Efficient Chess Board Representation

Shashank S*, Raghava G. Dhanya†
Email: *contact@knhash.in, †public@ragv.in

Abstract—This paper delves into the intersection of chess and computer science, specifically focusing on the efficient representation of chess game states. We propose two methods: the Static Method and the Dynamic Method, each offering unique advantages in terms of space efficiency and computational complexity.

The Static Method aims to represent the game state using a fixed-length encoding, allocating 192 bits to capture the positions of all pieces on the board. This method introduces a protocol for ordering and encoding piece positions, ensuring efficient storage and retrieval. However, it faces challenges in representing pieces no longer in play.

In contrast, the Dynamic Method adapts to the evolving game state by dynamically adjusting the encoding length based on the number of pieces in play. By incorporating Alive Bits for each piece kind, this method achieves greater flexibility and space efficiency. Additionally, it includes provisions for encoding additional game state information such as castling rights and en passant squares.

Our findings demonstrate that the Dynamic Method offers superior space efficiency compared to traditional Forsyth-Edwards Notation (FEN), particularly as the game progresses and pieces are captured. However, it comes with increased complexity in encoding and decoding processes.

In conclusion, this study provides insights into optimizing the representation of chess game states, offering potential applications in chess engines, game databases, and artificial intelligence research. The proposed methods offer a balance between space efficiency and computational overhead, paving the way for further advancements in the field.

Keywords—chess, optimisation, encoding, bit manipulation

I. Introduction

CONSIDER a typical chess board. It is a 8x8 grid of 64 black and white squares. At the beginning of a game there are 32 pieces in play, 16 per side, of the following kind:

- Pawn - 8, Rook - 2, Bishop - 2, Knight - 2, Queen - 1, King - 1

To save the game state we need to store the location of every piece in play at any given point in time. A set of game states from start to finish constitutes a match. There is some meta information that is also required, the state of castling, en-passant captures, next player's turn, half counts and full counts. With all this information you can reliably store and consistently recreate any game that has been played.

A. Background and Motivation

Chess, a popular two-player strategy board game, has been a subject of interest for computer scientists and mathematicians for decades [1], [2]. Efficient representation of the game state is of concern for various applications, including chess engines, game databases, and artificial intelligence research [3], [4]. This paper aims to explore two methods for space-efficient representation of chess game states.

B. Objectives and Scope

The primary objective of this research is to propose and analyze two methods for representing the game state of a chess board: the Static Method and the Dynamic Method. We will evaluate their respective protocols, advantages, and disadvantages, and compare their space efficiency with the existing FEN notation [5].

C. Organisation of the paper

This paper is organized as follows: Section 2 provides a literature review on chess game state representations and existing space-efficient techniques. Sections 3 and 4 describe the Static and Dynamic Methods, respectively. Section 5 discusses the code implementation of the Dynamic Method and presents test cases. Section 6, the epilogue, provides the initial seed idea and true motivation for this endeavor.

II. Literature Review

A. Chess Game State Representations

Various methods have been proposed in the literature for representing chess game states. These methods can be broadly classified into two categories: human-readable representations and computer-friendly representations.

1) Human-Readable Representations: Human-readable representations focus on conveying game state information in a format that can be easily understood by humans. Some common human-readable representations include:

1) **Algebraic Notation:** A widely used notation that represents each move in the game by specifying the initial and final squares of the moving piece. It also includes additional information for special moves, such as castling and en passant captures [6].
2) **Descriptive Notation:** An older notation system that describes each move in terms of the piece being moved and its destination square, using a combination of letters and numbers. This notation is less compact than algebraic notation and has largely been replaced by it in modern chess literature.

2) Computer-Friendly Representations: Computer-friendly representations are designed to facilitate efficient processing and storage of game states by computers. Some common computer-friendly representations include:

1) **Bitboards:** A data structure that represents the position of each piece type on the board using a 64-bit integer. Bitboards enable fast move generation and evaluation through bitwise operations, making them well-suited for high-performance chess engines.

2) **Forsyth-Edwards Notation (FEN):** A compact, human-readable format that encodes the position of each piece on the board using a single character, along with additional information such as castling rights, en passant target square, half-move clock, and full-move number. FEN is widely used for exchanging chess positions between software applications and for storing game states in databases.

B. Existing Space-Efficient Techniques

Several techniques have been proposed to improve the space efficiency of chess game state representations. These techniques can be grouped into two main categories: data compression algorithms and specialized data structures [10].

1) Data Compression Algorithms: Data compression algorithms aim to reduce the size of game state representations by exploiting patterns and redundancies in the data. Some common data compression algorithms applied to chess game states include:

1) **Huffman Coding:** A lossless compression algorithm that assigns shorter binary codes to more frequent symbols, based on their probabilities of occurrence. Huffman coding has been used to compress chess game databases, resulting in significant space savings [7].
2) **Run-Length Encoding:** A simple lossless compression algorithm that replaces consecutive occurrences of the same symbol with a single instance of the symbol followed by a count of its repetitions. Run-length encoding can be applied to compress FEN strings, particularly for positions with large areas of empty squares.

