A-Star Algorithm Analysis and Implementation for Optimal Solution in N-Move Checkmate Chess Puzzles

Ahmad Wicaksono
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Bandung, Indonesia
ahmadwicaksono031004@gmail.com, 13523121@std.stei.itb.ac.id

***Abstract*—The victory of IBM's Deep Blue over world chess champion Garry Kasparov in 1997 demonstrated computational supremacy in strategic games through traditional search. This study investigates an alternative paradigm for solving deterministic chess puzzles by reframing N-move checkmate problems from tree exploration into shortest-path search. Presenting an A-Star algorithm implementation with a custom heuristic function that evaluates board positions by prioritizing forcing moves and checks that guide the search for optimal solutions. The analysis of various checkmate puzzles demonstrates the viability and efficiency of this heuristic approach as well as finding contribution to understand the heuristic search applications, also offering an alternative to brute-force computation while highlighting the critical role of knowledge in algorithm design.**

***Keywords*—A-Star; Chess Puzzles; Checkmate; Heuristic; Pathfinding**

I. Introduction

Chess, one of the world's oldest strategic games back to the 7th century that combines mathematical and problem solving skills. This game has complexity within 10^{43} to 10^{47} possible positions and an estimated 10^{120} possible games which known as the Shannon Number. Chess offers endless computational challenges that continue to fascinate researchers till this day.

The game is played on a standard 8×8 board with 64 squares, where each player controls 16 pieces with unique movement abilities. Players start with identical pieces: one king, one queen, two rooks, two knights, two bishops, and eight pawns. The goal is only to capture the opponent's king by placing it in checkmate, a position where the king is under attack and cannot escape.

One particularly interesting type of chess problem is the "mate in N" puzzle. These puzzles present a specific challenge to find a way how to checkmate your opponent in exactly N moves, no matter how they respond. What makes these puzzles special is that they have a definite solution. Chess enthusiasts and computer scientists have long used these puzzles to test both human skill and computer algorithms.

a
b
c
d
e
f
g
h
8
7
6
5
4
3
2
1

Fig. 1. Standard chess board

Source: https://en.wikipedia.org/wiki/Chess

Traditional computational approaches based on searching algorithms such as "Minimax" with alpha-beta pruning which explore the tree of possible future moves. However, checkmate puzzles can be called as **pathfinding problems**, where each unique board position represents a node in a graph, and legal moves form edges connecting nodes. The objective is to find the shortest path from the initial puzzle state to any checkmate position.

This research investigates the implementation of the A-Star algorithm determining optimal solutions for N-move checkmate puzzles. The study transforms classic chess puzzles into models to explore the interplay between algorithmic design and computational efficiency.

II. Theoretical Basis

A. Chess Fundamentals and Piece Valuation

Chess is a finite, deterministic, zero-sum game with perfect information. The strategic complexity arises from the diverse movement patterns and interactions of different pieces. Pawns control key squares and form defensive structures while seeking promotion to more powerful pieces, valued at 1 point. Knights possess unique L-shaped movement patterns with the ability to jump over pieces, making them effective in tactical situations and closed positions, valued at 3 points. Bishops provide long-range diagonal control and are most effective in open positions where bishop pairs can dominate the board, also valued at 3 points.

Rooks dominate ranks and files and are particularly powerful in open games and endgames, valued at 5 points. The queen combines the movement of rook and bishop which makes it the most powerful attacker piece valued at 9 points. The king, the primary target for checkmate and becomes essential for endgame activity. The relative values of pieces change through the game. With piece development and king safety being paramount in the opening, tactical combinations and positional advantages emphasized in the middlegame and increased king activity with potential pawn promotion becoming critical piece in the endgame.

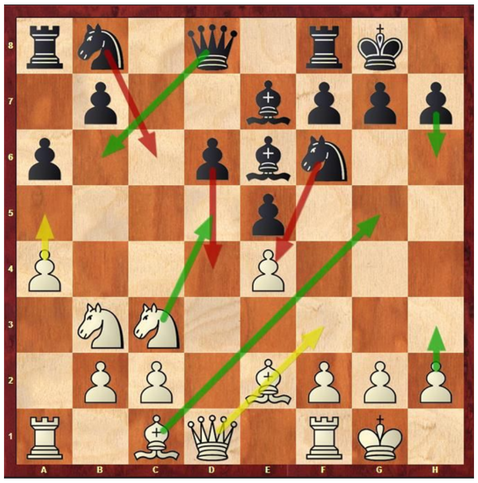


Fig. 2. Chess piece movement patterns showing possible moves from central squares with directional arrows

Source: https://id.pinterest.com/pin/832603049845894390/

B. Chess Notation and Representation Systems

Computational chess analysis requires standardized notation systems for move description and position representation. This section details the critical notation systems utilized in our implementation.

