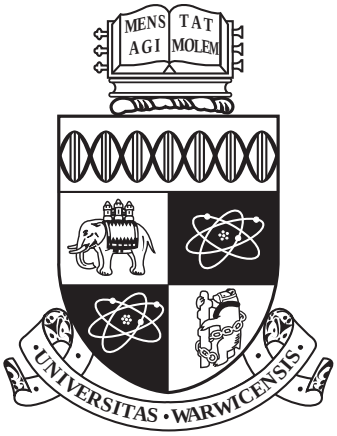

A Universal Library for Compressing Chess Games

CS310 Third Year Project

Jaden Strudwick

Supervisor: Alexander Dixon

Department of Computer Science

University of Warwick

2024

Abstract

As online chess has continued to grow in popularity over the past few years, there has been renewed interest in compression techniques for Portable Game Notation (PGN) files. These files are a standardised plain text format for recording chess games, readable by both humans and most chess software. While space-efficient, a single chess game can still take several kilobytes to be represented in an uncompressed PGN file. However, due to the well-defined ruleset of chess, multiple space-saving techniques can be applied to significantly reduce PGN file sizes. After conducting new research and using existing concepts from Information Theory, this paper presents a new universal (or programming language agnostic) library for compressing and decompressing PGN files, demonstrating a state-of-the-art compression ratio of 25.14%. For reference, Lichess.org (a popular online chess website) had a previous state-of-the-art solution that attained an average compression rate of 4.44 bits-per-move, meanwhile this library achieves 4.37 bits-per-move and is only 2.1% higher than the theoretical lower bound of 4.28 bits-per-move. The value of this library is immense, as it can help reduce storage costs for online chess websites and allow users to store larger personal repositories of chess games to study or train neural networks upon.

Keywords: Chess, Portable Game Notation, Huffman Coding, Rust, WebAssembly, Compression, Genetic Algorithms, State-of-the-art

Acknowledgements

I would like to thank my partner, family, friends, and supervisor Alexander Dixon for their utmost support throughout this endeavour. I would also like to pay special tribute to Steven J. Edwards, the creator of Portable Game Notation (PGN), who sadly passed away in 2016 [7]. His contributions, while not widely celebrated, have been invaluable to every corner of the chess community and the history of the game itself. May his legacy and selflessness continue to be a source of inspiration for all.

Contents

1 Introduction

In recent years, online chess has witnessed an unprecedented surge in popularity, a trend that has been further accelerated by global phenomena such as the COVID-19 pandemic, high-profile livestream competitions, and viral TV content such as *"The Queen's Gambit"* [4]. This has propelled online chess platforms like Chess.com to over 100 million users (seen in figure 1), as the modern iteration of chess continues to attract fresh players [4]. However, this growth has not been without challenges and Chess.com previously announced that they are struggling to scale their servers to keep up with this demand [5]. With approximately 16,000 chess moves played every second, these immense data volumes have completely saturated their databases' throughput speeds, leading to frequent downtime [5].

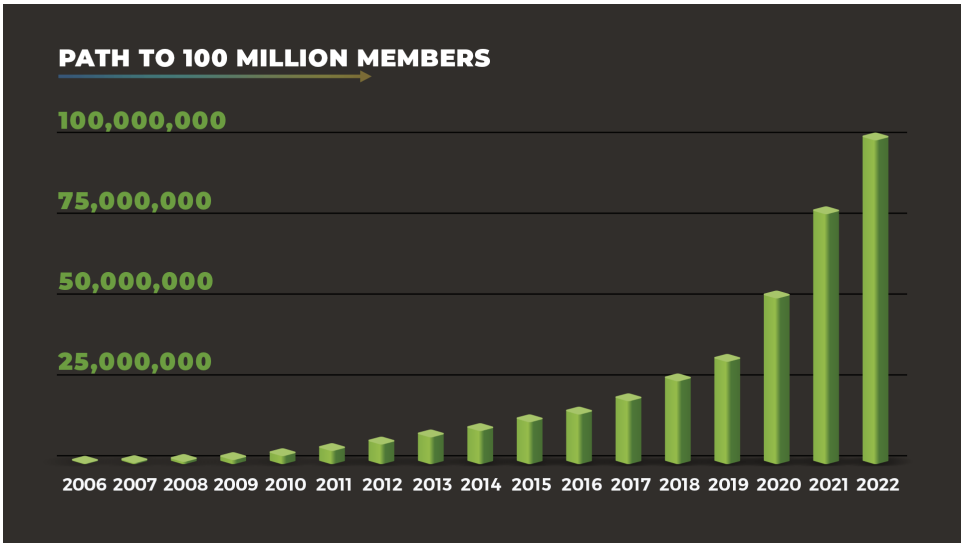

Figure 1: Growth in Chess.com membership count [4]

One of the key challenges in managing this data volume is the effective storage of Portable Game Notation or PGN files, the standard ASCII plain-text format for recording chess games [22]. Developed by Steven J. Edwards in 1993 and known for being easily parsable and human readable, PGN files have become immensely popular and are well-supported by nearly all chess engines, GUIs, and utilities. However, due to being a plain-text file format, PGN files can sometimes

be kilobytes in size when uncompressed. For example, the PGN file in figure 2 would require 726 bytes of memory.

```
[Event "F/S Return Match"]
[Site "Belgrade, Serbia JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]
[Result "1/2-1/2"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 {This opening is called the Ruy Lopez.} 3... a6
4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6 8. c3 O-O 9. h3 Nb8 10. d4 Nbd7
11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7 14. Bg5 b4 15. Nb1 h6 16. Bh4 c5 17. dxe5
Nxe4 18. Bxe7 Qxe7 19. exd6 Qf6 20. Nbd2 Nxd6 21. Nc4 Nxc4 22. Bxc4 Nb6
23. Ne5 Rae8 24. Bxf7+ Rxf7 25. Nxf7 Rxe1+ 26. Qxe1 Kxf7 27. Qe3 Qg5 28. Qxg5
hxg5 29. b3 Ke6 30. a3 Kd6 31. axb4 cxb4 32. Ra5 Nd5 33. f3 Bc8 34. Kf2 Bf5
35. Ra7 g6 36. Ra6+ Kc5 37. Ke1 Nf4 38. g3 Nxh3 39. Kd2 Kb5 40. Rd6 Kc5 41. Ra6
Nf2 42. g4 Bd3 43. Re6 1/2-1/2
```

Figure 2: PGN representation of a game between Fischer and Spassky [12]

Despite the small size of individual PGN files, the sheer volume of games being played on platforms such as Chess.com and Lichess.org leads to substantial storage requirements. For example, as of March 2023, Chess.com averages 37.67 million games per day, equating to 27.35 gigabytes daily (assuming 726 bytes per game), or around 10 terabytes annually [49]. This exponential increase in data volume necessitates more efficient chess-specific compression techniques to help these online chess platforms reduce storage costs and improve system scalability.

As such, this project aims to resolve these issues by developing an innovative solution for PGN file compression, focusing on creating a universal, language agnostic WebAssembly (WASM) library. The proposed library will employ advanced compression techniques to minimise storage needs while retaining cross-system compatibility due to using WASM [8]. Rust, a modern and memory-safe language with WASM compilation support, will be used for development, given its efficiency and suitability for high-performance applications [47].

Beyond meeting current PGN compression ratios, the project also seeks to surpass existing benchmarks and establish a new state-of-the-art so-

lution in the field. This advancement has the potential to significantly reduce storage costs for online chess platforms, improve user access to game records, and enable larger personal game libraries for players and researchers. Conclusively, this project aims to address the growing scalability needs of the online chess community and enrich the digital chess experience for enthusiasts and professionals alike.

2 Background Research

2.1 Standard Algebraic Notation

Given the niche nature of this project, we need to first gather a deeper understanding of the chess ecosystem, along with its standard notation for depicting individual moves. This is not only crucial for understanding the PGN files we wish to compress, but also for ensuring that our solution is compatible with these existing chess notation standards. Historically, there have been many different notation systems used to record chess moves, varying from culture to culture as the game spread around the world. Fortunately, with the game being at least 1,000 years old, the natural selection of ideas has resulted in these various systems being consolidated into one standardised notation referred to as Standard Algebraic Notation or SAN [13].

The earliest precursor to SAN was developed by Syrian chess master Philip Stamma in the 18th century, which slowly became adopted by German and Russian chess literature in the 19th century [13]. It was not ubiquitous, however, as existing English, Spanish, and French chess literature continued to use the older Descriptive Notation based on abbreviated natural language, rather than the coordinate system of SAN [13]. These two notations would co-exist until 1980 when FIDE (the International Chess Federation established in 1924) announced it would no longer recognise the use of Descriptive Notation [13]. Nearly overnight, SAN would become the standard method for recording and describing chess moves, with Descriptive Notation falling into obscurity.

As previously stated, SAN operates based on a system of board coordinates and piece identifiers. On an 8-by-8 chess board, the columns (also known as files) are enumerated *'a'* through *'h'*, while the rows (known as ranks) are enumerated 1 through 8 [13]. Hence, each square on the chessboard can be uniquely identified in SAN notation via its file letter followed by its rank number. For example, when observing a chessboard from the White player's perspective the bottom left square would be *'a1'*, and the top right would be *'h8'*. It is important to note that the chessboard coordinates are enumerated relative to White's

perspective, so the Black player's bottom left square would be *'h8'*, and the top right would be *'a1'*. An example of this coordinate system can be seen in figure 3.

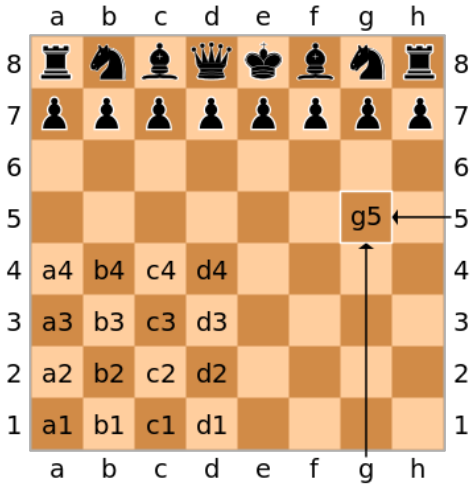

Figure 3: SAN board coordinates [13]

In addition to the coordinate system, SAN also uses five unique uppercase letters to identify each piece. In English, these identifiers are *'K'* for King, *'Q'* for Queen, *'R'* for Rook, *'B'* for Bishop, and *'N'* for Knight [13]. Pawns are the only pieces without an identifier, with the implicit assumption that any SAN move without an identifier is a pawn move [13]. While these English letters are the international standard, other languages commonly use different sets of characters for identifying the pieces. For example, the French words for King and Queen (*Roi* and *Dame*) yield *'R'* and *'D'* for their respective pieces [13]. For the sake of compatibility with other chess software, this project will use the international English identifiers outlined above.

To represent a single move in SAN, you take the identifier of the piece that moved, and append the destination coordinate to it. In most cases, this system is enough to uniquely pinpoint each move, but there are some cases where the SAN move can be ambiguous when two or more identical pieces can move to the same square [13]. In such cases, the moving piece is instead specified by its uppercase identifier followed by the departure file and/or rank to disambiguate which piece we are

referring to [13]. Two examples of this notation can be found in figures 4 and 5.