2) Specialized Data Structures: Specialized data structures aim to store and manipulate chess game states more efficiently than general-purpose data structures. Some examples of specialized data structures for chess game states include:

1) **Tries:** A tree-like data structure that stores game states based on their common prefixes. Tries can be used to store and retrieve chess positions in a space-efficient manner, particularly for large game databases [8].
2) **Succinct Data Structures:** A family of data structures that store and manipulate information using a number of bits close to the information-theoretic lower bound. Succinct data structures have been proposed for various chess-related tasks, such as move generation and position evaluation, offering potential space and time efficiency improvements over traditional data structures [9].

In this paper, we propose and analyze two novel methods for space-efficient representation of chess game states, the Static Method and the Dynamic Method, and compare their performance with existing techniques, such as FEN notation.

III. The Static Method - 192 bits

64 (2^6) squares means we need 6 bits to store a position a position on the board. With 32 pieces in play at the start, we would need 192 $(32 * 6)$ bits to store the position of all the pieces. But that is still not enough information, which position belongs to which piece?

A. The Static Protocol

We encode the kind of piece using the following protocol:

- Bits are chunked in sets of 6, each 6 bit chunk representing the position of a piece
- Chunks are ordered by kind, white pieces of a kind first.
  - Pawn, Rook, Bishop, Knight, Queen, King
  - So, for instance, the bits in the indexes $[113, 119]$ tell the position of the second White Bishop

We therefore have the position of every piece, stored in 192 bits.

There is one more problem to solve, how to represent a piece that is no longer in play? Every piece apart from Pawn and certain Bishop can reach every square on the board, so we cannot set the value of a dead piece to any valid value within $[000000 - 111111]$.

We use the following two insights from the game:

1) The Kings are always in play
2) No two pieces can ever occupy the same square at a given instance

The final protocol thus becomes:

- Bits are chunked in sets of 6, each 6 bit chunk representing the position of a piece
- Chunks are ordered by kind, white pieces of a kind first.
  - Pawn, Rook, Bishop, Knight, Queen, King
  - So, for instance, the bits in the indexes $[113, 119]$ tell the position of the second White Bishop
- Dead pieces have their position set to the position of their corresponding King

IV. The Dynamic Method - [42-228] bits

Maintaining location information for every dead piece as we keep progressing through the game is wasteful. We could possibly omit this if we are able to save the count of pieces in play, per kind, per side.

Consider the case for White Pawns. Lets add 4 more bits before the first 8 chunks, and call these the Alive Bits (AB). A value of 0 (0000) would indicate no Pawns left alive, a value of 8 (1000) would indicate all Pawns alive.

The AB will indicate how many of the following chunks to read. So if there are 5 pawns left alive, we would need 4 bits (0101) plus 30 bits $(6 * 5)$ in total. So the bit-length to represent White Pawns would range between 4 $(4 + (0 * 6))$ bits and 52 $(4 + (0 * 6))$ bits.

Extending this to all the pieces, we have the dynamic protocol:

A. The Dynamic Protocol

We start from the Static Protocol and have the following additional points:

- Every set of chunks of a particular kind, on every side, is preceded by a set of Alive Bits. The number of Alive Bits, per side per kind, are as follows:
  - Pawn: 4 bits
  - Rook, Bishop, Knight: 2 bits
  - Queen, King: 1 bit

The Alive Bits signify how many of the following chunks to be read. Thus, the number of bits needed to represent a particular game state can range between the following cases:

- Best case: (4 + (0*6*2)) + (2 + (0*6*2)) + (2 + (0*6*2)) + (2 + (0*6*2)) + (1 + (0*6*2)) + (1 + (1*6*2)) = 36 bits
- Worst case: (4 + (8*6))*2 + (2 + (2*6))*2 + (2 + (2*6))*2 + (2 +(2*6))*2 + (1 + (1*6))*2 + (1 + (1*6))*2 = 216 bits

Thus with the Dynamic Protocol the number of bits necessary to represent a game state matches the Static Protocol after two pieces' death and reduces going further.

B. Additional game state information

In addition to the piece locations and Alive Bits, we also need to store information about en passant, castling, and the turn indicator in our dynamic protocol. To do this, we will add some extra bits to our representation:

- En passant: 1 bit to indicate if en passant capture is possible, and if it is, 6 bits to represent the target square (total of 7 bits, 1 bit when not possible).
- Castling: 4 bits to represent the castling rights for both sides (1 bit for each: white king-side, white queen-side, black king-side, and black queen-side).
- Turn indicator: 1 bit to represent which side has the turn to move (0 for white, 1 for black).

With these additional bits, the total number of bits needed to represent a game state using the Dynamic Method now ranges between:

- Best case: 36 (from previous calculation) + 1 (no en passant) + 4 (castling) + 1 (turn indicator) = 42 bits
- Worst case: 216 (from previous calculation) + 7 (en passant) + 4 (castling) + 1 (turn indicator) = 228 bits

Thus, the Dynamic Method can represent a chess game state with a varying number of bits between 42 and 228, providing a more space-efficient representation compared to the Static Method, especially as more pieces are captured.

