Analysis of Iterative Deepening Search and Breadth-First Search Algorithm Complexity Implemented in Minimax for Solving Mate-in-N Chess Puzzle

Ibrahim Ihsan Rasyid - 13522018
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: 13522018@std.stei.itb.ac.id

***Abstract*—The minimax algorithm is a fundamental approach in game theory and artificial intelligence, particularly in solving chess puzzles. In this study, we investigate the complexity of two search algorithms, Iterative Deepening Search (IDS) and Breadth-First Search (BFS), when implemented within the minimax framework for solving the Mate-in-N chess puzzle. The Mate-in-N puzzle involves finding a sequence of moves that forces checkmate in a specified number of moves. This paper contributes to the understanding of search algorithm efficiencies in combinatorial game problems, presenting a nuanced evaluation that aids in selecting appropriate strategies for different computational scenarios in chess and beyond.**

Keywords—breadth-first search; iterative deepening search; checkmate; minimax

I. INTRODUCTION (*HEADING 1*)

Chess is one of the most popular board games, originating in the 7th century and evolving globally until the first competition was held in 1834. It remains well-known today. The increasing competitiveness of chess has heightened the urgency of chess education, with puzzles being one of the most frequently used teaching methods. Among the various types of puzzles, the mate-in-N puzzle is particularly effective in enhancing awareness to checkmate opportunities during the game.

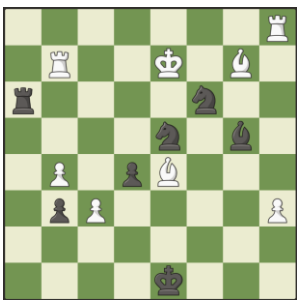

Fig. 1.1 Illustration of mate-in-n puzzle problem
(Source: chess.com)

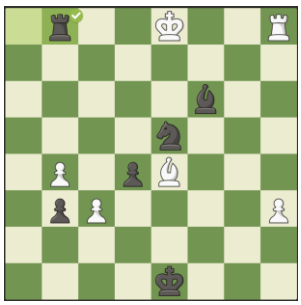

Fig. 1.2 Illustration of a solved mate-in-n puzzle
(Source : chess.com)

In the modern era, numerous chess engines have been developed to help humans gain a deeper understanding of chess and to attempt to find concrete solutions to chess problems. The rapid development of chess engines inspired the idea of hosting tournaments between these engines, leading to the inaugural Top Chess Engine Competition (TCEC) in 2010. To date, TCEC has been held for 25 seasons, with the chess engine Stockfish winning 15 seasons and being acclaimed as the strongest chess engine currently.

One of the algorithms frequently used in chess engines is the minimax algorithm. The minimax algorithm is a decision-making algorithm widely used in game theory and artificial intelligence to minimize the possibility of losing in the worst-case scenario. This algorithm is commonly applied in zero-sum games or turn-based games with two players, such as tic-tac-toe, checkers, shogi, and chess.

In these games, minimax aims to maximize the smallest possible gain. In other words, a player strives to maximize their advantage while the opponent attempts to minimize it. Typically, the evaluation of the board position is a positive number for one player and a negative number for the other. In chess, a positive value signifies an advantage for the player with the white pieces, and vice versa.

II. THEORETICAL FOUNDATION

A. Graph Traversal

Graph traversal algorithms systematically visit the nodes within a graph, ensuring that every node is explored in an organized manner. Assuming the graph is connected, meaning there is a path between any two nodes, it serves as an effective representation of complex problems. Traversing a graph thus involves searching for a solution to the problem encoded by the graph's structure. This process is fundamental in various applications, such as network analysis, pathfinding in games, and solving puzzles. Key traversal methods include Depth-First Search (DFS) and Breadth-First Search (BFS), each offering distinct strategies and advantages depending on the specific problem requirements.

B. *Breadth-First Search*

Breadth-First Search (BFS) is one of the algorithms used for graph traversing. It proceeds in a concentric manner by visiting first all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it, and so on, until all the vertices in the same connected component as the starting vertex are visited. If there remain unvisited vertices, the algorithm must be restarted at an arbitrary vertex of another connected component of the graph

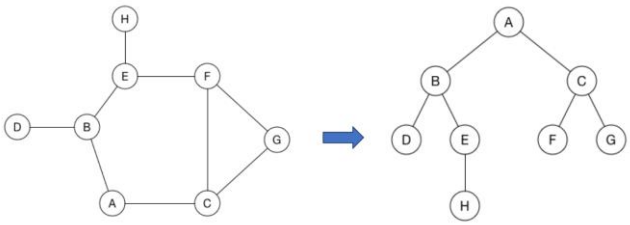

Fig. 2.1 Illustration of BFS algorithm for traversing an entire graph (Source: [1])

In Fig. 2.1, the sequence of nodes visited by BFS from node A would be A, B, C, D, E, F, G, H.

