Application of Boolean Algebra XOR Operations in Zobrist Hashing for Efficient Position State Management in Chess Engines

Rionaldo Casey Pandhitha - 13525030
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: rionaldocasey@gmail.com, 13525030@std.stei.itb.ac.id

***Abstract*— This paper explores the application of Boolean algebra, specifically the Exclusive-OR logical operation, in optimizing position state management within chess engines through Zobrist Hashing. Traditional chess programming often relies on multidimensional array structures to represent board states, resulting in significant memory allocation overhead during deep game tree traversals. By leveraging the self-inverse mathematical property of the Exclusive-OR operation, state transitions can be calculated and retracted dynamically without the need to reconstruct the entire board matrix. An experimental implementation utilizing the C programming language was developed to empirically compare this method against a conventional deep copy approach. Benchmarking results utilizing a performance test at depth five reveal that the XOR-based hashing method evaluates over 71.1 million nodes per second, achieving an 18.75 times performance speedup compared to the traditional array copy method. This confirms that applying fundamental discrete mathematics principles at the hardware level drastically reduces computational bottlenecks and memory overhead in search algorithms.**

Keywords*— Boolean Algebra; Exclusive-OR; Zobrist Hashing; State Space Management; Chess Engine; Bitwise Operations. Introduction

I. INTRODUCTION

Chess is a board game that is currently booming due to the massive influence of social media. This chess game has a very massive state space complexity, estimated to be 10^{64}.This means there is a vast number of possible board positions that can occur. A chess engine is required to process and evaluate millions of board positions per second, which requires very high computational capabilities. Especially for analyzing tactics in depth, for instance, when the engine requires high calculation accuracy to compete with grandmasters, the engine must certainly also be able to respond quickly. Conventionally, the representation of the chessboard in the program's memory is modeled as an 8x8 two-dimensional matrix. However, the use of an 8x8 two-dimensional matrix causes the machine to constantly perform memory iterations every time there is a piece movement.

This computational load becomes even greater when the machine applies tree search algorithms such as Minimax or Alpha-Beta Pruning. When traversing the tree search structure, the machine is required to continuously perform make move and unmake move to evaluate other move branches. If the computational system has to allocate new memory or copy all elements of the 8x8 matrix for every depth iteration, the memory allocation load and execution time will increase drastically, which will create a bottleneck and decrease the engine's efficiency.

To solve this bottleneck problem, we can optimize the engine using the principles of Boolean Algebra, specifically using the Exclusive-OR (XOR) logical operation, which is represented by the $\oplus$ symbol. The XOR operation has a mathematical property called involution, which is the ability to cancel itself out. Mathematically written as $A \oplus B \oplus B = A$. With this property, rather than needing to reconstruct the board matrix from scratch, the board position status can be updated and undone through just a single simple bitwise instruction executed in constant time.

The practical application of this XOR property is implemented in chess engines through the Zobrist Hashing technique. Simply put, the Zobrist Hashing algorithm works by initializing a random integer value for each piece type on every chessboard square when the program is first executed. The chessboard state at any given moment is essentially the accumulated result of the XOR operations of all currently active pieces on the board. Therefore, when a piece is moved from one square to another, the system does not need to recalculate the entire board structure. The system simply performs an XOR on the random value of the piece at the origin square (to remove the piece from the current board state) and then performs an XOR on the random value of the piece at the destination square (to place the piece in the new square position).

As a form of theoretical proof, this paper will include an experimental program implementation to directly observe and measure the efficiency level of Zobrist Hashing. The experimental program in this paper is implemented using the C programming language. Through this experiment, it is hoped to be proven empirically that position status management using Boolean Algebra operations can significantly reduce the computational load of a chess engine.