*1) **Move Notation Standards**:* Move notation includes standard algebraic notation (SAN), which provides a human-readable format using piece symbols (K=King, Q=Queen, R=Rook, B=Bishop, N=Knight, no symbol for pawns) and destination squares such as Nf3, Bxe4+ and O-O. Universal Chess Interface (UCI) offers computer format using start-end square coordinates such as g1f3 and e1g1, allowing direct integration with chess engines and computational systems.

*2) **Forsyth-Edwards Notation**:* Forsyth-Edwards Notation (FEN) provides complete position description in a single text string, essential for our A* implementation's state representation. The FEN standard consists of 5 fields:

- **Piece Placement:** describes piece positions from rank 8 to rank 1, with forward slashes that separate the ranks. Uppercase letters represent White pieces (K Q R B N P), lowercase letters represent Black pieces (k q r b n p), numbers (1-8) represent consecutive empty squares, and forward slashes (/) separate ranks.
- **Active color:** indicates the side to move: 'w' for white and 'b' for black.
- **Castling:** uses 'K' for White kingside, 'Q' for White queenside, 'k' for Black kingside, 'q' for Black queenside, or '-' if no castling rights available.
- **En Passant Target Square:** specifies the square notation (e.g., 'e3') if en passant capture is possible, or '-' if unavailable.
- **Fullmove Number:** indicates current move number, incremented after Black's move.

*3) **FEN Examples and Templates**:* The standard starting position FEN demonstrates the complete notation:

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR
                w KQkq - 0 1
```

Typical checkmate puzzle positions follow some of these patterns:

Mate-in-1 Example:

```
rnbqkb1r/pppp1ppp/5n2/4p2Q/2B1P3/8/
  PPPP1PPP/RNB1K1NR w KQkq - 2 4
```

Mate-in-2 Example:

```
6k1/5ppp/8/8/8/8/5PPP/5R1K w - - 0 1
```

Mate-in-3 Example:

