def minimax(self, board: Board, player: bool,
depth: int, evaluator: engine.SimpleEngine,
alpha=-inf, beta=inf):

        if depth == 0 or board.is_checkmate() or
board.is_stalemate():

            if player:
                return [evaluator.analyse(board,
engine.Limit(depth=15))["score"].white().score(ma
te_score=MATE_SCORE), None]

            else:
                return [evaluator.analyse(board,
engine.Limit(depth=15))["score"].black().score(ma
te_score=MATE_SCORE), None]

        moves = list(board.legal_moves)

        if board.turn == player:
            maxScore, bestMove = -inf, None

            for move in moves:
                test_board = board.copy()
                test_board.push(move)

                score = self.minimax(test_board,
player, depth-1, evaluator, alpha, beta)

                alpha = max(alpha, score[0])
                if beta <= alpha:
                    break

                if score[0] >= maxScore:
                    maxScore = score[0]
                    bestMove = move
```

```
        else:
            minScore, bestMove = inf, None

            for move in moves:
                test_board = board.copy()
                test_board.push(move)

                score = self.minimax(test_board,
player, depth-1, evaluator, alpha, beta)

                beta = min(beta, score[0])
                if beta <= alpha:
                    break

                if score[0] <= minScore:
                    minScore = score[0]
                    bestMove = move