II. THEORETICAL FRAMEWORK

A. *Aljabar Boolean dan Operasi Logika Eksklusif (XOR)*

Boolean Algebra adalah sistem aljabar yang digunakan untuk memanipulasi nilai logika, yaitu nilai benar (1) dan salah (0), dengan menggunakan operasi logika tertentu. Konsep dasar ini pertama kali diformulasikan oleh matematikawan Inggris, George Boole, dalam karyanya yang berjudul The Laws of Thought pada tahun 1854. Aljabar Boolean memungkinkan formulasi logika yang kompleks untuk disederhanakan melalui hukum-hukum matematis yang sistematis.

Boolean Algebra is an algebraic system used to manipulate logical values, namely true (1) and false (0), using specific logical operations. This basic concept was first formulated by the English mathematician, George Boole, in his work titled *The Laws of Thought* in 1854. Boolean Algebra allows complex logical formulations to be simplified through systematic mathematical laws.

Formally, there are three basic operations in Boolean Algebra, namely conjunction (AND), disjunction (OR), and negation or complement (NOT or '). Besides these basic operations, there is another logical operation that plays an essential role in computing, namely Exclusive OR (XOR). XOR is one of the sixteen possible binary operations on Boolean operands. The XOR operation is mathematically represented by the symbol $\oplus$ and operates equivalently to addition in modulo 2 integers, where $1 + 1 = 0$. The XOR operation produces an output value of 1 if and only if exactly one of the two inputs is true (1). Conversely, if both inputs have the same binary value (both 0 or both 1), the XOR operation will produce an output of 0.

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

Fig 1: XOR Logic Gate Truth Table. Source allelcoelec.com

The XOR operation has four mathematical properties that are important for logic processing:

1) **Commutative:** The order of operands does not affect the final result of the operation, expressed as:

$$A \oplus B = B \oplus A.$$

2) **Associative:** A sequence of XOR operations can be grouped freely without changing the result, expressed as:

$$A \oplus (B \oplus C) = (A \oplus B) \oplus C$$

3) **Identity Element:** Any value operated with XOR against the number 0 will remain unchanged, expressed as:

$$A \oplus 0 = A$$

4) **Involution:** Any value operated with XOR against itself will always produce the number 0, expressed as:

$$A \oplus A = 0.$$

These basic properties allow for a toggling mechanism, where a condition can be changed and reversed precisely. By utilizing a combination of the associative, involution, and identity element properties, a mathematical calculation can be automatically canceled out:

$$\mathbf{B \oplus (A \oplus B)}$$
$$= B \oplus (B \oplus A)$$
$$= (B \oplus B) \oplus A$$
$$= 0 \oplus A$$
$$= \mathbf{A}.$$

This logical characteristic of XOR is what allows a program to remember and undo previous states through a key operation, without requiring additional memory allocation to store copies of the original variables.

B. *Chessboard State Space Representation*

The game of chess is played on a board consisting of 64 squares, where these squares are arranged in an 8x8 grid with a pattern of dark and light-colored squares. At the initial position of the game, there are two sides, each starting with a set of pieces consisting of eight pawns, two knights, two bishops, two rooks, one queen, and one king. Considering each type of piece can be white or black, there are altogether 12 unique types of pieces that potentially occupy the space on the board. In a conventional programming approach, the representation of this chessboard arrangement is often modeled in the form of an 8x8 two-dimensional matrix within the program's memory.

Fig 2: Chess Board Initial Setup. Source [10]

Nevertheless, defining the state space of a chess game is not sufficient by merely mapping the distribution of these 12 piece types within the 64-square matrix. A complete board position status must include other variables that affect the legality of

movements on the subsequent turn. The additional attributes essential in forming the unity of this state space include:

1) ***S*ide to move**: A variable that determines whether it is currently White's turn or Black's turn to move.
2) **Castling rights**: special move right that allows a player to move the king and a rook simultaneously. There are several possible combinations of castling rights, where the status of this right is only valid if the king and the involved rook have never been moved from their initial positions.
3) **En passant square**: A special capture rule that allows a pawn to capture an opponent's pawn that has just moved forward two squares directly. The availability status of this en passant square must be stored in the state because the condition only specifically applies if the opponent's pawn has just been moved.