BFS is a fundamental graph traversal algorithm with notable properties regarding completeness, optimality, time complexity, and space complexity. BFS is complete, meaning it will find a solution if one exists, because it explores all possible nodes level by level. It is also optimal when the cost to reach a node is uniform, as it always finds the shortest path to the solution. The time complexity of BFS is O(b^d), where b is the branching factor (average number of successors per node) and d is the depth of the shallowest solution. However, BFS has a significant drawback in its space complexity, which is also O(b^d). This high space requirement arises because BFS stores all nodes at the current level before moving on to the next, leading to exponential growth in memory usage as the search depth increases.

C. *Depth-First Search*

Depth-First Search (DFS) is another algorithm used for graph traversing. On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in. (If there are several such vertices, a tie can be resolved arbitrarily. As a practical matter, which of the adjacent unvisited candidates is chosen is dictated by the data structure representing the graph. In our examples, we always break ties by the alphabetical order of the vertices.) This process continues until a dead end—a vertex with no adjacent unvisited vertices— is encountered. At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there. The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end. By then, all the vertices in the same connected component as the starting vertex have been visited. If unvisited vertices remain, the depth-first search must be restarted at any one of them.

In Fig 2.2, the sequence of nodes visited by DFS from node A would be A, B, D, E, H, F, G, C

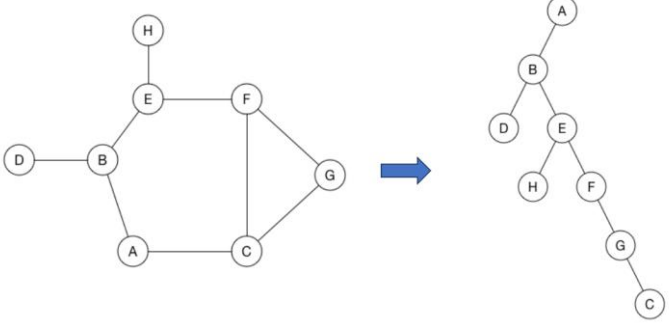

Fig. 2.2 Illustration of DFS algorithm for traversing an entire graph (Source: [1])

DFS is a fundamental graph traversal algorithm with distinct characteristics regarding completeness, optimality, time complexity, and space complexity. DFS is not complete in general because it can get stuck in infinite branches and may fail to find a solution even if one exists. DFS will be complete as long as the value of b is finite, and there is handling for redundant paths and repeated states. It is also not optimal, as it does not guarantee the shortest path to a solution. The time complexity of DFS is O(b^m), where b is the branching factor and m is the maximum depth of the search tree. Despite these limitations, DFS has a significant advantage in terms of space complexity, which is O(bm). This efficiency arises because DFS only needs to store a single path from the root to a leaf node along with the remaining unexplored siblings for each node on the path, resulting in much lower memory usage compared to BFS.

D. *Iterative Deepening Search*

Another algorithm used for graph traversing is Iterative Deepening Search (IDS). IDS performs a series of DFS, progressively increasing the depth cutoff until a solution is found. This approach combines the space efficiency of DFS with the optimality of BFS. The underlying assumption is that most solution nodes are located at deeper levels, making the repeated generation of nodes at higher levels inconsequential. By incrementally deepening the search, IDS efficiently handles large search spaces without the high memory consumption associated with BFS.

IDS is an algorithm that combines the strengths of DFS and BFS, making it complete, optimal, and relatively efficient in terms of time and space complexity. IDS is complete because it incrementally explores all depths, ensuring that it will find a solution if one exists. It is also optimal when the cost to reach a node is uniform, as it guarantees finding the shortest path to the solution by progressively deepening the search. The time complexity of IDS is O(b^d), where b is the branching factor and d is the depth of the shallowest solution. Although it seems that repeatedly exploring nodes might be inefficient, the overhead is minimal since most nodes are near the bottom level. The space complexity of IDS is O(bd), similar to DFS, because it only needs to store the current path and a few sibling nodes at each level, resulting in much lower memory usage compared to BFS.