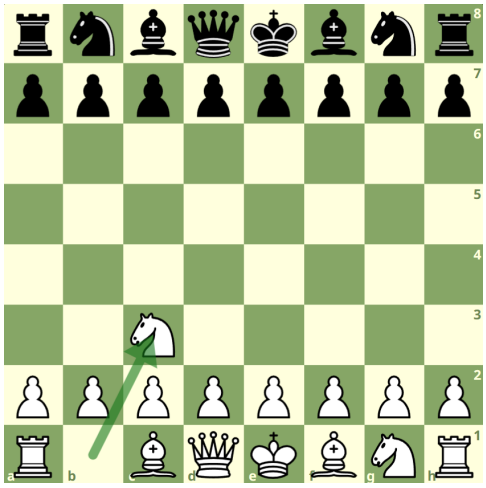

Figure 4: Knight to *c3* yields *Nc3* in SAN

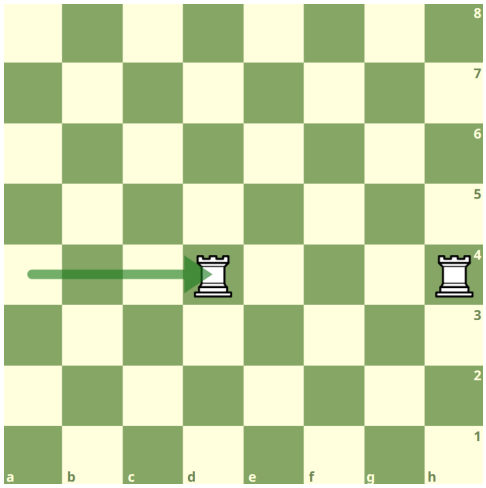

Figure 5: Ambiguous Rook to *d4* yields *Rad4* in SAN (since it was the Rook on file *'a'* that moved)

2.2 Portable Game Notation

After the standardisation of SAN in 1980, chess games were still predominately recorded on paper by the sequence of SAN moves that occurred along with other relevant information such as the players, event name, and game result. While this was not an issue at the time,

the rise of personal computing in the 1990s suddenly meant that there was a wide variety of chess software that struggled to communicate with each other without a common format for sharing games [50]. As such, it soon became clear that the birth of digital chess necessitated an additional notation for representing entire games, hopefully standardising a game's sequence of SAN moves and other metadata into one file format.

In 1993, Steven J. Edwards proposed a solution called Portable Game Notation or PGN, a standardised ASCII plain-text format for recording chess moves and metadata [22]. Crucially, the format was easy to read and write for humans, and easy to parse and generate by computer programs. After being shared on the chess hobbyist Usenet group, PGN quickly became the de-facto standard for storing and sharing digital chess games, with nearly all subsequent chess software shipping with built-in PGN parsers and generators [50]. To this day, PGN remains the unrivalled file format for storing digital chess games, used by all major online chess platforms including Chess.com and Lichess.org.

According to the specification, a PGN file contains two main sections, the tag-pair section which contains the game's metadata, and the move-text section which documents the game's moves in SAN [22]. The tag pair section consists of zero or more tag-name and tag-value pairs, with each separated on its own newline. Each tag-name should only appear once, which means in a computer science context we can consider these tag-pairs to be a set of string key-value pairs analogous to a HashMap data structure. While the specification states that any arbitrary tag-pairs can be included in a PGN file, it also recommends that a specific Seven Tag Roster, or STR, always be included at the start of the section, serving as a common nomenclature for all chess software [22]. The STR includes the Event, Site, Date, Round, White, Black, and Result tag-pairs in that order. After the STR, the PGN can include additional tag-pairs for other metadata information, such as the player's ranking or game termination status. After the last tag-pair, a blank line is used to separate the tag-pair section from the subsequent move-text section.

As for the move-text section, this is simply the enumerated SAN moves

that occurred within the game [22]. At the end of the SAN move sequence, there is a termination marker that takes one of the following four values: *'1-0'* (White wins), *'0-1'* (Black wins), *'1/2-1/2'* (Draw), or *'*'* (Unknown result) [22]. Importantly, the termination marker at the end of the move-text section should always match the value of the Result tag in the STR. A single blank line then follows the move-text section, which is necessary to correctly separate and parse games from PGN database files, which are sequential collections of zero or more PGN games [22]. PGN files also use the file suffix of *'.pgn'* and the contents of such a file can be seen in figure 2.

The PGN specification also discusses the two formats of PGN, known as "import format" and "export format" [22]. Both formats adhere to the same set of rules mentioned above but differ in how strict they are regarding white space and other syntactic aspects of PGN. For example, the import format is much more flexible and is used in situations where a PGN file may have been manually prepared or handwritten [22]. Most chess software can still read import format, but their parser must be more sophisticated to manage the relaxed white-space rules and additional edge cases. In contrast, the export format is a stricter subset with additional rules regarding text justification, white space, and line length [22]. While these extra restrictions allow for simpler parsing techniques, it also means that this format is traditionally generated programmatically rather than by hand to ensure compliance with these stricter standards. Essentially, while the import format is flexible and meant for everyday use by human players, the export format is designed for the precise and cross-application encoding of chess games. Despite this, both variations are similar enough to be described by the same Backus-Naur Form found in figure 6.

```
<PGN-database> ::= <PGN-game> <PGN-database>
                   <empty>

<PGN-game> ::= <tag-section> <movetext-section>

<tag-section> ::= <tag-pair> <tag-section>
                  <empty>

<tag-pair> ::= [ <tag-name> <tag-value> ]

<tag-name> ::= <identifier>

<tag-value> ::= <string>

<movetext-section> ::= <element-sequence> <game-termination>

<element-sequence> ::= <element> <element-sequence>
                       <recursive-variation> <element-sequence>
                       <empty>

<element> ::= <move-number-indication>
              <SAN-move>
              <numeric-annotation-glyph>

<recursive-variation> ::= ( <element-sequence> )

<game-termination> ::= 1-0
                       0-1
                       1/2-1/2
                       *
<empty> ::=
```

Figure 6: Backus-Naur Form for PGN [22]

In addition, there is another minor PGN variation called "reduced export format", which is identical to the standard export format except for the removal of all commentary and optional tag-pairs from the PGN file [22]. Specifically, a game is said to be in reduced export format if it is already in export format and the following hold:

1. There are no additional tag pairs besides the Seven Tag Roster
2. There is no commentary (i.e. no move-text comments shown between curly braces)
3. There are no recursive annotation variations (i.e. no alternate move possibilities shown between parentheses)
4. There are no numeric annotation glyphs (i.e. no characters such as *'!'* used to highlight good moves)

These rules represent a minimum level of standard conformance required for PGN applications, meaning that the reduced export format is recommended by the PGN specification for archival and bulk storage of games [22]. Therefore, this format is incredibly useful for our PGN compression library and objective of alleviating PGN storage concerns for online chess platforms.

2.3 Existing Solutions for PGN Compression

2.3.1 Chess.com

Before starting our implementation, it is important to conduct further research into how online chess platforms currently manage the issue of PGN file compression. As mentioned in the introduction, the large storage requirements faced by online chess platforms suggest that they have already started to invest in creating new compression algorithms to tackle this large PGN file volume. This is also confirmed by Chess.com's Chief Technology Officer, Josh Levine, in a 2023 "Ask Me Anything" discussion on Reddit [30]. In response to one question about the issues Chess.com faces, he replies that "the biggest technological challenge is storage, hashing, compression, and the trade-offs between Disk and CPU" [30]. However, beyond this small piece of evidence, Chess.com provides no other information regarding exactly what their PGN compression techniques are. This is not particularly surprising, since as a privately held for-profit company, they are not incentivised to share their PGN compression algorithm with the wider chess community.

2.3.2 Lichess.org

Meanwhile, Lichess.org (Chess.com's closest competitor) takes the opposing philosophical approach to their technology. With the website's name being a literal combination of the words "libre" and "chess", this free and open-source chess platform is operated by a non-profit organisation and funded entirely through user donations [11]. As such, their GitHub page not only contains their server and client implementations but also the specific PGN compression algorithm they use on

their production servers. Along with publishing their algorithm Java source code, in 2018 they authored a blog post entitled *"Developer update: 275% improved game compression"* which detailed their algorithmic approach [37]. While this blog post does not cover how they store the tag-pair section of PGN files, their source code suggests that they store this metadata separately from the compressed moves and only link them together and generate the final PGN when requested by a user. Hence, while not offering an all-in-one algorithm for PGN compression, their approach to compressing the move-text section of a PGN file is still highly relevant to our work. From a high-level perspective, this compression algorithm works in two stages, firstly mapping each SAN move to an integer index before secondly encoding them into a bit stream using Huffman Coding [37].

Mapping Moves to Integers:

Firstly, the Lichess developers discuss how they assign each SAN move an integer index while minimising the number of distinct integers used [37]. If we recall that all moves in chess must be legal, it makes sense to first generate the complete set of legal moves for the current board position. Since the maximum number of legal moves in any position is 218, this significantly reduces the number of indices we need to enumerate any legal move [37]. For example, a novel approach would be to generate the legal set of moves for the current position, order them lexicographically, and then map the actual played move to its index in the lexicographically sorted list. For instance, consider the opening legal moves for White in figure 7.

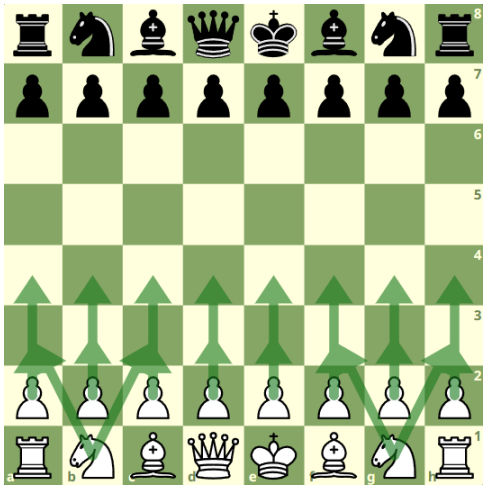

Figure 7: Opening legal moves for White

The legal moves for White sorted lexicographically are: *'Na3', 'Nc3', 'Nf3', 'Nh3', 'a3', 'a4', 'b3', 'b4', 'c3', 'c4', 'd3', 'd4', 'e3', 'e4', 'f3', 'f4', 'g3', 'g4', 'h3', 'h4'*. If White ended up playing *'e4'*, then we can map this SAN move to its integer index of 13, before repeating this same process of legal move generation, ordering, and index mapping for every subsequent move. While it may appear computationally expensive for Lichess to regenerate a set of legal moves for every SAN move, this is surprisingly efficient. The set of legal moves can be generated using a bit-board data structure, where each square on the chessboard corresponds to 1-bit in a 64-bit word [37]. Given a bit-board representing each piece's legal destinations, a few bitwise OR operations can combine them to efficiently compute the entire set of legal moves in a handful of CPU instructions [10].

As for the ordering, Lichess does not actually use a lexicographical sorting algorithm, as this arbitrary distribution of indices would be poorly compressed by the Huffman Coding algorithm in the next stage [37]. Instead, it is better to assign indices to legal moves in descending order of strength. For example, the strongest move in the current position would be assigned an index of 0, and the second strongest would be assigned an index of 1. This follows from the assumption that these strong moves are more likely to be played, and hence their indices should require fewer bits to encode [37]. There are numerous ways to

determine the relative strength of a given chess move, such as simply computing each player's piece values or using minimax search trees to evaluate all possible outcomes. However, since this strength calculation must occur once for every legal move in each board position, it is crucial that the heuristic is deterministic, fast to compute, and a decent approximation of the true move strength. To accomplish this goal, the Lichess developers settled on five criteria to consider when scoring each move and used Simultaneous Perturbation Stochastic Approximation to optimise the weights of each [37]. These criteria in descending order of importance are as follows:

1. Prefer pawn promotions (if available).
2. Prefer moves that capture opponent pieces. High-value captures are better.
3. Prefer moves that **do not** move your pieces onto squares defended by opponent pawns.
4. Prefer moves that strategically position pieces, guided by the "piece-square" table. This data structure assigns weights to each square on the board for each piece type, reflecting the average effectiveness of having that piece on that square.
5. Prevent ties using lexicographical order.