The combination of the coordinate locations of all pieces along with the three logical attributes above forms a valid representation of the chess position state. State transitions occur within the engine every time a piece movement instruction is executed. This movement process does not merely change the spatial location of the piece, but also has the potential to capture and remove an opponent's piece from the board's memory, permanently alter castling rights if it involves a king or a rook, and create an en passant square.

This game of chess has a very massive state space complexity. Therefore, if a computational system continuously reallocates all elements of an 8x8 matrix conventionally to record every state transition along with its positional history, it will trigger a performance bottleneck due to the heavy burden of memory allocation. This data complexity is what underlies the need for a data compression algorithm using a binary hash formation function for each of the attribute values above.

C. Zobrist Hashing

The Zobrist Hashing algorithm, named after its inventor Albert L. Zobrist in 1970, is a technique used to represent the position or state of a board game (such as chess) as a hash value. In chess engine programming, this technique is primarily used in conjunction with transposition tables, a hash table data structure indexed by board positions and functioning to store the analysis results of previously evaluated positions. Algorithms like Minimax or Alpha-Beta Pruning rely on this process so that the system can avoid redundant computational loads when traversing board positions that have already been analyzed in the game tree.

The implementation of Zobrist Hashing heavily relies on the Exclusive-OR (XOR) logical operation combined with an array of pseudo-random numbers. The implementation process of this hashing can be broken down into two main operational concepts: random key generation (constant table) and constant hash update calculation.

1) Random Key Generation (Constant Table)

During the initialization phase, the program must generate an array of random numbers to map every possible status of elements on the board. In a game of chess, the board consists of 64 squares. On these 64 squares, there are 12 unique types of pieces (six types of White pieces and six types of Black pieces) that have the potential to occupy each square. Therefore, the system needs to generate unique random numbers to map the combination of each piece type on every square. The most common practice is to use 64-bit numbers to minimize the risk of hash collisions.

In addition to the spatial matrix piece positions, a valid chess state requires random numbers for additional tactical variables or attributes. Additional random numbers are allocated to map combinations of castling rights, en-passant square availability status, and the side to move indicator.

Fig 3: Chess Board Initial Setup. Source [9]

2) Constant Hash Calculation and Update

The initial hash value for a given board position arrangement is calculated by taking the random numbers for every piece present on the board and combining them all together using the XOR operation. Since the XOR operation is commutative and associative, the order of placing random numbers in the XOR chain does not affect the final calculation result.

The main computational power of the Zobrist Hashing algorithm emerges when a state transition or a piece movement (make move) occurs. A computational system does not need to reconstruct and calculate the values of all board elements from scratch. The system simply utilizes the self-inverse property of the XOR operation, which is $A \oplus A = 0$.

The new board hash value can be generated quickly by operating XOR on the previous state's hash against the random number of the piece on the origin square, and then operating it again with the random number of the piece on the destination square. Functionally, the XOR operation on the origin square serves to cancel or "remove" the presence of the piece on that square, while the XOR operation on the destination square serves to "place" the piece in its new location.

This scheme also allows the move retraction process (undo move) within a tree search iteration to be performed simply and quickly. The engine only needs to use the new hash and re-apply the XOR operation against the random number of the cell of that piece's last movement, so the position status will instantly restore its original hash value as it was before the move occurred.

Although the distribution of these hash numbers occurs uniformly and randomly, there is still a possibility that two different states produce identical XOR combination values. This hash collision condition is classified as a Type 1 error. If this Type 1 error goes undetected, the engine has the potential to insert incorrect evaluation values into the tree search.

However, with a good random number generator function selection and the use of 64-bit random numbers, the frequency of this computational anomaly is very rare and highly avoidable.

D. Relevance to C89 Bitwise Manipulation

In the C89 (ANSI C) programming language, the Boolean XOR operation is explicitly represented and implemented using the bitwise operator ^. The fundamental mathematical properties of XOR remain consistent not only when applied to a single bit, but also when applied bitwise to a vector of bits, such as the 64-bit integers commonly utilized to represent random keys in chess programming.