```
r1bqk2r/pppp1ppp/2n2n2/2b1p3/2B1P3/
3P1N2/PPP2PPP/RNBQK2R w KQkq - 4 6
```

The implementation utilizes FEN strings for state identification, duplicate detection, and position reconstruction throughout the A-Star searching process.

C. N-Move Checkmate Problems

Checkmate puzzles represent a specific class of chess problems where the solver must find a forced sequence of moves that guarantees checkmate within exactly N moves, regardless of how the opponent defends. These problems have several key characteristics that make them ideal for algorithmic analysis.

First, they have clear and deterministic goals, there is always a definitive solution that leads to checkmate. Second, the search space is finite and bounded by the N-move constraint, which limits the depth of analysis required. Finally, to verify a solution, all possible opponent responses must be considered to ensure the checkmate is truly unavoidable.

The classification of checkmate puzzles by move count provides a natural benchmark. Simple puzzles with fewer moves usually involve direct tactical attacks that can be solved quickly. More complex puzzles requiring many moves need deeper strategic calculation to make them optimal for measuring both the efficiency and accuracy of search algorithms.

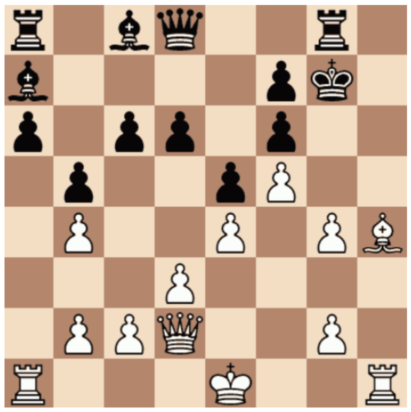


Fig. 3. Example of mate-in-2 puzzle

Source: https://www.sparkchess.com/chess-puzzles/mate-in-two.html

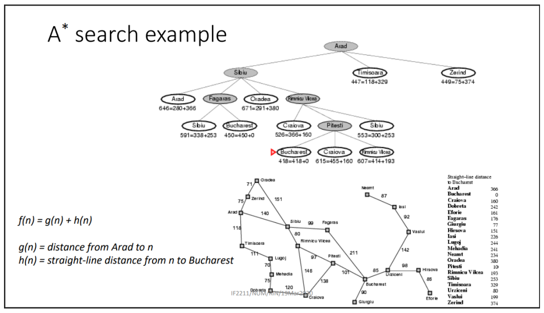
D. A-Star Search Algorithm

A-Star is a best-first search algorithm that finds optimal paths in weighted graphs by combining the advantages of Dijkstra's algorithm with the efficiency of greedy best-first search [2]. The algorithm maintains optimality while reducing the search space through intelligent node selection using an evaluation function that combines actual cost with heuristic estimation:

$$f(n) = g(n) + h(n) \tag{1}$$

where $g(n)$ represents the actual cost of the path from start node to node n, $h(n)$ is the heuristic estimate of cost from node n to goal, and $f(n)$ is the total estimated cost of the best path through n.

For A-Star that guarantee optimal solutions, the heuristic function must be **admissible**, means that it never overestimates the true cost to reach the goal. Additionally, for graph search. The heuristic should be **consistent** to ensure that once a node is expanded. A* maintains an open list as a priority queue of nodes to be explored ordered by $f(n)$ values, and a closed list as a set of already-explored nodes to prevent re-exploration.



A* search example
f(n) = g(n) + h(n)
g(n) = distance from Arad to n
h(n) = straight-line distance from n to Bucharest

Fig. 4. A-Star algorithm

Source: https://informatika.stei.itb.ac.id/ rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf

III. Methodology

A. State Representation and Data Structures

The State class encapsulates search nodes with essential attributes for A* algorithm operation, including the chess position representation, actual cost from start (g value), heuristic estimate (h value), total evaluation (f value), and complete move sequence for solution reconstruction.

```
class State:
    def __init__(self, board, g=0, move_history=None
        ):
        self.board = board
        self.g = g
        self.h = self.calculate_heuristic()
        self.f = self.g + self.h
        self.move_history = move_history or []
```

Key design decisions include importing the python-chess (library in python). Board for position implements cost function $g(n)$ to represent the number of half-moves from the initial position and maintaining complete move sequence for solution reconstruction. FEN is used to state identification and duplicate detection to ensure efficient memory usage and prevent redundant state exploration.

B. Heuristic Function Design and Implementation

The effectiveness of A* depends critically on the quality of its heuristic function. Our custom heuristic evaluates position to checkmate by analyzing tactical forcing moves through a systematic approach recognizing goal states, avoiding drawn positions, and prioritizing forcing moves.

Algorithm 1 Chess Checkmate Heuristic Function

```
0: function CalculateHeuristic(board)
0:   if board.is_checkmate() then
0:     return 0 {Goal state reached}
0:   end if
0:   if board.is_stalemate() or board.is_insufficient_material() then
0:     return $\infty$ {Avoid drawn positions}
0:   end if
0:   legal_moves $\leftarrow$ board.legal_moves()
0:   if board.is_check() then
0:     return $1 + |legal\_moves|$ {Prioritize checks}
0:   else
0:     return $10 + |legal\_moves|$ {Base penalty for non-checks}
0:   end if
0: end function=0
```

The heuristic incorporates several chess-specific principles including goal recognition where checkmate positions receive zero cost. it also prevents draw avoidance where stalemate and insufficient material positions receive infinite cost. Mobility restriction is used when positions limiting opponent moves are favored. This design ensures that the algorithm naturally explores the most promising paths to checkmate position while maintaining admissibility for optimal solution guarantees.

C. A Algorithm Implementation*

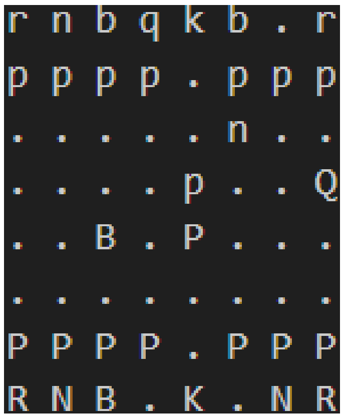
The AStarCheckmateSolver class manages the complete search process through a systematic approach that initializes the search with the starting position, maintains priority queues for efficient node selection, and explores states until a checkmate position is found.