These criteria were picked as they are all individually quick to compute for each legal move while also approximating the evaluation a human player performs when considering each move [37]. In addition to being a suitable heuristic and efficient, this ordering algorithm is also deterministic which is an essential property for ensuring both the encoder and decoder remain in alignment.

Compression via Huffman Coding:

After detailing how they generate all legal moves, heuristically order them by strength, and map them to their corresponding strength index, the Lichess developers then move on to discussing how they convert this sequence of indices into the final bit stream using Huffman Coding [37]. This entropy encoding method takes each index (also known as a

symbol) and converts it into a binary code. The length of a binary code is determined by the respective symbol's frequency in a PGN dataset, such that the more frequent symbols (i.e. strength indices from 0 to 30) are associated with shorter binary codes. The original symbol sequence then gets encoded into one bit stream by appending all the binary codes together [37].

For the decoding process, the same two stages happen in reverse. The bit stream is split back into separate binary codes, these codes are decoded back into the original symbols, and these symbols (i.e. strength indices) can be used to recover the original SAN move that was played on each turn [37]. By using this compression algorithm, the Lichess developers were able to achieve 4.4 bits-per-move on average (a 67% reduction from their previous 13.4 bits-per-move format), save 70GB from their database, and significantly slow the growth rate of their PGN data [37]. This provides us with a solid foundation and approach to PGN compression that we will expand upon in this project.

2.4 Information Theory

To understand how the Huffman Coding algorithm works, it is important to first grasp the basics of Information Theory and how it relates to compression algorithms. Established by Claud Shannon in the 1940s, Information Theory is the mathematical study of the storage and communication of information [17]. Before this, "information" was a poorly understood and abstract concept. It was not until Shannon's paper called *"A Mathematical Theory of Communication"* that it became clear that "information can be treated very much like a physical quantity, such as mass or energy" [38].

Information is traditionally measured in bits, where a single bit of information allows you to differentiate between (or encode) 2 equiprobable outcomes (e.g. 0 for the first outcome and 1 for the second) [38]. For instance, 4 equiprobable outcomes carry 2 bits of information each, and 8 equiprobable outcomes carry 3 bits each. Therefore, information is simply the number of bits you require to differentiate (or encode) what outcome has occurred from a set of possibilities. The general form of

information can be computed using equation 2.1, where $p(x_i)$ is the probability of outcome x_i in the random variable x [38].

$$I(x_i) = -\log_2 p(x_i) \tag{2.1}$$

While information tells us how many bits are associated with a specific outcome, entropy is the average amount of information for all outcomes of a random variable [38]. This makes entropy a useful measure of uncertainty, as random variables with high entropy require a lot of information on average to determine their outcome. Meanwhile, random variables with low entropy require little information to decide and are more certain. The entropy of any discrete random variable x can be computed using equation 2.2 [38].

$$H(x) = \sum_{i=1}^{m} p(x_i) \log_2 p(x_i) \tag{2.2}$$

To understand how entropy can represent an abstract concept such as uncertainty let us compare the entropy of a fair and biased coin toss. Using equation 2.2, we find the entropy of a fair and biased coin (with a 90% chance of heads) to be 1 bit and 0.47 bits respectively. As we can see, the more uncertain fair coin has higher entropy than the biased coin. This is to be expected, since if you were tasked with predicting the outcome of the biased coin, you can do so with greater confidence due to the probability distribution. This equation for entropy, also known as Shannon Entropy, is pivotal to the field of Information Theory and simultaneously represents the lower bound on the average number of bits an outcome or symbol can be compressed into [38].

2.5 Huffman Coding

Numerous encoding algorithms attempt to approach this lower bound provided by a source's entropy. Such an algorithm is commonly referred to as an Entropy Coding, and we have already briefly discussed one of the most popular approaches known as Huffman Coding [16]. Devel-

oped by David Huffman while at MIT, his 1952 paper *"A Method for the Construction of Minimum-Redundancy Codes"* established Huffman Coding and laid the foundation for many other important compression technologies (e.g. JPEG and MP3) which rely upon it [16]. While other entropy encoding methods have since been developed (e.g. Arithmetic Coding and Asymmetric Numeral Systems), Huffman Coding's simplicity, speed, and near-optimality have been instrumental in its continued relevance within Information Theory and compression algorithms [16]. The common pseudocode representation of the Huffman Coding algorithm can be seen in algorithm 1, where the input C is a set of symbols paired with their frequency within some source data set. An example visualisation of the algorithm output, a Huffman Tree, can be seen in figure 8.

Algorithm 1 HUFFMAN-TREE(C) [32]

```
1: n ← |C|
2: Q ← C                          ▷ is a min-priority queue keyed by frequency
3: for i ← 1 to n − 1 do
4:     Allocate new node z
5:     z.left ← x ← EXTRACT-MIN(Q)
6:     z.right ← y ← EXTRACT-MIN(Q)
7:     z.freq ← x.freq + y.freq
8:     Q.INSERT(z)
9: return EXTRACT-MIN(Q)          ▷ return the root of the tree
```

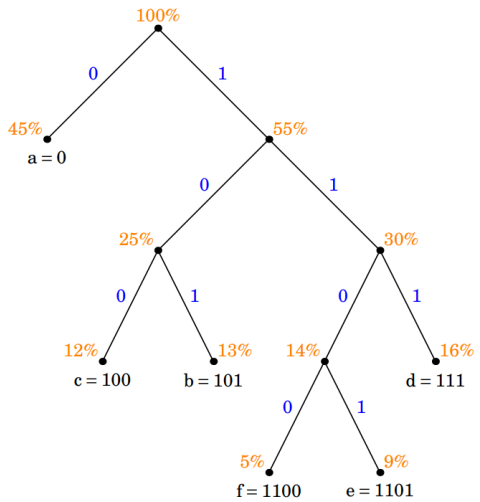

Figure 8: Huffman Tree example, with symbols and their associated codes [32]

Evaluating this algorithm in terms of asymptotic runtime, building the initial binary priority queue will take $n * O(\log n)$ insertions, the loop will take $n * O(\log n)$ insertions along with $2n * O(1)$ extract-min operations, and returning the root of the tree will take $O(1)$ [32]. Altogether, producing a Huffman Tree with a binary priority queue takes $O(n \log n)$, where n is the number of symbols in our encoding. Therefore, in addition to the algorithm's simplicity, the low linearithmic runtime for constructing such a useful data structure is also advantageous for numerous encoding applications. As for the encoding and decoding operations themselves, these correspond to a traversal of the Huffman Tree, which takes only $O(n)$ using any standard tree traversal algorithm [42]. This can be further improved by converting the Huffman Tree into a bi-directional lookup table, where each symbol is mapped to a binary code and vice-versa, allowing for $O(1)$ encoding and decoding. The space complexity of the Huffman Tree is trivially $O(n)$, due to storing one node per symbol and the use of a priority queue as the underlying data structure [42].

The Huffman Coding algorithm is also provably optimal for symbol-by-symbol encoding with a fixed probability distribution [16]. This makes it ideal for compressing symbols that are independent of each other (i.e. a symbol is not influenced by its neighbours) and that originate from the same probability distribution. This property is commonly abbreviated to IID, for "Independent and Identical Distributed" [16]. If a set of symbols does not adhere to this IID property, then other more flexible entropy encoding methods such as Arithmetic Coding can provide better compression than Huffman Coding [16].

However, from our preliminary analysis of Lichess's approach to PGN compression, we can reason that the strength indices they use as their symbols are indeed independent and mostly identically distributed. Firstly, it is trivial to show that each symbol is independent since playing the best legal move (e.g. symbol 0) on your last turn does not influence how strong of a move you will play on the subsequent turn. Secondly, while it may not be possible to find the exact probability distribution of these strength indices (due to player skill differences,

game time constraints, etc), we can instead construct a representative frequency distribution from a large dataset of PGN files and use it to estimate the underlying probability distribution [37]. While individual PGN files may differ from this distribution, on average it would provide a good model which we can use within the Huffman Coding algorithm. For these reasons, we would expect that using Huffman Coding for PGN files would indeed approach the optimal entropy limit, within a small margin of error due to the approximated probability distribution.

3 Project Management

As a reminder, the goal of this project is to develop a universal library to more efficiently compress PGN files, motivated by the large storage costs that online chess platforms are incurring. Due to this, the nature of the project involves a mix of research and experimentation in addition to the implementation of the library in Rust. While an existing Lichess solution does exist, they focus entirely on move-text compression and store any tag-pairs in a database tightly coupled with their existing infrastructure. This means that our approach to this problem will be fundamentally different, as we are attempting to build a library capable of compressing both the tag-pairs and move-text sections of a PGN file into a single encoded bit stream, instead of splitting the sections across multiple database tables like in Lichess's case. By doing so, our compression library will remain independent of any other aspects of a backend tech stack, providing a more modular and integrated toolkit for developers to use. Through effective project management, the end goal of this project is to provide a robust, well-documented Rust and WASM library for others to use, and to hopefully see an indication of its use within the chess community.

3.1 Development Methodology

To achieve these goals, we have selected Test Driven Development or TDD as the primary software development methodology. This methodology involves a continuous cycle of writing unit tests, writing code to pass the tests, and then refactoring to ensure simplicity [48]. Since we are developing a library with an API that will be used by other developers, our code must be well-tested and simple to understand. TDD will assist with this via its rigorous focus on testing and refactoring, which should result in our library succinctly fulfilling our goals with minimal technical debt.

However, given the lengthy time horizon for developing this project, it is also important to have some method for planning long-term goals and key architectural components of the library. For this purpose, we

have also decided to use some Agile Development principles through weekly sprint management sessions [33]. During these sessions, we attempt to plan and organise the necessary tasks for that week, while also considering other deadlines and responsibilities. From this session, we construct a more comprehensive and insightful progress plan for our project, while replanning tasks when the need arises. The tasks required for each sprint are implemented within the library using TDD, combining the robustness of frequent testing with the longer-term and iterative Agile approach.

In addition to these software development techniques, a developer's log is also updated daily as changes are made to the repository. While there is no formal structuring to this journal, it contains various thoughts, concerns, and justifications regarding choices made during the development of the project. This historical record is also critical to the project's management, as it eliminates the possibility of re-evaluating previous decisions, such as what data structures to use or how to organise common utilities between files. With justifications for these actions readily available, time wasted on fruitless re-evaluation or refactoring is minimised.

3.2 Languages and Tools Used

Alongside our development methodology, another key aspect of our project management is the selection of languages and tools we will use while building the universal library. Due to our desire to compile directly to WASM, sufficient production-level support for this low-level target was required for our primary programming language. At the time of writing, only C, C++, Rust, and Go meet this criterion [2]. Go can be swiftly eliminated from consideration due to its garbage collector which would introduce additional overhead to our compression algorithms. As for C/C++, while we have prior experience with these systems programming languages, the potential for memory errors and complicated package management also makes them unattractive options. This leaves Rust as the programming language of choice for this project, with its strong memory safety, modern package manager, and

low-level speed [18]. Rust also has excellent multi-threading support, as its type system makes it impossible to accidentally introduce data races [43]. This will be very useful for benchmarking our strategies and allow for the full utilisation of all system threads. Alongside the enjoyable development experience of writing Rust, it also has best-in-class tools such as "wasm-pack" for WASM compilation [8].