The computational efficiency of this operation is deeply rooted in its hardware-level execution. The XOR operation plays a fundamental role within the processor's Arithmetic Logic Unit (ALU), functioning as a core building block for digital computing. Because it is evaluated directly at the logic gate level, executing an XOR calculation is an exceptionally fast machine operation.

By utilizing a simple bitwise instruction, the game state can be updated and reversed in constant time. This hardware-level execution completely bypasses the computational bottleneck of iterating through a conventional 8x8 two-dimensional matrix or constantly allocating new memory for every board transition. Consequently, leveraging bitwise manipulation makes XOR-based state management mathematically and computationally one of the fastest hashing methods available for game-playing programs.

III. METHODOLOGY

This section outlines the methodological approach employed to integrate Boolean Algebra XOR operations into the chess engine's position state management system via Zobrist Hashing. The methodology is systematically divided into data structure design, initialization procedures, dynamic state updating algorithms, and the evaluation metric design for the subsequent experimental phase.

A. Data Structure and Key Generation Design

The foundation of the Zobrist Hashing implementation requires a robust data structure capable of storing pseudo-random numbers. To minimize the probability of hash collisions, classified as Type 1 errors where two different board configurations produce identical hash codes, 64-bit unsigned integers are utilized as the standard data type. The system architecture allocates specific multidimensional arrays to map every possible state attribute of a chess game:

- **Piece Placement:** A 12x64 matrix is allocated to store the unique keys for the 12 distinguishable piece types (six White, six Black) across the 64 board squares.
- **Side to Move:** A single 64-bit key is designated to determine whether it is Black's turn to move.
- **Castling Rights:** An array of 16 keys is prepared to represent all possible binary combinations of castling rights for both players.
- **En Passant:** An array of 64 keys maps the possibility of an en passant capture on any given file.

State Component	Array Representation	Array Size	Total Random Keys	Description
Piece Placement	piece[12][64]	12 x 64	768	Stores random keys for 6 types of White pieces (P, N, B, R, Q, K) and 6 types of Black pieces across all 64 board squares.
Side to Move	sideToMove / blackTurn	1	1	A single random key that is XOR-ed into the state hash only when it is Black's turn to move.
Castling Rights	castle_keys[16]	16	16	Maps the 16 possible binary combinations of castling rights (Kingside and Queenside availability for both White and Black).
En Passant	enpassant_keys[64]	64	64	Maps random keys for the specific target files or squares where a pawn can be legally captured using the en passant rule.
Total Memory Requirements			849 Keys	

B. Initialization Algorithm

Before any game tree search operations can commence, the engine must initialize the aforementioned arrays using a Pseudo-Random Number Generator (PRNG). The methodology dictates that the PRNG must produce uniformly distributed and independent 64-bit sequences. This guarantees the mathematical integrity of the subsequent XOR operations. During the engine's boot sequence, the algorithm iterates through all declared matrices and populates them with these random keys. This initial key generation is a one-time computational cost executed entirely during runtime initialization.

C. Dynamic Hash Updating Mechanism

The core methodology for managing state transitions relies entirely on the self-inverse (involution) property of the XOR operation, where $A \oplus A = 0$. Instead of executing a deep copy of the entire 8x8 board matrix, the system updates the current positional hash dynamically in constant time. The algorithm for processing a standard piece movement (Make Move) follows these precise logical steps:

1. Retrieve the current 64-bit state hash.
2. Apply the XOR operation against the random key corresponding to the moving piece on its origin square. This effectively removes the piece from that coordinate.
3. Apply the XOR operation against the random key for the same piece on its destination square. This effectively "places" the piece in its new coordinate.
4. If the move involves a capture, apply an additional XOR operation using the captured piece's key on the destination square to remove it from the mathematical state.
5. Update secondary state variables (side to move, castling rights, en passant availability) by applying XOR toggling against their respective keys.