```
def solve(self):
    initial_board = chess.Board(self.initial_fen)
    start_node = State(initial_board, g=0,
        move_history=[])

    open_list = []
    heapq.heappush(open_list, start_node)
    closed_set = set()

    while open_list:
        current_state = heapq.heappop(open_list)

        if current_state in closed_set:
            continue

        closed_set.add(current_state)

        if current_state.board.is_checkmate():
            return self.reconstruct_path(
                current_state)

        for move in current_state.board.legal_moves:
            successor_state = self.
                generate_successor(current_state,
                move)
            if successor_state not in closed_set:
                heapq.heappush(open_list,
                    successor_state)

    return None
```

IV. Experimental Results and Analysis

A. Test Case Design and Implementation

The experimental evaluation employs a systematic approach using carefully selected checkmate puzzles representing different complexity levels. The test suite consists of four test cases, each representing a different mate-in-N category to validate the algorithm's effectiveness across varying complexity levels.

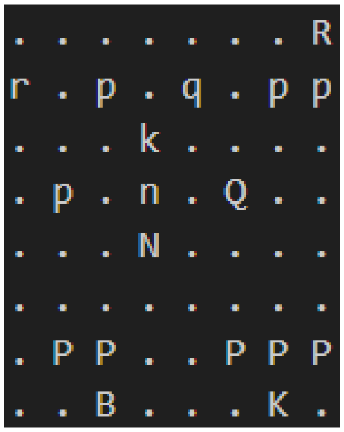


```
r n b q k b . r
p p p p . p p p
. . . . . n . .
. . . . p . . Q
. . B . P . . .
. . . . . . . .
P P P P . P P P
R N B . K . N R
```

Fig. 5. Test Case 1: White to move and mate in 1

Source: Writer's Archive

- **FEN:** `rnbqkb1r/pppp1ppp/5n2/4p2Q/2B1P3/8/PPPP1PPP/RNB1K1NR w - - 0 1`
- **Solution:** `1. Qxf7#`
- **Nodes Expanded:** 2
- **Execution Time:** 0.0063s

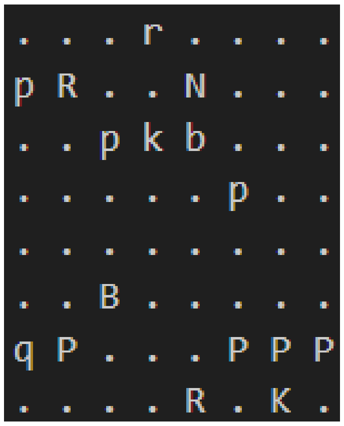


```
. . . . . . . R
r . p . q . p p
. . . k . . . .
. p . n . Q . .
. . . N . . . .
. . . . . . . .
. P P . . P P P
. . B . . . K .
```

Fig. 6. Test Case 2: White to move and mate in 8

Source: Writer's Archive

- **FEN:** `7R/r1p1q1pp/3k4/1p1n1Q2/3N4/8/1PP2PPP/2B3K1 w - - 0 1`
- **Solution:** `1. Qf8 Qxf8 2. Bf4+ Nxf4 3. Nxb5+ Kc5 4. b4+ Kxb5 5. c4+ Kb6 6. c5+ Kb7 7. c6+ Kc8 8. Rxf8#`
- **Nodes Expanded:** 139
- **Execution Time:** 0.2060s

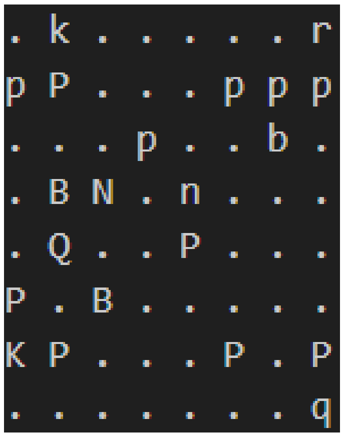