We have chosen WASM as our compilation target due to our goal of developing a universal library that can be used natively in various programming languages. This is a much deeper level of integration than simply being cross-platform, as any number of Python or JavaScript packages can make this claim but cannot be used outside of the language ecosystem from which they are developed. By compiling our library to WASM, it becomes possible to use this single binary in any language that has a WASM runtime implementation, providing the same universal and efficient API to all such languages. For instance, there are production-ready WASM runtimes for Rust, Python, JavaScript, C/C++, .NET, Go, Java, and Ruby [52]. Hence, WASM is a critical tool and technology that is required for the successful deployment of our library.

Finally, this project also uses Continuous Integration and Continuous Deployment (CI/CD) to further streamline the software development lifecycle [29]. For each commit to the repository, a GitHub action will automatically trigger a complete rebuild of the library before running all unit and integration tests. If these tests pass without issue, the commit is merged into the repository and the percent of code coverage is recorded. If any test fails, an alert is sent back to us, with information about what went wrong. Through this CI/CD workflow, we establish a second line of defence against introducing bugs to the main git branch, along with an emphasis on maintaining sufficiently high code coverage. In fact, as of the last commit, 97% of all lines of code in the library have at least one unit test covering them.

3.3 Legal, Social, Ethical and Professional Issues

Due to the nature of this project as a research and development-oriented exploration, there are no legal, social, ethical, or professional issues to consider. The project does not require any personal user data and only requires access to a Lichess PGN database to benchmark the compression algorithms against real PGN files. These databases are available at *database.lichess.org* and released under the Creative Commons CC0 licence, making them free for use in research, publication, or commercial purposes [31].

4 Library Implementation

4.1 Design and Development

Given that this Rust library is the primary output of our project and exposes a public API, the architectural design of the library must lend itself to these goals. Fortunately, Rust has a sophisticated module system that allows us to split code into logical units and manage the public/private visibility between these units [44]. While this is like the class-based access patterns you find in object-oriented languages, it is critical to acknowledge that Rust is not strictly object-oriented. Rust draws inspiration from the simple procedural style of C, while also incorporating multi-paradigm concepts from object-oriented and functional languages alike [18]. This provides great flexibility, but in general, Rust encourages the separation of data from control flow in a more functional style [9].

For this reason, we separate our library (or crate, as they are called in the Rust ecosystem) into five primary modules, one for the general-purpose PGN data struct and the others for each compression strategy implementation. The compression modules share their common functionality through an additional utility module, enabling effective Separation of Concerns and reducing code repetition. Additionally, Rust has a built-in test runner provided via its "cargo" command line tool [45]. This allows us to simply write our unit tests at the bottom of each module and rerun them frequently, following our TDD approach. As for the API of the library, each compression strategy only exports its relevant compression and decompression functions, while hiding all the intermediate utilities and data structs. By providing this minimal declarative API, we simplify the process of using our library for any third-party developers.

Since this library is compiled to WASM, we must also consider any constraints this imposes on our design and architecture. Fortunately, there are few requirements for exporting and compiling a Rust function to WASM. Critically, these functions can only accept and return primitive types (i.e. no custom structs), and the functions must be explicitly

marked with the "wasm-bindgen" macro [8]. Rust macros are the language's equivalent of metaprogramming, meaning that at compile-time the "wasm-bindgen" macro automatically wraps the function in the necessary WASM compatibility layer. This layer is then detected by tools such as "wasm-pack" which identifies these functions and compiles them directly into the final WASM module [8]. Additionally, any functions used within a "wasm-bindgen" function will automatically be in-lined and compiled to WASM as well. This makes the compilation process very straightforward, however, we do have to ensure we create primitive-only variations of all our compression and decompression functions. An overview of the architecture for the four strategies can be found in figure 9.

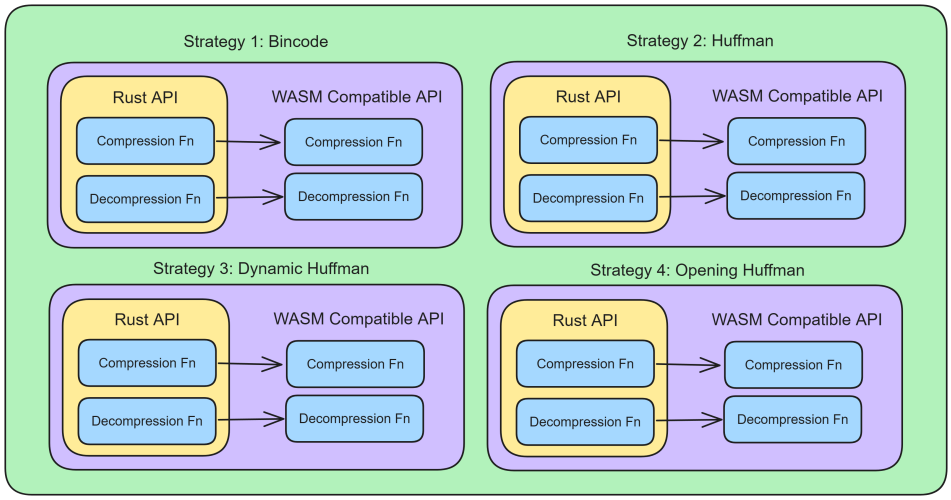

Figure 9: High-level architecture of the library

4.2 PgnData Structure

Given that this library operates on PGN files, we require the ability to parse these files into some intermediate data structure which the compression and decompression algorithms can easily manipulate. With our background research into the PGN specification, we already know that the "reduced export format" is the recommended format for the archival storage of PGN files [22]. Crucially, this format only contains the Seven Tag Roster (STR) and SAN move sequence with no com-

mentary or annotations. Since this represents the minimum level of standards compliance for PGN files and removes most of the unnecessary data, this is the best format to use for our compression purposes.

For our PgnData struct to properly model this format, we first declare a PgnHeaders struct which contains seven public fields for the Event, Site, Date, Round, White, Black, and Result tag-pairs. We then access this struct via the "headers" field within the PgnData struct, allowing for simple mutable access to any of these tag-pairs. For instance, we could reassign the Event tag with `pgn_data.headers.event = 'Event 1'`. While we considered using a HashMap for storing the tag-pairs (since they are just string key/value pairs), this was rejected since it would allow additional tag-pairs to be added besides those outlined in the STR. This would make it possible to violate the rules of the reduced export format, causing unexpected behaviour in other aspects of the library. The moves of a PGN file are located under the "moves" field of the PgnData struct, within a Vec (Rust's equivalent of a dynamic array) which allows for easy and efficient $O(1)$ access to each element [46].

For parsing PGN files and converting them into PgnData structs, we originally considered writing our own parser. However, given the nuances of both PGN import/export formats, this would have been a lengthy task with many edge cases to consider. Instead, we use a third-party Rust crate called "pgn-reader", which provides a pre-built parser for PGN files with many options for configuration [25]. Using this parser, we implement a "from_str" method on the PgnData struct, allowing us to generate a fully populated PgnData struct from any valid PGN game string. For example, `let pgn_data = PgnData.from_str(pgn_string)` would automatically parse the STR and SAN moves before assigning the final PgnData struct to the "pgn_data" variable.

In addition to parsing, we need a method for converting a PgnData struct back into a PGN game string, with the correct white-space, line length, and tag-pair order. This is crucial for the decompression process, where we need the reconstructed PgnData struct to generate the final decoded PGN file. Doing so also allows us to abstract away

the underlying textual representation of a PGN file and focus on manipulating data within the PgnData struct itself, with the confidence that the final formatted output will be correct. Hence, we implement the "fmt" method on the PgnData struct for converting it to the reduced export format, ensuring that the STR is printed in the correct order, the SAN moves are correctly enumerated, and the line length is limited to 80 characters. With the "from_str" and "fmt" functions working correctly and passing the unit tests, we now have a sufficient data structure for the reduced export format which can be created from and outputted to a PGN game string. A diagram offering an overview of the PgnData struct can be seen in figure 10.

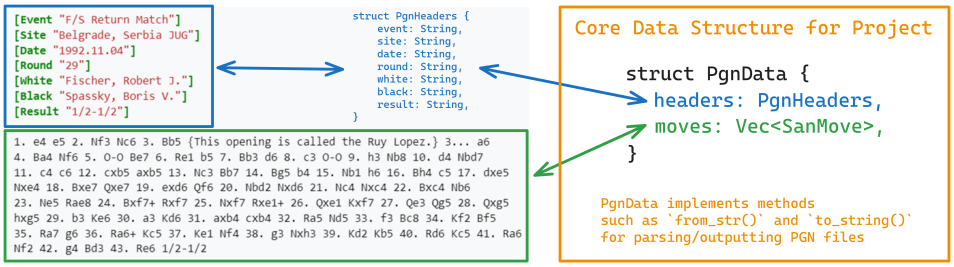

Figure 10: Overview of PgnData struct

4.3 Strategy 1: Bincode

Considering that our project is primarily an exploration into the realm of PGN compression, it is important that we build our knowledge and experience iteratively with each compression strategy we develop. Therefore, instead of trying to outperform the Lichess implementation with our first strategy, we opt for creating a simple and elegant compression algorithm that opens the door to further optimisation and experimentation. Crucially, this strategy allows us to conceptualise the library's API and acts as a proof-of-concept for our PgnData struct and its feasibility for use in our later compression strategies.

To create this minimal compression algorithm, we are focused primarily on how to serialise a PgnData struct into some sequence of bytes that can be further compressed with any general-purpose compression algorithm. While this approach does not incorporate any chess-specific

knowledge, this also means that we can use a wider range of pre-existing crates to rapidly develop this strategy and verify our approach. While searching Rust's central crate registry (aptly named "crates.io"), we found a popular crate called "bincode" under the serialisation category with over 60 million downloads [35]. According to its description, this crate provides "a binary serialisation/deserialisation strategy for transforming structs into bytes and vice versa" [35]. After reading the associated documentation, we can verify that this crate would indeed be capable of serialising our PgnData struct. Hence, after writing some unit tests as per TDD and creating the "compress_pgn_data" and "decompress_pgn_data" functions in the relevant Rust module, we have a working PgnData serialisation prototype.

As for compression of the serialised bytes, we use another Rust crate called "flate2", and the ZlibEncoder it provides [21]. Zlib is a well-known compression library that adapts the lossless DEFLATE algorithm (which incidentally uses Huffman Coding) by providing an additional header and footer to the binary stream for error detection purposes [20]. Today, Zlib is the de-facto standard way to use DEFLATE, so much so that they are often used as interchangeable terms. For instance, Zlib is used within Git, OpenSSH, and FFmpeg, as well as being responsible for decompressing the Linux kernel image at boot time [20]. Given its importance to the field, Zlib is the natural choice for the general-purpose compression algorithm we require. After positioning the ZlibEncoder to compress the serialised bytes (and vice-versa for decompression), our first strategy works correctly and passes all unit tests. A visualisation of the final encoder and decoder for the Bincode strategy can be seen in figure 11.