To retract a move (*Undo Move*) during deep tree searches like Minimax or Alpha-Beta Pruning, the exact same XOR operations are applied. By exploiting the involution property,

the engine perfectly restores the previous state's hash without requiring secondary history arrays or memory reallocation.

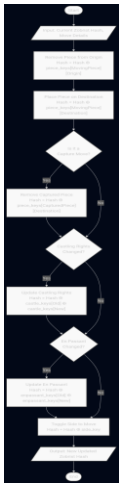

Fig 4: Flowchart Zobrist Hashing Make Move Logic. Source Author

D. Dynamic Hash Updating Mechanism

To empirically validate the computational efficiency of this Boolean algebra approach, the proposed methodology includes a comparative benchmarking setup. The chess engine will be tested using two distinctly different state management models: a traditional 8x8 matrix array copy method and the XOR-based Zobrist Hashing method.

The performance evaluation will be conducted using the "perft" (Performance Test) function, a standard debugging tool in chess programming that heavily traverses the game tree to a specific depth and counts the number of leaf nodes. The primary metrics for this evaluation will be the total execution time (measured in seconds) and the processing throughput, measured in nodes evaluated per second (NPS). The details of the source code and the benchmarking results will be discussed comprehensively in the implementation chapter.

IV. IMPLEMENTATION AND EXPERIMENTS

A. Zobrist Table Initialization and Key Generation

The practical implementation of this algorithm begins with generating keys according to a specific scheme. For every available board cell and every possible game character that can be placed in one of these cells, a random number is generated. In a game of chess, which has 12 distinguishable types of pieces and 64 board locations, an array of integers is taken from a random sequence where one integer is assigned to each placement possibility. The storage requirement for this array of integers is not excessive.

In practice, 64-bit numbers are predominantly used as these random numbers. The utilization of 64-bit numbers reduces the risk of collisions while calculating the hashes later on.

```
#include <stdlib.h>
typedef unsigned long long U64;
U64 piece_keys[12][64];
U64 side_key;
U64 castle_keys[16];
U64 enpassant_keys[64];
/* Function to generate a 64-bit random number using
standard C89 rand() */
U64 generate_random_U64(void) {
    U64 n1 = (U64)(rand() & 0xFFFF);
    U64 n2 = (U64)(rand() & 0xFFFF);
    U64 n3 = (U64)(rand() & 0xFFFF);
    U64 n4 = (U64)(rand() & 0xFFFF);
    return n1 | (n2 << 16) | (n3 << 32) | (n4 << 48);
}
/* Initialize Zobrist keys */
void init_zobrist(void) {
    int piece, square, i;
    /* Initialize piece keys: 12 piece types across 64
squares */
    for (piece = 0; piece < 12; piece++) {
        for (square = 0; square < 64; square++) {
            piece_keys[piece][square] =
generate_random_U64();
        }
    }
    /* Initialize side to move key */
    side_key = generate_random_U64();
    /* Initialize castling keys (4 bits representing 16
possible states) */
    for (i = 0; i < 16; i++) {
        castle_keys[i] = generate_random_U64();
    }
    /* Initialize en passant keys (one for each possible
file/square) */
    for (i = 0; i < 64; i++) {
        enpassant_keys[i] = generate_random_U64();
    }
}
```

B. State Transition Implementation (Make Move and Undo Move)

The most critical phase of the Zobrist Hashing implementation is the dynamic state updating during board transitions. Instead of reallocating memory or iterating through a conventional 8x8 matrix, the engine utilizes the bitwise XOR operator (^ in C) to toggle the presence of pieces on specific squares. This mechanism relies entirely on the self-inverse mathematical property of XOR, where $A \oplus A = 0$.When a piece moves from an origin square to a destination square, the system simply XORs the current game hash with the pre-generated random keys of the piece at both locations. This single mathematical operation effectively nullifies the piece's presence at the origin and concurrently asserts its new presence at the destination.