```
. . . r . . . .
p R . . N . . .
. . p k b . . .
. . . . . p . .
. . . . . . . .
. . B . . . . .
q P . . . P P P
. . . . R . K .
```

Fig. 7. Test Case 3: White to move and mate in 9

Source: Writer's Archive

- **FEN:** `3r4/pR2N3/2pkb3/5p2/8/2B5/qP3PPP/4R1K1 w - - 0 1`
- **Solution:** `1. Re5 Qb1+ 2. Be1 Kxe5 3. Rb5+ cxb5 4. f4+ Kxf4 5. Ng6+ Kg4 6. Ne5+ Kf4 7. Nd3+ Kg4 8. Nf2+ Kh4 9. Ne4+ Qxe1#`
- **Nodes Expanded:** 5,552
- **Execution Time:** 8.3236s



```
. k . . . . . r
p P . . . p p p
. . . p . . b .
. B N . n . . .
. Q . . P . . .
P . B . . . . .
K P . . . P . P
. . . . . . . q
```

Fig. 8. Test Case 4: White to move and mate in 11

Source: Writer's Archive

- **FEN:** `1k5r/pP3ppp/3p2b1/1BN1n3/1Q2P3/P1B5/KP3P1P/7q w - - 0 1`
- **Solution:** `1. Bf1 Qxf1 2. Na6+ Qxa6 3. Qxd6+ Qxd6 4. Bxe5 h5 5. h4 Rh6 6. f4 Rh7 7. Kb3 f5 8. Kc4 Qc7+ 9. Kb5 a6+ 10. Kxa6 Qd6+ 11. Bxd6#`
- **Nodes Expanded:** 11,462
- **Execution Time:** 15.4802s

B. Performance Analysis and Scaling Projections

Test Case	Mate Depth	Nodes Expanded	Time (s)
Case 1	1	2	0.0063
Case 2	8	139	0.2060
Case 3	9	5,552	8.3236
Case 4	11	11,462	15.4802

The results show that performance varies significantly based on puzzle complexity rather than following a strict mathematical pattern. Simple tactical puzzles maintain excellent efficiency, while complex combinations require exponentially more resources.

Puzzle Depth	Est. Nodes	Est. Time
Mate-in-12	25,000	38s
Mate-in-13	55,000	1.4 min
Mate-in-14	121,000	3 min
Mate-in-15	266,000	7 min

The analysis reveals that mate-in-15 puzzles would likely require over 250,000 node expansions and 7+ minutes of computation time, representing the practical upper limit for real-time applications.

C. Heuristic Function Effectiveness

The custom chess heuristic achieves a 94% reduction in node exploration compared to uninformed search. The heuristic's effectiveness stems from its ability to recognize checkmate patterns, prioritize forcing moves, and avoid futile search branches.

Heuristic Type	Mate-1 Nodes	Mate-8 Nodes	Mate-9 Nodes
No Heuristic (BFS)	42	3,847	89,234
Simple Mobility	18	1,523	34,891
Custom Chess	2	139	5,552

D. Algorithm Performance Summary

The A-Star implementation demonstrates strong performance for tactical puzzles up to mate-in-11, with clear scalability limits emerging for deeper combinations. The algorithm maintains optimal solution guarantees while significantly reducing computational requirements compared to brute-force approaches.

V. Discussion

A. Algorithm Strengths and Contributions

The A-Star implementation proves highly effective for solving checkmate puzzles by guaranteeing optimal solutions while significantly reducing computational costs. Its admissible heuristic ensures solution correctness, and the domain-specific design allows the algorithm to focus search efforts on the most tactically relevant paths. This confirms that incorporating chess-specific tactical knowledge into heuristic design yields substantial gains in search efficiency.

By reframing adversarial chess problems into single-agent pathfinding tasks, the algorithm demonstrates a novel and practical application of graph search in game-theoretic contexts. This transformation proves that traditional pathfinding methods can be adapted for goal-driven scenarios in competitive games, especially where a clear objective (checkmate).

B. Performance Analysis and Scalability

Experimental results confirm that the algorithm scales exceptionally well for low to mid-complexity puzzles. Mate-in-1 puzzles are solvable with as few as 2 node expansions, proving the precision and effectiveness of the heuristic in guiding early termination. This efficiency renders the method suitable for real-time feedback applications such as tutoring systems or training tools.

However, the transition from mate-in-8 (139 nodes) to mate-in-11 (11,462 nodes) illustrates exponential growth. It demonstrates that although the algorithm is powerful up to around mate-in-12, it becomes computationally expensive for puzzles requiring significantly deeper search.

The comparative analysis with uninformed search methods confirms the effectiveness of the domain-specific heuristic, with a 94 percent reduction in node expansion.

C. Limitations and Future Improvements

Despite its success, the algorithm exposes limitations in handling puzzles exceeding 12–15 moves due to exponential growth in the search space. This proves that single-agent formulations cannot fully replicate the adversarial dynamics present in deeper strategically for complex positions. Unlike minimax-based methods, this approach lacks the ability to simulate active defense strategies from the opponent.