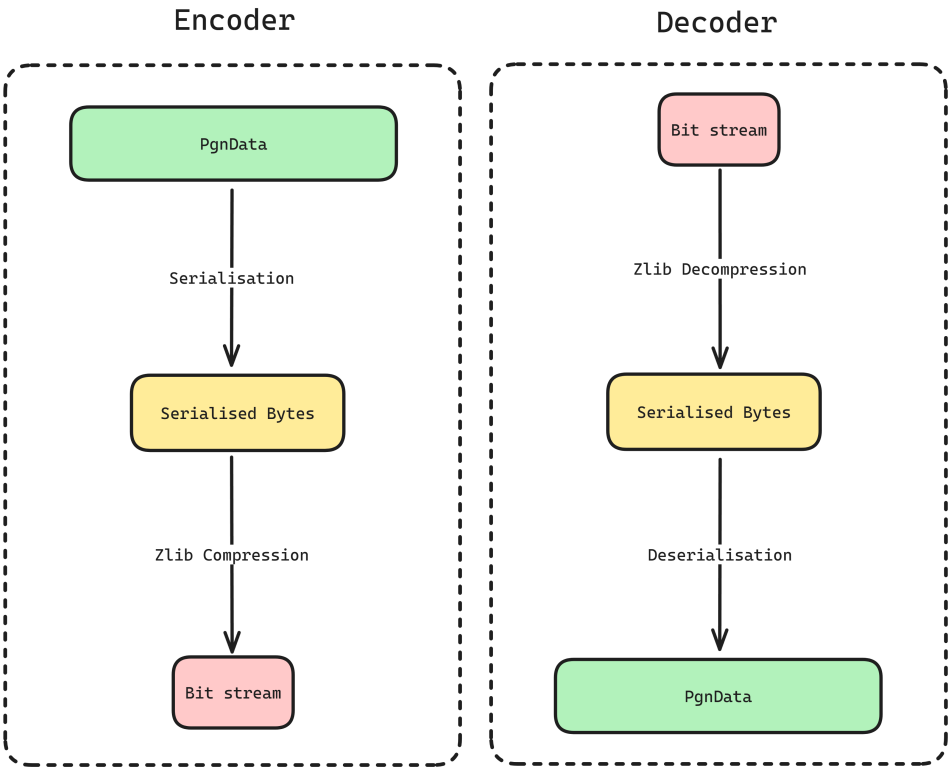

Figure 11: Bincode encoder/decoder control flow

After some preliminary testing, we are correct in our assumption that the lack of chess-specific knowledge will indeed impact compression performance. Even without running an entire benchmark, the average bits-per-move of the Bincode strategy appears to be much larger than Lichess's 4.4 bits-per-move. However, the benefit is that the compression and decompression speeds are extremely fast, due to the highly optimised "bincode" and "flate2" crates. Additionally, this general approach will provide useful experience when it comes to dealing with the PgnHeaders in the subsequent strategies. This is because it simply encapsulates the STR strings and does not contain any other chess-specific information we can exploit for compression ratio improvements. For this reason, traditional compression algorithms such as Zlib will perform much better on this part of the PgnData, while other domain-specific techniques are applied to the SAN moves.

4.4 Strategy 2: Huffman

For our next strategy, we aim to incorporate some chess-specific knowledge into our compression algorithm, with the hope that this will result in better compression ratios than the initial Bincode strategy. Recalling Lichess's existing implementation, they have a sophisticated methodology of compressing the move-text section of a PGN file by first using move strength ordering and then Huffman Coding [37]. From their demonstration, we can see that this is a promising method of utilising chess-specific knowledge to reduce final compression sizes. However, they do not discuss how they compress and store the PGN tag-pairs, meaning that we must implement this ourselves and somehow concatenate this data to the move-text binary stream. Therefore, by taking Lichess's approach and extending it to encode PGN tag-pairs too, we should produce a compression algorithm that approaches 4.4 bits-per-move on average, with some additional overhead due to the tag-pair section we are also encoding.

Firstly, we need a method of converting the PgnHeaders struct into a compressed binary stream. Luckily, from our development of the Bincode strategy, we know that we can serialise any struct with the "bincode" crate and then compress the bytes with the ZlibEncoder from the "flate2" crate. Given that we will need to similarly compress and decompress the PgnHeaders struct in our other compression strategies, it is best if we generalise this behaviour into a shared utility function called "compress_headers", which simply takes a PgnData reference as input and returns the compressed binary stream of its tag-pairs, or an empty stream if all tag-pairs are empty strings to save additional space. We place this at the start of the final binary stream, with the compressed move-text section being appended to it.

As for compressing the moves of the PgnData struct, we follow Lichess's two-stage approach of mapping all moves to their strength indices before then compressing these indices using Huffman Coding. To compute the strength index of a move, we start with generating the set of legal moves available on each turn (using the default chessboard setup for the first move) and then regenerate this set after each move is encoded.

The legal moves are computed using the "shakmaty" crate, which contains many convenient chess data structures and utilities [26]. After generating the legal moves, we implement a copy of Lichess's heuristic evaluation algorithm from section 2.3.2, using the same criteria and weights to calculate a strength score for each legal move [37]. Finally, these legal moves are sorted in descending order of strength, the current SAN move is found within this list, and replaced with the corresponding index. We then make sure to play this move on a "shakmaty" chessboard data structure, which the "compress_moves" function uses to compute the new set of legal moves for the next turn.

After the moves have been converted to indices, we initialise a HashMap that contains index/frequency pairs from a Lichess PGN database of over 16 million games from December 2017 [31]. In the future, we reuse this database for benchmarking our compression strategies, in which case it will simply be referred to as the Dec2017 database. Within this database, players picked index 0 (the best move) 225,883,932 times while index 50 (the 51st best move) occurred only 26,341 times. This HashMap is then used by the "huffman-compress" crate to generate a Huffman Book (for encoding symbols), and a Huffman Tree (for decoding symbols) according to the frequency distribution within the HashMap [23]. Using the Huffman Book, we iterate over the strength index sequence, converting each one into a binary code. Additionally, since these binary codes are instructions on how to traverse the Huffman Tree (i.e. go left if 0, right if 1) until reaching a leaf node, we can safely append them together without the need for a separation symbol. Instead, the decoder continuously reads from the move-text binary stream until it reaches a leaf node, where it will emit the decoded symbol and return to the root of the Huffman Tree to resume reading.

At this point, we have two functions for compressing each part of the PgnData struct, "compress_headers" and "compress_moves". All that is left is concatenating them together. However, since both the tag-pair and move-text sections can be any arbitrary number of bits once encoded, we require some method of informing the decoder about the

length of the encoded tag-pair section at the start of the stream. Otherwise, the decoder would not know where to stop decoding the tag-pairs or start decoding the move-text. Additionally, if the tag-pairs were empty, we need some method of informing the decoder of this so that it can immediately start decoding the move-text. After brainstorming potential solutions, we decided to use a 1-bit flag at the start of the final binary stream to signal to the decoder if there is a tag-pair section that requires decoding or not. If the original PgnHeaders struct is empty, this bit flag is set to 1, telling the decoder that all subsequent bits belong to the move-text section and that it should use the Huffman Tree to decode the SAN moves. Inversely, if the PgnHeaders contain any information, the bit flag is set to 0 and the next 7 bits tell the decoder the length of the compressed tag-pair stream in bytes. We use bytes (not bits) as the unit since the "compress_headers" function will always pad the tag-pair stream to the nearest byte, and with 7 bits we can inform the decoder of longer lengths up to 128 bytes. While this does impose a theoretical upper limit on the total length of tag-pairs our Huffman strategy supports, benchmarking across thousands of PGN files never uncovered any games which exceeded this limit. If the limit were exceeded, the error would be caught, and the compression process would gracefully terminate. An overview of the encoder and decoder control flow can be seen in figure 12.

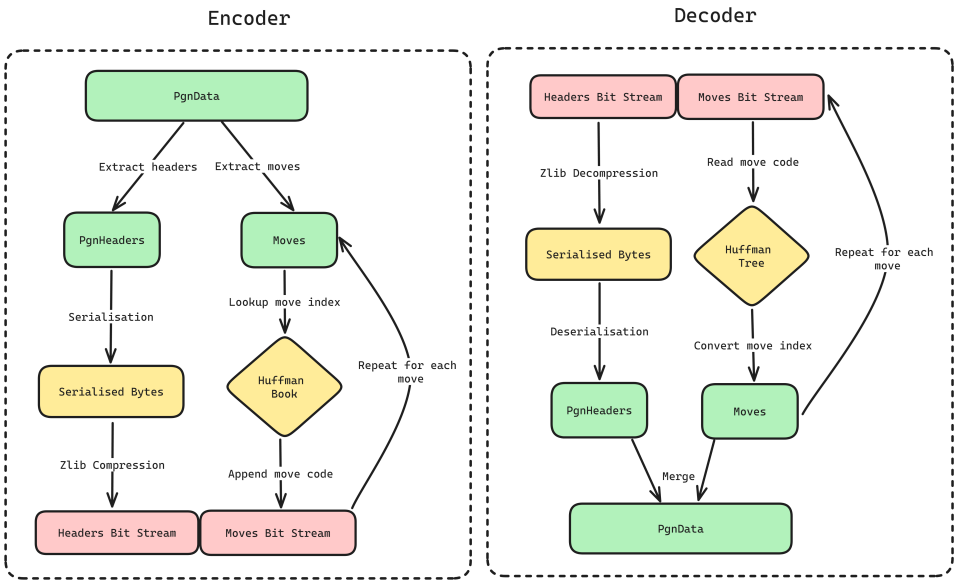

Figure 12: Huffman encoder/decoder control flow

To briefly assess the compression algorithm and compare it against Lichess's state-of-the-art (SOTA) solution, we run it against some PGN files that have had their tag-pairs removed. This is to account for the fact that Lichess's 4.4 bits-per-move only corresponds to their move-text compression, so by removing the tag-pairs from these sample PGN files we can more accurately compare our move-text compression to theirs. A notable exception is that even with these empty tag-pairs, our approach still includes the 1-bit flag that informs the decoder that no tag-pairs are present. This adds a small amount of overhead that increases our average bits-per-move, but when removing the extra flag bit from the calculation, we indeed see that our efficiency very nearly matches Lichess's. This is great news, and indicative of the fact that integrating chess-specific knowledge into our compression algorithm does indeed improve compression ratios just as Information Theory would suggest. This indicates that by further incorporating domain-specific knowledge into our compression strategies, we may continue to improve and hopefully surpass Lichess's SOTA solution.

One downside of this Huffman strategy is that it is slower than the Bincode strategy, which uses more general-purpose and rapid compression

algorithms such as Zlib. This suggests that there is a greater performance overhead associated with the move-generation and Huffman-based approach. Additionally, by using a static frequency distribution for the Huffman Coding algorithm, we are implicitly assuming that each player performs about the same as the average chess player, with no regard to individual skill level. For instance, if two perfect chess AIs played each other, they would consistently pick the index 0 move every time. Due to index 0 having the highest frequency in the index/frequency HashMap, it has the shortest binary code in the resulting Huffman Coding. Hence, this perfect chess game would be compressed to a minimal size. In contrast, if two inexperienced players consistently picked indices greater than 30, their moves would be assigned longer binary codes and result in a larger final binary size. If we instead update the frequency distribution at runtime, we may be able to account for these skill differences and ensure that even poor gameplay still yields near-minimal compression sizes.

4.5 Strategy 3: Dynamic Huffman

Building upon the findings of our previous Huffman strategy, we now seek to adapt our approach such that each player's in-game performance is used by our compression algorithm. In other words, if a player is consistently playing the n^{th} best move, we want to artificially "boost" the frequency of this index in our Huffman Coding such that subsequent moves of similar strength are assigned smaller binary codes. This should help our strategy capture a rough estimation of a player's skill and use this chess-specific knowledge to further improve compression ratios. To implement this idea, we can use the previous Huffman strategy as a starting point but augment it to have two Huffman Codings (one for each player) that have their index/frequency pairs updated after each move is encoded. It is important to remember that we cannot update the frequency distribution first and then use it to encode the same move, as it would be impossible for the decoder to reverse this process and determine the original symbol. In other words, instead of using a player's n^{th} move to update the distribution and encode the same n^{th} move, this new distribution will be used to

encode the $(n+1)^{th}$ move instead. Moreover, we can reuse our previous "compress_headers" and "decompress_headers" utilities, as we are not looking to change how the PGN tag-pairs are being compressed for this strategy. Other aspects of the Huffman strategy, such as the use of a 1-bit tag-pair flag, will also stay the same. The only adaptations being made are to the "compress_moves" and "decompress_moves" functions.