            return [minScore, bestMove]
```

We can observe that in minimax method, the function is called recursively with the base cases being when depth equals 0, the game is in a checkmate state, or the game is a stalemate. If the base cases are not met, the program proceeds with the search based on the current state of the board. If it is the player's turn, the program looks for the maximum value among each move from the board node; otherwise, it looks for the minimum value. The search returns a list of integers representing the maximum or minimum value and a Move object as the best move

IV. Implementation Testing and Result Discussion

In this section, the test results from 3 samples will be displayed. Each test is conducted using a depth of 4. Then, we will discuss about the result on the test.

A. Unit Testing

We use a game between Hikaru Nakamura against Liam Vrolijk as our first test to our program.

Fig. 4.1 Hikaru Nakamura (Hikaru) vs Liam Vrolijk (LiamVrolijk), 13 October 2020
(Source: chess.com)

In this game, Hikaru finds the best move Bb5 (e2b5). We save the PGN file as test-2.pgn.

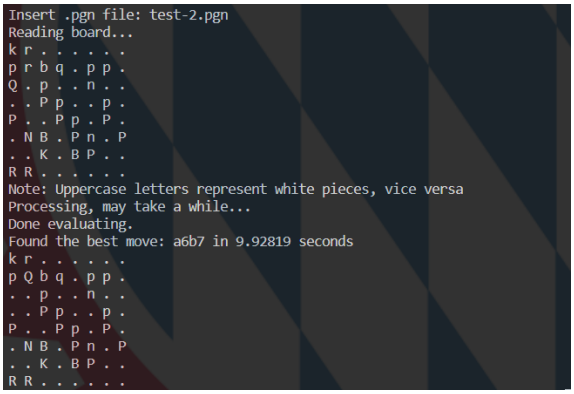

Fig. 4.2 Test 1 Result

In above figure, we see that the minimax's best move is Qxb7 (a6b7)

Next, we do another test from my own game online in chess.com.

Fig. 4.3 r0drig0r0sa vs sunegoo, 5 December 2023
(Source: chess.com)

Here, I honestly felt confused while I am playing. In game review, I figured out that the best move for me is Rxg2 (g6g2). We save the PGN file as test-3.pgn

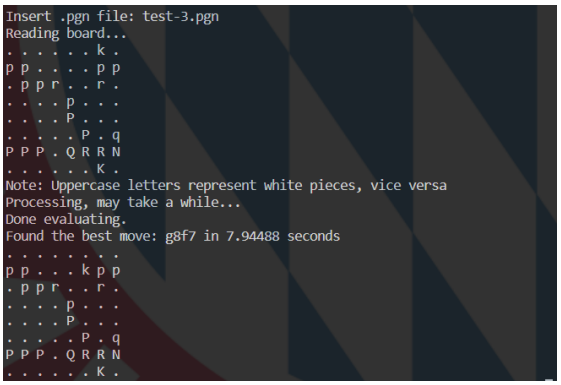

Fig. 4.4 Test 2 Result

In above figure, minimax evaluates that the best move is Kf7 (g8f7).

Lastly, we do another test from another own game I recently played online.

Fig. 4.5 Silvio371 vs sunegoo, 10 December 2023 (Source: chess.com)

In this position, my opponent made a mistake by taking the pawn on 12. Bxe4, leads to a loss in material after 12. .. Qxd1 13. Rxd1 Nxe4. For my opponent, the best move is to check the king with Bc4+ (d3c4). We save the PGN file of this position as test-4.pgn

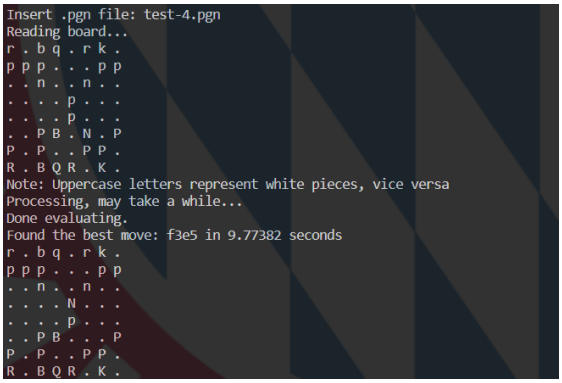

Fig. 4.6 Test 3 Result

In above figure, we can see that minimax's best move is Nxe5 (f3e5).

B. Result Discussion

It can be observed that several experiments have been conducted, and the results do not align with the expected optimal moves. When each result of the experiments is tested using an analysis board available on various websites like lichess.org, the outcomes significantly deviate from the best moves. For Test 1, the evaluation drops from +5.0 to -2.7. In Test 2, the evaluation decreases from +1.8 to +3.8. Finally, in Test 3, the evaluation drops from +0.6 to -3.9. It is important to note that in Test 2, the calculations were performed considering the perspective of the black pieces, so a positive value indicates a disadvantage for the black side.

Many aspects may affect this result, including:

1. Game complexity
   Chess is highly complex, especially during the middlegame phase. The Minimax algorithm, while powerful, may struggle to accurately model all potential game scenarios.
2. Limitations in heuristic evaluation
   The heuristic evaluation used in the Minimax algorithm might not be sophisticated enough to identify optimal moves, particularly if there is a lack of knowledge about specific strategies or positions.
3. Limited search depth
   If the Minimax algorithm only searches a few moves ahead (limited search depth), it might be unable to assess the long-term consequences of a move accurately.
4. Time and resource constraints
   Optimal searching in Minimax requires significant computation time, especially for a large decision tree. Time and resource constraints may force the algorithm to make rough estimations.
5. Lack of parameter tuning
   The algorithm has parameters that need fine-tuning, and inadequate parameter tuning can lead to inaccurate evaluations.

V. Conclusion

Finding the best move in chess middlegame using Minimax Algorithm is a possible task to do and tends to be effective as it can compare many possibilities of moves in a certain depth. It searches every possible move from a position until a certain depth, calculates every single position reached by the moves, and returns the best move possible in that position.

Some flaws are encountered in the implementation, such as bad analysis and the algorithm time complexity. Despite the weaknesses in the implementation, we can see the implementation of tree in finding the best move in chess.

VI. Appendix

Program used in this paper can be seen here

VII. Acknowledgment

All Praise and gratitude to Allah Subhanahu wa Ta'ala, for by His mercy, the paper titled "A Decision Tree-Based Approach to Optimal Move Selection in Chess Middlegame Using Minimax Algorithm with Stockfish Engine" has been successfully completed. Also, gratitude is extended to the lecturer for IF2120 Discrete Mathematics course, Dr. Nur Ulfa Maulidevi, S.T, M.Sc., for the guidance and motivation provided throughout her tenure in teaching the students.

References

[1] handbook.fide.com. 2023. FIDE Laws of Chess. [online] Available at: <https://handbook.fide.com/chapter/E012023> [Accessed 7 December 2023 21:08]

[2] Munir, Rinaldi. 2010. *Matematika Diskrit Edisi 3.* Bandung: Penerbit INFORMATIKA Bandung.

[3] Osborne, Martin J., Ariel Rubenstein. 1994. *A Course in Game Theory.* Cambridge, Massachusetts: The MIT Press

Statement of Originality

I hereby declare that this paper is my own writing, not an adaptation, or translation of someone else's paper, and not plagiarized.

Bandung, 11 Desember 2023

Ibrahim Ihsan Rasyid - 13522018