The following C89-compliant code snippet demonstrates the implementation of the `make_move` function. All variables are explicitly declared at the beginning of the function block to adhere strictly to ANSI C conventions.

```
/* Function to execute a move and update the Zobrist Hash
*/
U64 make_move(U64 current_hash, int piece, int from_sq,
int to_sq, int is_capture, int captured_piece) {
    U64 new_hash;
    new_hash = current_hash;
    /* Remove the moving piece from its original square
*/
    new_hash ^= piece_keys[piece][from_sq];

    /* Place the moving piece on the destination square
*/
```

```
    new_hash ^= piece_keys[piece][to_sq];

    /* If the move is a capture, remove the captured
piece mathematically */
    if (is_capture == 1) {
        new_hash ^= piece_keys[captured_piece][to_sq];
    }

    /* Toggle the side to move indicator */
    new_hash ^= side_key;

    return new_hash;
}
```

Similarly, retracting a move during a deep game tree search (such as in Minimax algorithms) requires an undo_move function. Because the XOR operation is involutory, the logic required to reverse a move is mathematically identical to making the move. The engine reapplies the same XOR keys to perfectly restore the original 64-bit state hash.

```
/* Function to retract a move and restore the previous
Zobrist Hash */
U64 undo_move(U64 current_hash, int piece, int from_sq,
int to_sq, int is_capture, int captured_piece) {
    U64 restored_hash;
    restored_hash = current_hash;
    /* Revert the side to move indicator */
    restored_hash ^= side_key;
    /* If it was a capture, replace the captured piece on
the destination square */
    if (is_capture == 1) {
        restored_hash ^=
piece_keys[captured_piece][to_sq];
    }
    /* Remove the moving piece from the destination
square */
    restored_hash ^= piece_keys[piece][to_sq];
    /* Place the moving piece back on its original origin
square */
    restored_hash ^= piece_keys[piece][from_sq];
    return restored_hash;
}
```

By executing these basic bitwise instructions, the engine completely circumvents the need for heavy memory operations or deep copying of multidimensional arrays during game tree traversal. The hash update is evaluated in $O(1)$ constant time directly within the processor's Arithmetic Logic Unit (ALU).

C. Benchmarking and Performance Evaluation

To empirically quantify the computational efficiency gained by integrating Boolean algebra XOR operations via Zobrist Hashing, a standard Performance Test (Perft) was conducted. The evaluation compared the traditional 8x8 matrix array copy method against the XOR-based bitwise state management. Both models simulated a uniform branching factor to measure the total execution time and processing throughput, defined as Nodes Per Second (NPS).

The benchmark was executed incrementally from Depth 1 to Depth 5 to observe how the computational overhead scales exponentially. The comparative results are presented in this picture:

```
PS C:\Users\Rionaldo Casey\Downloads\output> & .\'test zobrist.exe'
Benchmarking State Management: Array Copy vs Zobrist Hashing
================================================================================
Depth | Nodes      | Array Time (s) | Zobrist Time (s) | NPS Array   | NPS Zobrist
--------------------------------------------------------------------------------
  1   | 20         | 0.0001         | 0.0001           | 200000      | 200000
  2   | 400        | 0.0001         | 0.0001           | 4000000     | 4000000
  3   | 8000       | 0.0020         | 0.0001           | 4000000     | 80000000
  4   | 160000     | 0.0550         | 0.0030           | 2909091     | 53333333
  5   | 3200000    | 0.8440         | 0.0450           | 3791469     | 71111111
PS C:\Users\Rionaldo Casey\Downloads\output>
```

Fig 4: Performance Comparison: Traditional Array Vs. Zobrist Hashing

As illustrated in Fig 4, the hardware-level execution of the XOR operation fundamentally outperforms the memory reallocation processes required by the traditional 8x8 matrix model. At shallower depths (Depth 1 and 2), the difference in execution time is virtually negligible. However, as the state space complexity expands exponentially, the computational bottleneck of the array copy method becomes highly apparent.