For this approach to work, we must focus on updating two frequency distribution HashMaps for the White and Black players. This is because the Huffman Coding can simply be recreated every turn using the relevant HashMap for the current player. As such, we need to conceptualise what this update function will look like, given the original HashMap and the index of the move that was just played. For example, one novel approach would be to double the frequency of index n in the HashMap whenever index n is played. The downside with this approach is that index $n+1$ would remain unaltered, which is undesirable given how similar in strength those two moves are. Additionally, when doubling a small frequency, it is unlikely that this increase will be enough to shorten its binary code in the resulting Huffman Coding.

We instead opt for a Gaussian Curve which would be centred around index n and discretely calculated for every index (i.e. 0-255) in the HashMap. The benefit of this approach is the ease-of-control it provides for the deviation (or spread) and height of the curve, making it possible to optimise these parameters for the best compression ratio. The general form of the Gaussian Curve can be seen in equation 4.1, where a is the height of the curve at its peak, b is the position of the peak, and c is the width of the curve [15]. Given that we want this curve to be centred around the index n that was just played, we use this as the parameter b, leaving only parameters a and c that require further exploration and optimisation. Before diving into this optimisation process, it is useful to first observe an example visualisation of the HashMap updating process in figures 13, 14, and 15.

$$f(x) = ae^{-\frac{(x-b)^2}{2c^2}} \tag{4.1}$$

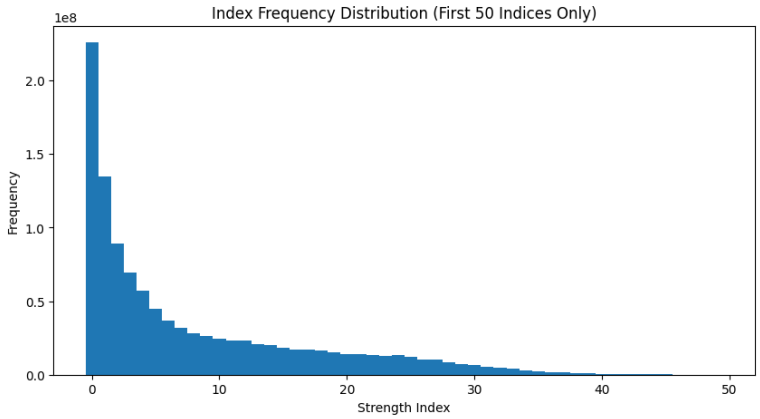

Figure 13: Original Index Frequency Distribution

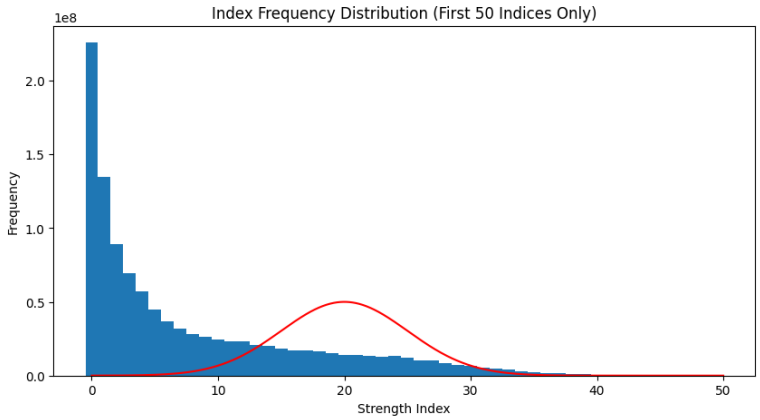

Figure 14: Example Gaussian Curve created by encoding strength index 20

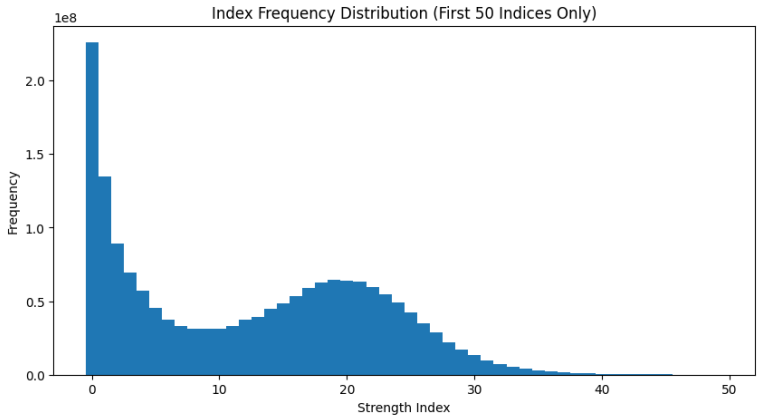

Figure 15: Updated Index Frequency Distribution

As seen in figures 13, 14, and 15, by using a Gaussian Curve we can update the frequency distribution with specific control over the spread and magnitude of each change. Given this approach, we can boost the frequency of specific moves as the game progresses for each player, hopefully resulting in a distribution that is more closely related to their actual skill level, rather than a static approximation of the average player. However, the primary challenge with this approach is deciding upon what parameters for the height and deviation of the curve to pick, such that the final compressed PGN size is minimal. As this is an optimisation problem, we can employ a gradient descent algorithm to try and find the optima.

We begin by writing a simulated annealing algorithm in a separate Rust module outside of the main library and within the command line interface (CLI) subproject. The idea behind this approach is to start with some random height and deviation parameters (h_1, d_1), and randomly perturb them to yield a new set [3]. We then calculate the average bits-per-move that these new parameters would yield in our Dynamic Huffman strategy, to see if it is better or worse than the previous set of parameters. An additional variable T is used to control the probability of accepting a worse solution, which is initially high (to promote exploration) and decreases as we perform more iterations [3]. While this may appear to be an effective way to discover the optimal height and deviation of our Gaussian Curve, it was exceedingly difficult to find a random value to perturb both these variables. Since the state space is continuous and infinite, too large a random jump would overshoot the optima, while small jumps risk never reaching it. For this reason, we shall discontinue this approach and use an alternative gradient descent algorithm that is better suited for exploring a wider range of state space values at the same time.

Therefore, we switch to using a genetic algorithm instead. A genetic algorithm is an adaptive heuristic search algorithm which uses the concepts of natural selection and genetics to explore a state space across multiple generations [27]. Each generation contains a collection of individuals representing a single solution or (h_i, d_i) pair. The most "fit"

individuals are crossbred with each other to create a new generation of children, which each undergo a slight chance of random mutation [27]. After searching for existing genetic algorithm crates available to us in the Rust ecosystem, it was disappointing to find that none of them are well-suited to precisely explore two continuous parameters at the same time. For this reason, we need to write our own genetic algorithm from scratch, a task that we fortunately have prior experience with. Specifics regarding our algorithm and optimisation process can be found in section 5.3, as it pertains more to the implementation of the CLI sub-project, rather than this library. With the discovered optima of $height = 1,895,192.82$ and $deviation = 1.29$, equation 4.1 produces a Gaussian Curve to update each player's HashMap on their respective turns, a process outlined in figure 16.

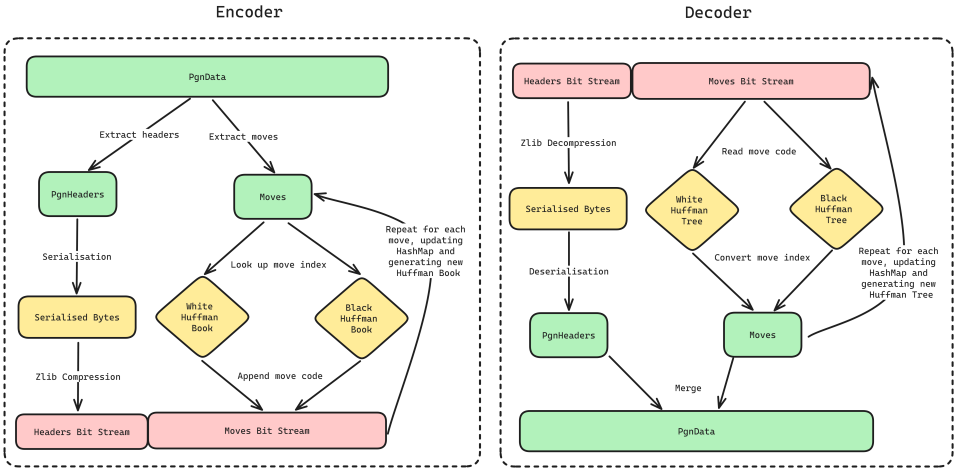

Figure 16: Dynamic Huffman encoder/decoder control flow

After running this Dynamic Huffman strategy over a sample of PGN files, we are happy to see that it marginally surpasses our previous Huffman strategy and approaches Lichess's SOTA solution of 4.4 bits-per-move. This is promising and supports the idea that integrating additional chess-specific knowledge into these compression algorithms does indeed yield minor improvements. However, the major downside of this strategy is that it is significantly slower than both the Huffman and Bincode strategies. This is because updating the HashMaps after every move also necessitates rebuilding the Huffman Coding as well.

So, instead of the one-off cost of $O(n \log n)$ to construct the static coding, reconstructing it for m moves in a single game yields $O(mn \log n)$ [32]. Given that on average, there are 79 moves in a chess game, this is a significant performance impact that makes this algorithm about twice as slow as the Huffman strategy [1]. Another downside is that despite surpassing the Huffman Strategy, it does so only marginally, with less of an improvement than expected. With these higher performance costs and near-negligible compression gains, it appears that trying to approximate each player's relative skill within the algorithm is not a viable solution. In fact, most players seem well represented by the default frequency distribution used in the Huffman strategy, and the marginal improvement we do see is likely due to overfitting in the optimisation process. For this reason, our next strategy will focus on using chess-specific knowledge that pertains to all chess games universally, rather than trying to capture and incorporate trends from individual games like this Dynamic Huffman approach.

4.6 Strategy 4: Opening Huffman Coding

For our final strategy, it is important to reflect upon the lessons learnt from our previous compression algorithms to distill this knowledge into our last best attempt at compressing PGN files. From our Bincode strategy, we learnt how to efficiently compress non-move data such as the PgnHeaders using traditional serialisation and compression techniques. The Huffman strategy taught us that incorporating chess-specific knowledge and information can significantly improve the compression of the move-text section, in line with the lessons of Information Theory. Finally, the Dynamic Huffman strategy taught us not to focus too much on capturing specific in-game trends, such as skill, and to look for more general properties that apply to all chess games. Using these lessons, we can now derive our final strategy.

One such interesting and fundamental property of chess is the importance of "openings". Over its millennia-long history, players have determined that specific sequences of moves during the initial first few turns can have a profound impact on the later stages of the game.

Whether they allow you to position your pieces better or take your opponent's pieces more quickly, opening move sequences have become a critical part of every chess game. To become truly advanced at the game requires dedicated study of these openings, both how to play them against your opponent, and the correct moves to defend against them. Common openings include "The Queens Gambit" and "The Sicilian Defence", but there are over 1,327 uniquely named openings and more than 3,000 when including all variations [14] [24]. Given how ubiquitous chess openings are, this is precisely the type of chess-specific knowledge we can exploit within the final compression strategy.

Given that most games start with an opening and will normally follow it for a handful of moves, representing that entire sequence of moves with the corresponding Opening ID is nearly guaranteed to be more space-efficient than individually encoding each of those moves. For example, with roughly 3,000 openings sourced from Lichess's codebase, we would need 12 bits to assign each one a unique ID number [24]. Given that Lichess's current SOTA solution averages 4.4 bits-per-move, any game that has 3 or more moves belonging to an opening would be more efficiently compressed by replacing that sequence with the Opening ID. This is improved by the fact that most of those 3,000 openings are very uncommon, meaning that fewer than 12 bits would be needed. Therefore, determining exactly how many bits to use for this Opening ID code will require some experimentation.