E. *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 $(\{1, 2\}, (A_i), (\succsim_A))$ is strictly competitive if for any $a \in A$ and $b \in A$ we have $a \succsim_1$ if and only if $b \succsim_2$. A strictly competitive game is sometimes called zero-sum because if player 1's preference relation $\succsim_1$ 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

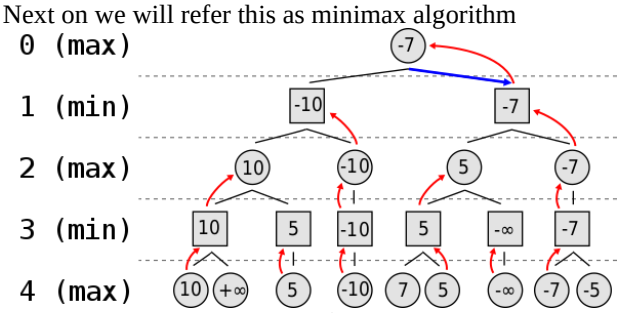

Fig. 2.3 Illustration of minimax algorithm tree
(Source: https://en.wikipedia.org/wiki/Minimax)

F. Chess and Chess Puzzle

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

A chess puzzle is a puzzle set by the composer using chess pieces on a chess board, which presents the solver with a particular task. For instance, a position may be given with the instruction that White is to move first, and checkmate Black in two moves against any possible defence. Most positions which occur in a chess puzzle are 'unrealistic' in the sense that they are very unlikely to occur in over-the-board play.

III. Problem Analysis

In this section, we will discuss how solving a mate-in-n chess puzzle relates to and can be accomplished using IDS and BFS algorithms.

A. Mapping the Problem to the Elements of IDS and BFS

In the problem of solving an n-move checkmate in a chess puzzle, our task is to determine the sequence of moves that leads the game to the desired outcome, specifically when the opponent's king is in checkmate. Initially, there is a starting position with the pieces arranged in such a way that checkmating the opponent's king is possible within a few moves. In this problem, we can treat a game state as a node, and each edge connecting two nodes represents a legal move that changes the board's state. Therefore, IDS and BFS will search with the checkmate condition as the target node.

In searching with IDS, the algorithm will begin from the initial state, then proceed to the next board state by making a legal move from the previous state. The search is conducted in a depth-first manner with an initial depth of 1, incrementing by 2 if a certain depth fails to find the target node. This process continues until the checkmate condition is found.

In searching with BFS, the algorithm will examine each board state resulting from legal moves made from the initial state. If the target node is not found in a particular board state, the algorithm will continue searching from each child of every board state. This process continues until the checkmate condition is found.

It is important to note that in solving this puzzle, only one player is trying to reach the checkmate state, while the opponent will attempt to delay the checkmate. This consideration is crucial because, without it, the quality of the solution would be poor due to unrealistic search results. A rational chess player would not foolishly move their king toward the opponent's pieces, so the opponent must make the best moves, determined with the help of a chess engine.

In fact, the implementation of IDS and BFS in chess is not limited solely to solving n-move checkmate puzzles. However, if the assurance of checkmate within a few moves is not guaranteed, the generation of nodes will be extensive. Therefore, to simplify the implementation in programming, the author specifies the discussion only on solving n-move checkmate puzzles.

B. 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. A dot represents an empty square, uppercase letters represent white's pieces, vice versa

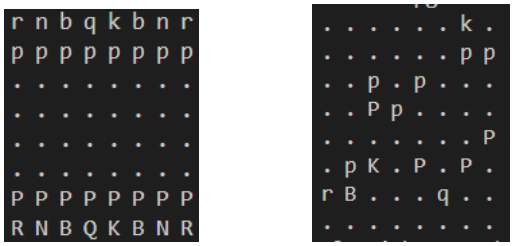

Fig. 3.1 Illustrations of Board output

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.

IV. Implementation

A. *IDS Implementation Program*

IDS is implemented as a class with a method `findSolution` to execute IDS algorithm as shown in Fig. 4.1. It is instantiated with the initial board as its attribute. The method `findSolution` will call `DLS` method with initial depth 1, and will be repeated until it reaches checkmate state and increments the depth with 2.

The `DLS` method is a function that performs recursive `DFS` search with a specific maximum depth. It iterates through each legal move of a board and then calls the `DLS` method. The program will stop if no solution is found at that depth or if a checkmate position is encountered.