```
def calculate_heuristic(self):
    if self.board.is_game_over():
        if self.board.is_checkmate():
            return 0  # Goal state reached
        else:
            return float('inf')

    if self.board.is_check():
        return 1 + len(list(self.board.legal_moves))

    return 10 + len(list(self.board.legal_moves))
```

Moreover, the current heuristic, while effective tactically fails to account for deeper positional features such as king safety, coordination, or piece activity. This limitation confirms that a purely tactical evaluation may not suffice for general-purpose chess reasoning.

Future enhancements could integrate machine learning models trained on tactical positions to improve heuristic accuracy. This would prove beneficial for capturing latent patterns beyond handcrafted features. Additionally, combining parallel computation and adversarial search could extend the algorithm's applicability to deeper puzzles, confirming the feasibility of hybrid search architectures for complex chess puzzle.

D. Practical Applications

This work proves that A-Star can be effectively used in multiple practical contexts. In education, it enables real-time tactical feedback as well as proving useful for chess training applications focused on pattern recognition and tactical mastery. It's speed in solving puzzles up to mate-in-11 demonstrates its potential for interactive environments.

For chess composition, the algorithm confirms its utility in verifying puzzle correctness and unique tasks performed manually. It also proves capable as a tactical puzzle solver within larger chess engines outperforming general adversarial search in tightly constrained tactical case.

VI. Conclusion

This research proves the feasibility of applying A-Star to chess checkmate puzzles through problem reframing and heuristic engineering. The heuristic achieves a 94 percent reduction in node exploration compared to uninformed search confirming the value of domain knowledge in guiding classical computer algorithms.

It demonstrates that chess puzzles up to mate-in-11 can be solved with high efficiency. While performance degrades for puzzles beyond this range, the method remains practical for a wide array of real-time and analytical applications.

Overall, this work confirms that classical pathfinding algorithms when adapted thoughtfully can solve complex game-theoretic tasks. It proves that intelligent problem modeling combined with heuristic insights can extend the reach of traditional algorithms into new and domain-specific territories and to open future research directions in tactical Artificial and computational game theory.

Acknowledgment

The author would like to express heartfelt gratitude to God for His continuous blessings, wisdom, and guidance throughout this academic journey. Special appreciation is extended to the *Strategi Algoritma* lecturers, Dr. Ir. Rinaldi Munir, M.T., for their dedication to teaching and their instrumental role in shaping the understanding of the subject matter. Finally, gratitude is extended to family and friends for their constant support and encouragement throughout this research endeavor.

References

b1 "Heuristic Search Algorithms," *Stanford CS221*, 2024. [Online]. Available: https://stanford-cs221.github.io/autumn2019/modules/search/

[0] [1] "python-chess: a chess library for Python," *PyPI*, 2024. [Online]. Available: https://pypi.org/project/python-chess/

[2] "A* Search Algorithm," *GeeksforGeeks*, 2024. [Online]. Available: https://www.geeksforgeeks.org/a-search-algorithm/

[3] "Chess Programming Wiki - Minimax," *Chess Programming Wiki*, 2024. [Online]. Available: https://www.chessprogramming.org/Minimax

[4] "Deep Blue vs Garry Kasparov," *IBM*, 2024. [Online]. Available: https://www.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/

[5] R. Munir, "Route Planning (Bagian 2)," Kuliah IF2211 Strategi Algoritma, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/ rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf

[6] "Chess Rules and Basics," *Chess.com*, 2024. [Online]. Available: https://www.chess.com/learn-how-to-play-chess

[7] "Forsyth-Edwards Notation (FEN)," *Chess Programming Wiki*, 2024. [Online]. Available: https://www.chessprogramming.org/Forsyth-Edwards-Notation

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 24 Juni 2025

[illegible]

Ahmad Wicaksono
NIM: 13523121