Firstly, with the 3,000 openings obtained from Lichess's GitHub, we can scan through the Dec2017 database of over 16 million games and record how frequently each occurs [31]. We then sort these openings in descending order of frequency and configure our code to automatically fetch the first 2^n entries, where n is the length of the Opening ID in bits, specified as a constant variable near the top of the Rust module. These openings are then entered into a Trie data structure provided by the "trie-rs" crate [34]. A Trie (or prefix-tree) was chosen for this task as it excels at the efficient storage and lookup of strings with common prefixes [19]. Given that a chess opening is just a sequence of SAN moves, there is often a lot of overlap between variations. For example,

"1. e4 e5 2. Bc4" (i.e. the "Bishop's Opening") is only slightly different from "1. e4 e5 2. f4" (i.e. the "King's Gambit") [14]. Within a Trie, these openings would share nodes for their common prefix, yielding $O(n)$ search and space complexity [19].

We now use this Trie to identify openings at the start of a PGN's move-text section. If one is found, its n bit Opening ID is used to represent the sequence instead. However, we still need to account for situations where there is no opening, along with providing a way to signal to the decoder if it needs to treat the next n bits as an Opening ID and not the Huffman Coded move-text stream. So, using our experience of compressing the tag-pairs in the Bincode strategy, we will once again introduce a 1 bit-flag at the start of the move-text binary stream. In this case, 1 represents the fact that there is no opening detected and informs the decoder that all the following bits belong to the normal move-text stream. Inversely, a 0 informs the decoder that an opening was detected in the original PGN file, and to decode it, the next n bits need to be treated as an Opening ID. Upon extracting the Opening ID and uncovering the original opening, the decoder can then copy this SAN move sequence before continuing with the rest of the move-text stream as normal. A visualisation of this process can be seen in figure 17.

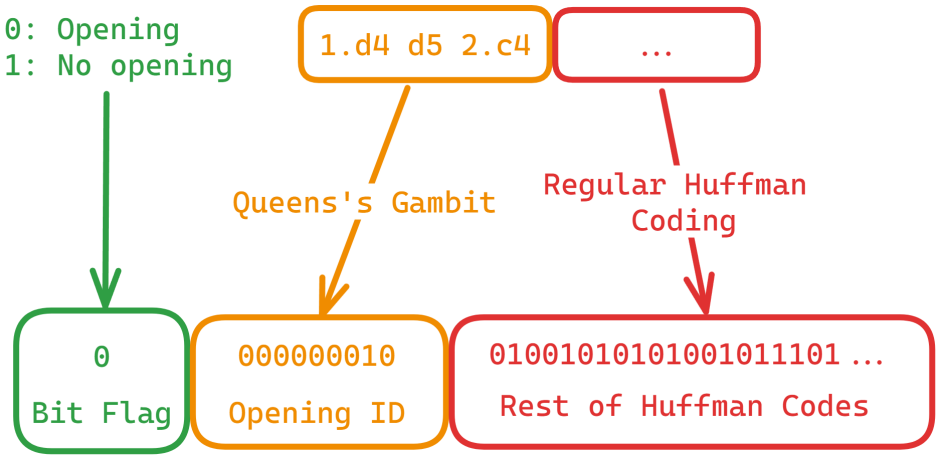

Figure 17: Example identification and encoding of an Opening ID

After running the strategy over a sample of PGN files, we can assert that 9 bits was the optimal length for the Opening ID. This allows us to encode up to 512 openings, with over 44% of games having an identifiable opening from this set. While this percentage may seem small, it is important to remember that not every game even contains a recognised opening and if we were to adopt a 10-bit Opening ID, we would not gain enough of a percentage share of games to offset the cost of an additional bit. In addition, we also experimented with imposing a minimum move length on these original 3,000 openings by removing any that fell below a given threshold. For instance, we know that any openings of 2 or fewer moves would take less than 8.8 bits (given Lichess's 4.4 bits-per-move solution as a rough estimate), so using a 9-bit Opening ID is wasteful in such situations. However, when imposing this minimum move length on the openings, we found no difference in the efficiency of the compression. Upon further inspection, there is no opening in the 512 that consists of 2 or fewer moves, which does make sense in retrospect. Building from the existing PgnHeaders compression and Huffman Coding techniques developed in previous strategies, implementing the rest of this strategy is straightforward. The final overview of it can be found in figure 18.

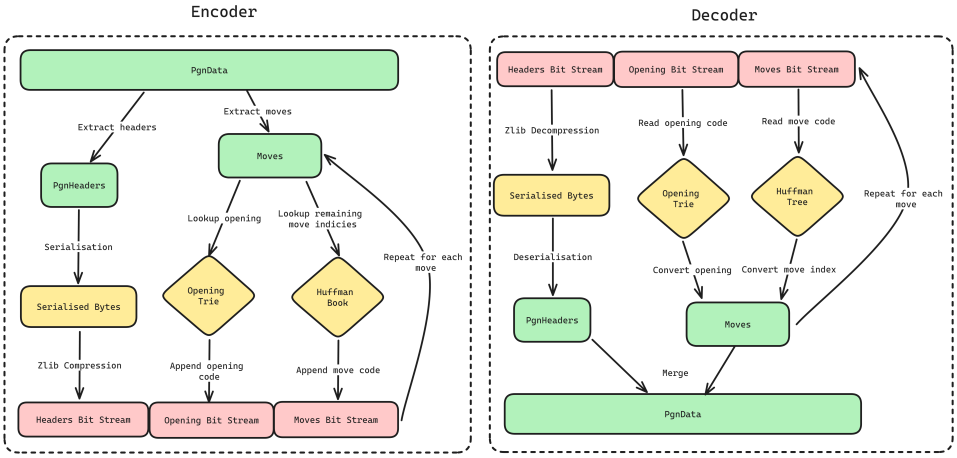

Figure 18: Opening Huffman encoder/decoder control flow

Compared to the Dynamic Huffman strategy, this new compression algorithm not only surpasses it in terms of compression efficiency but is

also significantly faster. It is almost as quick as the Huffman strategy, which is surprising given the additional work regarding the Trie and Opening ID detection. This helps demonstrate the importance of choosing your data structures correctly, and how careful experimentation during the development process can also assist with optimisation. The only potential downside identified is the 1-bit flag used for Opening ID detection, with it acting as additional overhead for games that contain no opening. However, on average and for the bulk compression of PGN files this 1-bit flag more than pays for itself. Hence, we are happy to present this strategy as our final solution to the PGN compression problem.

4.7 Evaluation

Due to using Test Driven Development as our primary development methodology, all the testing for this library took place during the implementation stages, ensuring that introducing additional functionality and strategies did not jeopardise any previous work. These tests were especially useful for refactoring parts of the Huffman strategy into the common utility module, including the "compress_headers" and "decompress_headers" functions. In total, there are 55 individual tests for this library, ranging from the unit testing of our strength index heuristic algorithm to larger integration tests of entire strategies. Additionally, by using an Agile approach to Sprint management, we effectively planned the development of each strategy by breaking it up into more manageable tasks (e.g. tag-pair compression, move-text Huffman Coding, or Opening ID detection) and assigning these to each week.

Although each strategy has already undergone some preliminary benchmarking within their respective sections, a full in-depth analysis and evaluation of the library's performance is still required. To accomplish this task, we develop our own benchmarking suite within the CLI subproject, whose exact implementation details can be found in section 5.2. Using this custom benchmarking tool, we can specify how many PGN games we want to benchmark the strategies upon, what PGN database file we should source these games from, and the output path

of the results file. Given our prior use of the Dec2017 PGN database, this 32GB file containing 16,232,215 games is used to benchmark our project. We also collect the following five statistics per PGN game: time to compress (in milliseconds), time to decompress (in milliseconds), compressed size (in bits), decompressed size (in bits), and bits-per-move. Running each strategy on the same 100,000 games and then averaging these five statistics provides us with a good foundation for evaluating these strategies, quantifying the effectiveness of each, and justifying their results. We limit ourselves to this number of games to ensure that the benchmarks do not take too long to run and so that others may replicate our results on similar consumer hardware.

4.7.1 Run Time Efficiency

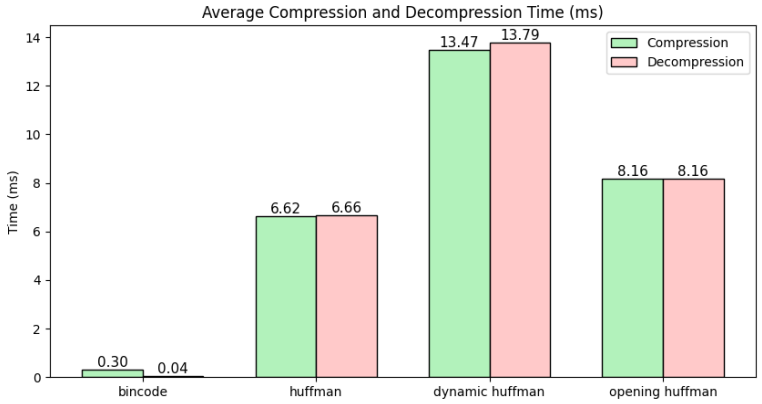

Figure 19: Average Compression and Decompression Time (ms)

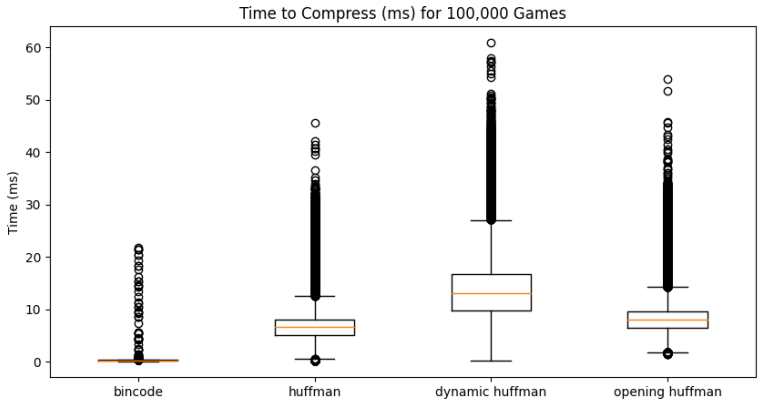

Figure 20: Time to Compress (ms) for 100,000 Games (Box Plot)

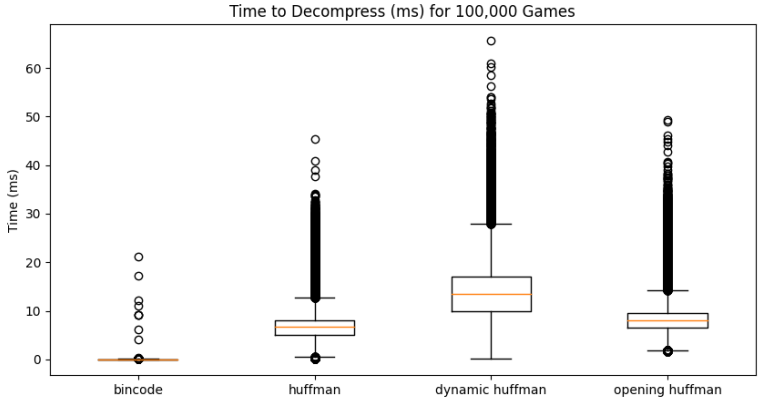

Figure 21: Time to Decompress (ms) for 100,000 Games (Box Plot)

Looking at figures 19, 20, and 21, we can see the Bincode strategy is the fastest, making it highly effective for real-time communication applications. The Huffman strategy results are also as expected, although it is interesting that the decompression step takes 0.04ms longer than the compression step. This is likely because it takes longer to decode (using a Huffman Tree) than encode (using a Huffman Book) since the first corresponds to a $O(n)$ tree traversal while the latter is just a $O(1)$ table lookup. The Dynamic Huffman strategy is much slower than all others and shows the largest deviation in its compression times. Hypothesising, this is probably due to the reconstruction of the Huffman Coding after every move, making the final compression time highly dependent on the number of moves within a PGN file. This also explains the 0.32ms slower average decompression, as the small decompression overhead observed in the Huffman strategy is further amplified in the Dynamic Huffman strategy. Finally, our most promising algorithm, Opening Huffman, is only 23% slower than the more basic Huffman strategy. This is a surprising result, as we originally assumed that the additional work regarding the Opening ID identification and encoding would add much more overhead to the strategy. However, this is not the case due to the efficient use of a Trie data structure, making the additional compression efficiency from the Opening Huffman strategy worth the slight increase in run time overhead.