B. *BFS Implementation Program*

Similar to IDS, BFS is implemented as a class with a method called `findSolution` to execute the BFS algorithm as depicted in Fig. 4.2. The class is instantiated with the initial board as its attribute. The `findSolution` method will call the `runBFS` method to generate a solution.

The `runBFS` method executes the BFS algorithm using a deque data structure. Each legal move of a board is enqueued into the deque as a child node representing a board that has undergone one move. The program will stop when a board in a checkmate position is found, and it will call the `makePath` method to reconstruct each move taken to reach that board.

```python
from chess import Board, Move, engine
from algo import Algo, Result, sort_moves_by_valuation
import time

class IDS(Algo):
    def __init__(self, board : Board):
        super().__init__(board)
        self.result = Result()
        self.evaluator = engine.SimpleEngine.popen_uci(r"E:/stockfish/stockfish-windows-x86-64-avx2.exe")

    def getName(self):
        return "IDS"

    def findSolution(self):
        depth = 1
        moves = []
        player = self.board.turn

        start_time = time.time()
        while True:
            mate_moves = self.DLS(self.board, player, moves, depth)
            if mate_moves != None:
                end_time = time.time()
                self.result.depth = depth
                self.result.moves = mate_moves
                self.result.time = (end_time-start_time)*1000
                self.evaluator.close()
                return
            depth += 2

    def DLS(self, board : Board, player, moves : list[Move], depth):
        if (board.is_checkmate()):
            return moves

        if depth <= 0 or board.is_stalemate():
            return None

        child_moves = list(board.legal_moves)
        if (player != board.turn):
            child_move = sort_moves_by_valuation(board, child_moves, self.evaluator)[0]
            moves.append(child_move)
            board.push(child_move)
            next = self.DLS(board, player, moves, depth-1)
            if next != None:
                return next
            moves.remove(child_move)
            board.pop()
        else :
            for move in child_moves:
                moves.append(move)
                board.push(move)
                next = self.DLS(board, player, moves, depth-1)
                if next != None:
                    return next
                moves.remove(move)
                board.pop()

        return None
```

Fig. 4.1 Implemetation of IDS class

```python
from chess import Board, Move, engine, pgn
from algo import Algo, Result, sort_moves_by_valuation
from collections import deque
import os
import time

class BFS(Algo):
    def __init__(self, board: Board):
        super().__init__(board)
        self.result = Result()
        self.evaluator = engine.SimpleEngine.popen_uci(r"E:/stockfish/stockfish-windows-x86-64-avx2.exe")

    def getName(self):
        return "BFS"

    def findSolution(self):
        player = self.board.turn

        start_time = time.time()
        solution = self.runBFS(self.board, player)
        end_time = time.time()
        self.result.moves = solution
        self.result.time = (end_time-start_time)*1000
        self.evaluator.close()
        return

    def runBFS(self, board : Board, player):
        queue = deque([board])
        while len(queue) != 0:
            new_board = queue.popleft()
            if new_board.is_checkmate():
                return self.makeMoves(board, new_board)

            if new_board.is_stalemate():
                continue

            child_moves = list(new_board.legal_moves)
            if (player != new_board.turn):
                child_move = sort_moves_by_valuation(new_board, child_moves, self.evaluator)[0]
                c_board = new_board.copy()
                c_board.push(child_move)
                queue.append(c_board)
            else :
                for move in child_moves:
                    c_board = new_board.copy()
                    c_board.push(move)
                    queue.append(c_board)

    def makeMoves(self, start_board : Board, end_board : Board):
        moves = []
        while start_board != end_board:
            move = end_board.pop()
            moves.insert(0, move)
        return moves
```

Fig. 4.2 Implementation of BFS class

C. *Other Functions and Classes Implemented*

To support the IDS and BFS algorithms, some function and class is implemented as depicted in Fig. 4.3. The `Algo` class is an abstract base class that inherits from the `IDS` and `BFS` classes. The `Result` class is a data structure for storing the output of the program. The `sort_moves_by_evaluation` function is a function to sort a set of legal moves owned by a board state. This function is used to determine the moves taken by the opponent in the puzzle because we want the opponent to always make the best move to avoid making foolish moves.

```python
from chess import Board, Move, engine, pgn
from abc import ABC, abstractmethod
import os

class Algo(ABC):
    def __init__(self, board: Board):
        self.board = board

    @abstractmethod
    def findSolution(self):
        pass

class Result:
    moves = []
    time = 0.0
    depth = 0

def evaluate_move(board : Board, move : Move, evaluator : engine.SimpleEngine):
    board.push(move)
    info = evaluator.analyse(board, engine.Limit(depth=5))
    board.pop()
    return info["score"].pov(not board.turn).score(mate_score=10000)

def sort_moves_by_valuation(board : Board, moves : list[Move], evaluator : engine.SimpleEngine):
    return sorted(moves, key=lambda move: evaluate_move(board, move, evaluator), reverse=False)
```