V. Code Implementation and Test Cases

We implemented the Dynamic Method in Python and compared its space efficiency with FEN notation using a set of test cases.

A. Python Code Implementation

The simulation code can be found on GitHub

The ChessEncoder class provides functionality for encoding and decoding chess positions between Forsyth-Edwards Notation (FEN) and a custom binary format, referred to as the Dynamic Protocol.

The `encode_dynamic` method takes a FEN string as input and converts it into a binary string according to the Dynamic Protocol. This binary string represents the chess position, including the active color, castling rights, en passant square, and the positions of all pieces on the board.

The `decode_dynamic` method takes a binary string in the Dynamic Protocol format and converts it back into a FEN string, reconstructing the original chess position.

The `encode_base85` and `decode_base85` methods provide additional functionality for encoding and decoding the binary string into a base85 string, which can be useful for more compact storage or transmission of the chess position.

Our results show that the Dynamic Method is consistently more space-efficient than FEN notation, with an average space savings of around 35%. However, there are some caveats to consider:

- The increased space efficiency comes at the cost of increased complexity in the encoding and decoding process.
- In some cases, the computational overhead of the Dynamic Method may outweigh its space efficiency benefits, particularly for applications with strict performance requirements.
- This does not represent the half moves and full moves counters

These test cases demonstrate that the Dynamic Method is indeed more space-efficient than FEN notation, but with some caveats related to complexity and computational overhead. Future work could explore further optimizations and hybrid approaches that combine the best aspects of both methods.

B. Test Case Results

```
.
----------------------------------------
Ran 1 test in 0.003s

OK
FEN: rnbqkbnr/pppppppp/8/8/8/8/
    PPPPPPPP/RNBQKBNR w KQkq - 0 1
FEN notation size: 56
Encoded bit size: 222
base85 size: 35
base85: K7t7-vkf@-!yL2IUAKW6=jb7
    By?`F$_<rJi

FEN: rnbqkbnr/pppp1ppp/8/4p3/4P3/
    5N2/PPPP1PPP/RNBQKB1R b KQkq - 1 2
FEN notation size: 62
Encoded bit size: 222
base85 size: 35
base85: 9)bxdvraw8lMHgtUAKW&=jb7
    By?`F$_<rJi

FEN: 8/4k3/8/8/
    8/8/8/4K3 w - - 0 1
FEN notation size: 29
Encoded bit size: 42
base85 size: 8
base85: 0ssI2B6I

FEN: rnbqkb1r/pppp1ppp/2n5/4P3/
    8/8/PPPP1PPP/RNBQKBNR w KQkq - 0 3
FEN notation size: 60
Encoded bit size: 216
base85 size: 34
base85: `UoVV%{ciV406w1w}Bd~m?3V
    xfF9%ce&T!

FEN: rnbqkbnr/pppp1ppp/8/4Pp2/
    8/8/PPPP1PPP/RNBQKBNR w KQkq f6 0 3
FEN notation size: 60
Encoded bit size: 228
base85 size: 37
base85: 5AA{pD6>vJNS6$9&t12H8t3RC
    ZoPmW<M@8!d;
```

VI. Conclusion

The genesis of this project was a simple question: Can we create a short URL to represent any state of a chess game? This question led us down a path of exploration into data compression, encoding schemes, and the intricacies of chess game states.

The `ChessEncoder` class, at the heart of this project, is a testament to this journey. It leverages the Forsyth-Edwards Notation (FEN) and a custom binary format, the Dynamic Protocol, to encode and decode chess positions. The class also employs `base85` encoding to further compress the data, enabling the representation of complex game states in a compact format suitable for a URL.

The Dynamic Protocol captures all necessary information about a chess position, including the active color, castling rights, en passant square, and the positions of all pieces on the board. This ensures that the full state of a chess match can be accurately represented and retrieved.

References

[1] Shannon, C. Programming a computer for playing chess. *First Presented At The National IRE Convention, March 9, 1949, And Also In Claude Elwood Shannon Collected Papers*. pp. 637-656 (1993)

[2] Campbell, M., Hoane Jr, A. & Hsu, F. Deep blue. *Artificial Intelligence*. **134**, 57-83 (2002)

[3] Stockfish, an open-source UCI chess engine. [Online]. Available: https://stockfishchess.org/

[4] Lc0, an open-source UCI chess engine based on AlphaZero. [Online]. Available: https://lczero.org/

[5] Edwards, S. Portable game notation specification and implementation guide. *Retrieved April*. **4** pp. 2011 (1994)

[6] Just, T., Burg, D. & Others United States Chess Federation's Official Rules of Chess. (Random House Incorporated,2003)

[7] Huffman, D. A method for the construction of minimum-redundancy codes. *Proceedings Of The IRE*. **40**, 1098-1101 (1952)

[8] Fredkin, E. Trie memory. *Communications Of The ACM*. **3**, 490-499 (1960)

[9] Navarro, G. Compact data structures: A practical approach. (Cambridge University Press,2016)

[10] Bell, T., Cleary, J. & Witten, I. Text compression. (Prentice-Hall, Inc.,1990)