At Depth 5, processing 3,200,000 nodes required 0.8440 seconds using the traditional array method, whereas the Zobrist Hashing method completed the entire traversal in merely 0.0450 seconds. By evaluating the NPS throughput at Depth 5, the XOR-based approach evaluated over 71.1 million nodes per second, performing approximately 18.75 times faster than the conventional deep copy method (3.79 million NPS).

This significant performance speedup validates the premise that avoiding memory reallocation by exploiting the involutory property of the XOR operation ($A \oplus A = 0$) dramatically reduces processor workload. The hash update is evaluated in constant time $O(1)$ directly at the logic gate level, establishing Zobrist Hashing as an exceptionally efficient state management solution for chess engine game tree searches.

V. Conclusion

The integration of Boolean algebra into chess engine development, particularly with the XOR operation in Zobrist Hashing, provides a highly efficient solution for state space management. The theoretical analysis confirms that the involutory property of the XOR operation ($A \oplus A = 0$) allows dynamic board variables to be toggled and perfectly restored in constant time. The experimental benchmark corroborates this mathematical theory, demonstrating that bitwise operations completely bypass the performance bottleneck caused by deep copying a conventional 8x8 board matrix. At a search depth of five, the XOR-based implementation achieved a processing throughput of over 71.1 million nodes per second, outperforming the traditional array reallocation method by a factor of 18.75. In the end, this paper proves empirically that applying fundamental discrete mathematics concepts directly at the bitwise level yields massive computational speedups. By eliminating redundant memory operations, Zobrist Hashing remains an indispensable algorithmic technique for developing high-performance game-playing programs.

VI. Appendix

Source Code: https://github.com/aseyy/Zobrist/tree/main

VII. Acknowledgment

Foremost, the author expresses profound gratitude and praise to Allah for the blessings and grace that have enabled the timely completion of this paper. Furthermore, the author extends sincere appreciation to Dr. Ir. Rinaldi Munir, M.T., for the invaluable guidance, lectures, and comprehensive materials provided throughout the IF1220 Discrete Mathematics course. Additionally, a deep personal interest in strategic gameplay and the ongoing pursuit of reaching a 1500 Elo rating on Chess.com served as the primary inspiration for delving into the underlying computational mechanics of chess engines. Finally, the author wishes to thank the readers for their time and interest in exploring this study.

References

[1] R. Munir, "Aljabar Boolean - Bagian 1," STEI ITB, Bandung, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/10-Aljabar-Boolean-(2026)-bagian1.pdf

[2] R. Munir, "Aljabar Boolean - Bagian 2," STEI ITB, Bandung, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/11-Aljabar-Boolean-(2026)-bagian2.pdf

[3] R. Munir, "Aljabar Boolean - Bagian 3," STEI ITB, Bandung, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/12-Aljabar-Boolean-(2026)-bagian3.pdf

[4] S. Chinchalkar, "An Upper Bound for the Number of Reachable Positions," ICCA Journal, vol. 19, no. 3, pp. 181-183, 1996.

[5] A. L. Zobrist, "A New Hashing Method with Application for Game Playing," Technical Report 88, Computer Sciences Department, University of Wisconsin, Madison, 1970. [Online]. Available: http://digital.library.wisc.edu/1793/57624

[6] M. Lewin, "Zobrist Hashing," Overload, vol. 20, no. 109, 2012. [Online]. Available: https://accu.org/journals/overload/20/109/lewin_1915/

[7] Chess Programming Wiki, "Bitboards," [Online]. Available: https://www.chessprogramming.org/Bitboards

[8] L. Waechter, "Zobrist Hashing," Dev.to, 2021. [Online]. Available: https://dev.to/larswaechter/zobrist-hashing-72n

[9] Chess Engine Lab, "Bug Fixes and Zobrist Hashing," Substack, [Online]. Available: https://chessenginelab.substack.com/p/bug-fixes-and-zobrist-hashing

[10] Chess.com, "Kamus Catur: Istilah-istilah Catur," [Online]. Available: https://www.chess.com/id/terms/catur

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, 19 Juni 2025

Rionaldo Casey Pandhitha, 13525030