Fig. 4.3 Supporting Functions and Classes

V. TESTING

In this section, the results of tests conducted on 4 samples will be presented. Subsequently, the results will be discussed based on their time and space complexity.

A. *Test Results*

We will test with 4 different depths. As mate-in-one is too easy to solve, we will start from mate-in-two.

1. Mate-in-two

   We use a game I recently played online in chess.com.

Fig. 5.1 alexnomad vs sunegoo, 1 June 2024
(Source: chess.com)

In this game, my opponent hung a clear mate-in-two just to grab my bishop at f6. I played as black

We save the PGN file as mate-in-2.pgn

```
Insert .pgn file: mate-in-2.pgn
r . b . . k . r
. . p . . . p p
p b N p . B . .
. p . P . q . .
. . . . . . . .
. Q P . . . . .
P P . . . P P P
R N . . R . K .

Algorithms to choose:
1. Iterative Deepening Search
2. Breadth First Search

Select an algorithm (1 or 2): 1
Found the solution in 1867.4466609954834 ms.
It was mate in 2!
The moves:
f5f2
g1h1
f2e1
```

Fig. 5.2 Mate-in-two test using IDS

```
Insert .pgn file: mate-in-2.pgn
r . b . . k . r
. . p . . . p p
p b N p . B . .
. p . P . q . .
. . . . . . . .
. Q P . . . . .
P P . . . P P P
R N . . R . K .

Algorithms to choose:
1. Iterative Deepening Search
2. Breadth First Search

Select an algorithm (1 or 2): 2
Found the solution in 59528.37538719177 ms.
It was mate in 2!
The moves:
f5f2
g1h1
f2e1
```

Fig. 5.3 Mate-in-two test using BFS

As we see, both algorithm can solve the mate-in-two problem

2. Mate-in-three

   We use a daily puzzle by chess.com 11 June 2024 edition.

Fig. 5.4 Daily Puzzle, 11 June 2024
(Source: chess.com)

This is an interesting puzzle, as white already has battery of two rooks and a queen ready to attack opponent's king. It is a mate-in-three situation, and it is white to move

We save the PGN file as mate-in-3.pgn

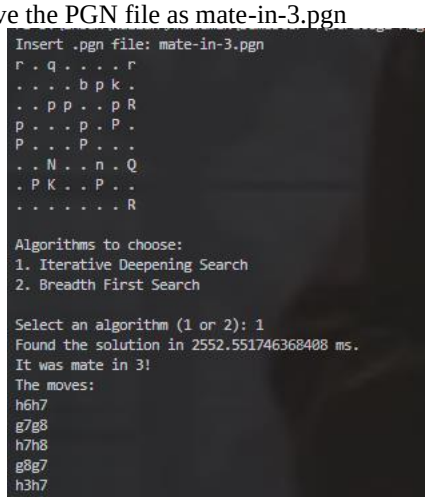

Fig. 5.5 Mate-in-three test using IDS

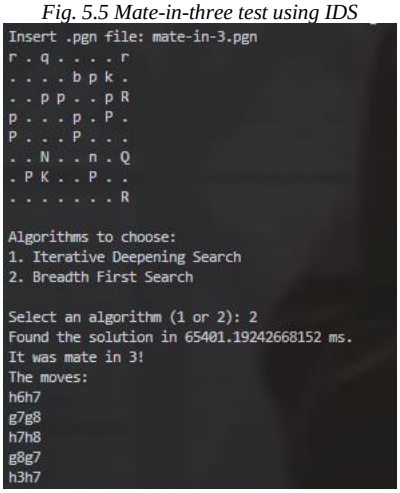

Fig. 5.6 Mate-in-three test using BFS

Both algorithm successfully solved the problem

3. Mate-in-four

We use another game I have played online recently.

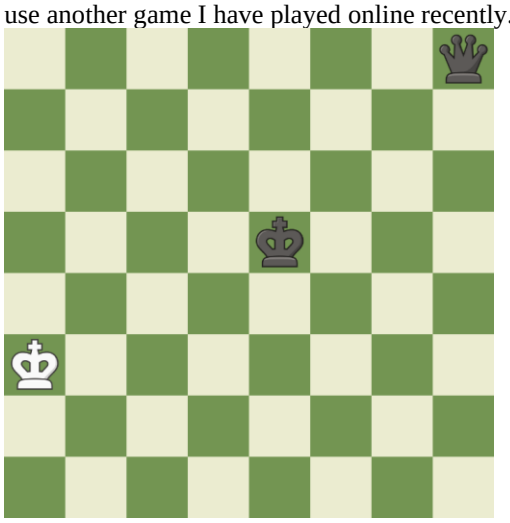

Fig. 5.7. Daily Puzzle, 8 June 2024
(Source: chess.com)

In this game, the evaluation bar indicated a mate-in-four. However, due to time pressure with only 17 seconds left, I ended up playing 8 moves before finally checkmating my opponent.

We save the PGN file as mate-in-4.pgn

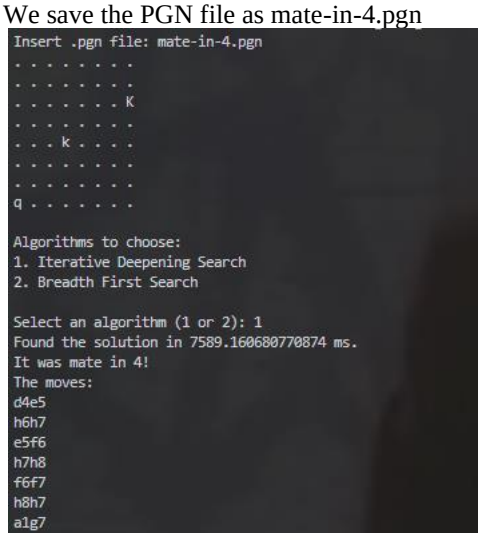

Fig. 5.8 Mate-in-four test using IDS

Unfortunately, my laptop crashed when I try to run the BFS

B. Result Analysis

Below is the table of test results that have been conducted:

Test Case	Depth	Time (ms)	
		IDS	BFS
mate-in-two	3	1867,44	59528,37
mate-in-three	5	2552,55	65401,19
mate-in-four	7	7589,16	infinite

Table 5.1 Test results

It is evident that the execution time of BFS is significantly longer than that of IDS. This aligns with the theory that the time complexity for IDS is O(bd), while the time complexity for BFS is $O(b^d)$. As for space complexity, I believe it doesn't need further explanation since it was clear that my laptop almost froze while attempting to run the mate-in-four test.

VI. CONCLUSION

The IDS and BFS algorithms are two of the graph search algorithms, and the problem of finding a solution for mate-in-n chess puzzle is one implementation of these algorithms. Despite some shortcomings in the implementation, the author has successfully demonstrated that both algorithms are relevant in solving this problem.

Moving forward, the author suggests:

- Optimizing BFS so that the program can run concurrently.
- Trying to run the BFS algorithm on more capable devices.

VII. APPENDIX

Program used in this paper can be seen here.

The author also made a video explanation of this paper. It can be seen here.

ACKNOWLEDGMENT

All Praise and gratitude to Allah Subhanahu wa Ta'ala, for by His mercy, the paper titled "Analysis of Iterative Deepening Search and Breadth-First Search Algorithm Complexity Implemented in Minimax for Solving Mate-in-N Chess Puzzle" has been successfully completed. Also, gratitude is extended to the lecturer for IF2211 Algorithm Strategies course, Dr. Nur Ulfa Maulidevi, S.T, M.Sc., for the guidance and motivation provided throughout her tenure in teaching the students.

REFERENCES

[1] Munir, Rinaldi. 2024. *Breadth/Depth First Search (BFS/DFS).* [online] Available at: <https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/BFS-DFS-2021-Bag1-2024.pdf> [Accessed 12 June 2024 08:56]

[2] Levitin, Anany. *Introduction to the Design & Analysis of Algorithms 3rd Edition*. Addison-Wesley, 2003.

[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, 12 Juni 2024

Ibrahim Ihsan Rasyid (13522018)