4.7.2 Compression Ratio Efficiency

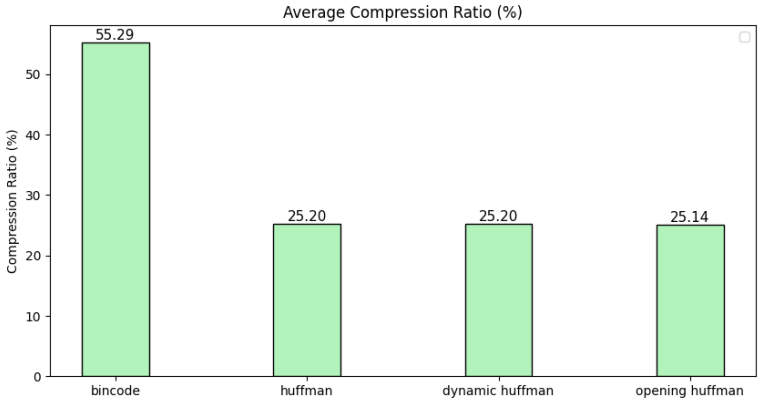

Figure 22: Average Compression Ratio Percent

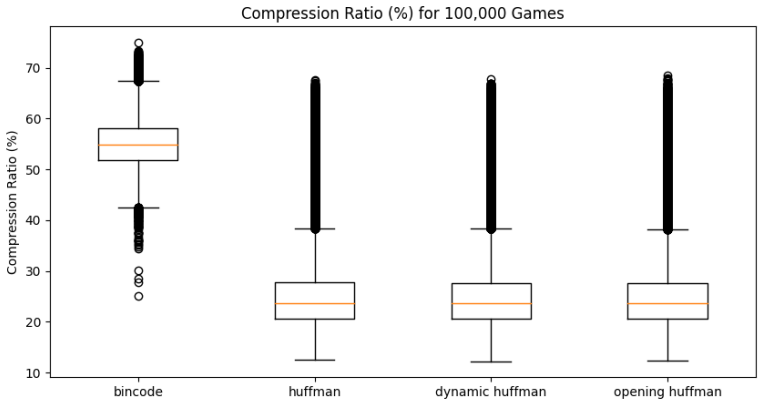

Figure 23: Compression Ratio Percent for 100,000 Games (Box Plot)

Evaluating the compression ratios, figures 22 and 23 clearly show the disparity between the Bincode strategy and the other Huffman Coding-based strategies. This is to be expected, as the Bincode strategy does not employ any chess-specific knowledge, providing less information for the encoder to exploit. However, a compression ratio of 55.29% is still impressive especially considering how quick the Bincode strategy is. The Huffman and Dynamic Huffman strategies have the same compression ratio of 25.20%, with Opening Huffman having a slightly lower 25.14%. This similarity is because they all compress the tag-pair section of a PGN file the same way, and this section forms much of the final binary stream. The move-text section contributes a smaller

portion to the binary stream, meaning that our improvements to the number of bits-per-move will not be too influential in lowering the overall compression ratios. This is not a significant drawback however, as a developer could employ their own database or storage solution for the tag-pair metadata, and still use our strategies to compress PGN files that have had their tag-pair sections removed. In this situation, where only the move-text is being compressed, there would be noticeable differences between the three Huffman Coding-based strategies as detailed in the following section.

4.7.3 Bits-per-move Efficiency

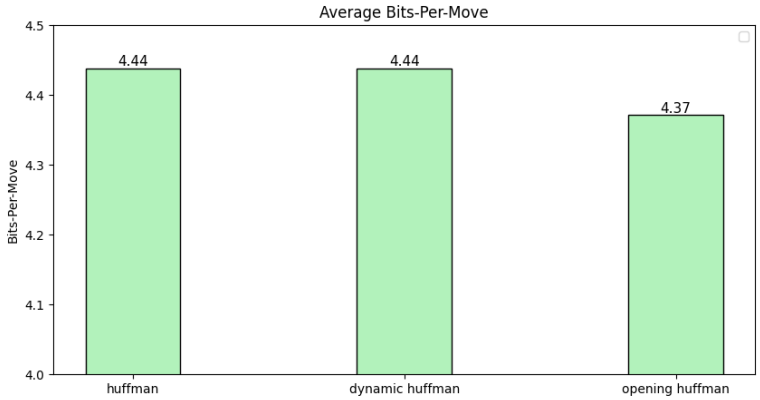

Figure 24: Average Bits-Per-Move

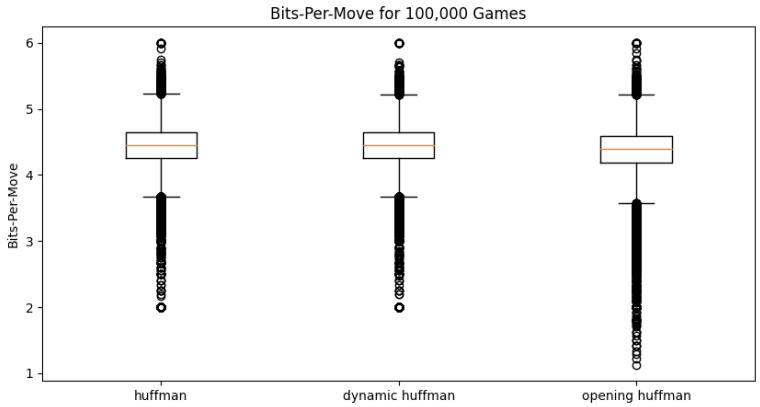

Figure 25: Bits-Per-Move for 100,000 Games (Box Plot)

It is important to note that within figures 24 and 25, we do not perform the same bits-per-move evaluation on the Bincode strategy, as there is not a clear distinction between the compression of the tag-pair and move-text sections like in the other strategies. Instead, this algorithm simply compresses the entire PgnData struct into one binary stream. However, if we ignore this nuance for the sake of comparison, this strategy achieved an average of 29.19 bits-per-move, computed by taking the entire binary stream and dividing it by the number of moves. For the other strategies, we can evaluate the move-text section in isolation, without the tag-pairs influence or the 1-bit flags used by the decoder. Within figure 24, we can see that the Huffman and Dynamic Huffman

strategies share the same 4.44 bits-per-move compression. This makes it clear that our Dynamic Huffman strategy is essentially a worse variation of our Huffman strategy, providing the same compression efficiency but at a significantly slower speed. However, the Opening Huffman strategy does much better, achieving an impressive 4.37 bits-per-move, which outperforms the 4.4 bits-per-move SOTA solution proposed by Lichess.

Finally, it is relevant to compare these bits-per-move results against the theoretical lower bound provided by the Shannon Entropy limit. Using the frequency distribution from our Huffman Coding-based strategies, and equation 2.2, we find that this lower bound is 4.28 bits-per-move. This means that the Huffman and Dynamic Huffman strategies are 3.7% higher than the entropy limit, but the Opening Huffman strategy is only 2.1% higher. Interestingly, none of the proposed strategies reach the entropy limit, likely because the strength indices we use are not perfectly "Independent and Identically Distributed" (IID) according to the approximated probability distribution. This condition is required for Huffman Coding to provide optimal compression, suggesting that other Entropy Codings could provide more flexibility and should be investigated for PGN compression [16].

5 Command Line Interface Implementation

5.1 Design and Development

Prior to the development of the main PGN compression library, there were no plans to create a command line interface (CLI) or any other type of front end for the library. However, as the scope of our project continues to expand, we require additional custom tools such as an efficient benchmarking suite to evaluate our strategies and a tuneable genetic algorithm for optimising the Dynamic Huffman strategy. Adhering to best practices, specifically Separation of Concerns, these tools need to be located outside of our Rust library since they are not used within the compression algorithms themselves. Additionally, since we require a method of passing configuration parameters to both the benchmarking suite and genetic algorithm, designing a simple CLI around them and the library itself emerges as a viable solution. Such a CLI could rely on our library as a dependency and expose its API to the various tools we need to execute, while also providing direct access to the underlying compression algorithms through "compress" and "decompress" terminal commands. The architecture overview of the CLI subproject can be found in figure 26, highlighting our library as a dependency alongside the benchmark and genetic algorithm modules.

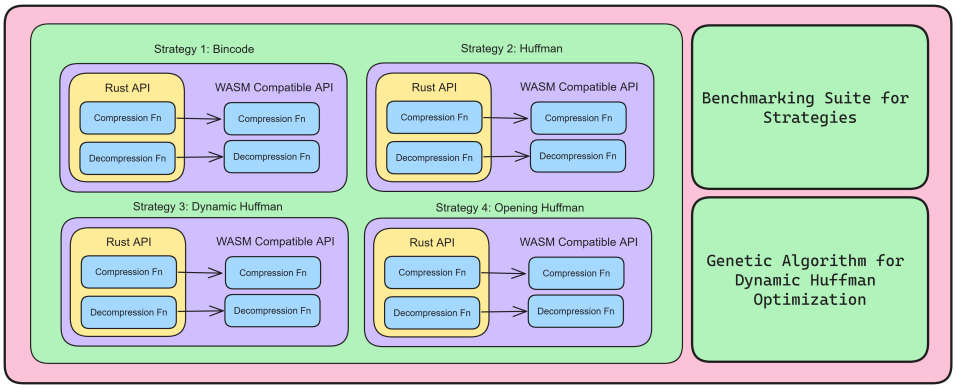

Figure 26: High-level architecture of the CLI

To interface with the underlying Rust modules of this subproject, we will develop our CLI using a Rust crate called "clap" [6]. This crate provides us with a command line argument parser that can be con-

figured in a declarative manner, making it easy to rapidly design a user-friendly CLI. Internally, "clap" uses Rust's powerful macro system to automatically generate much of the additional boilerplate code at compile time [6]. To get started, we first derive the parser macro on top of a custom Args struct that contains a command field. We also provide some additional details such as the name of the binary, the version, and the author, highlighted in listing 1.

Listing 1 Derivation of parser macro on Args struct

```
#[derive(Parser)]
#[clap(name="cgn-cli", #version="0.1.0", author="Jaden S")]
/// A command line interface for the cgn library
struct Args {
    #[clap(subcommand)]
    command: Commands,
}
```

After this, we need to declare the Commands enum and provide each command as a variant. Within these variants, we use a combination of macros and struct fields to specify the parameters that each command accepts, along with the parsing rules that should be followed. For example, for the Compress command in listing 2, we only want to accept optimisation levels between 0 and 3 (e.g. 0 for Bincode, 1 for Huffman, 2 for Dynamic Huffman, and 3 for Opening Huffman), so we specify a custom value parser that will throw an error if the optimisation level is outside that range. We then repeat this declarative process for each of the Decompress, Bench, and GenAlgo commands.

Listing 2 Argument specification and parsing for Compress command

```
#[derive(Subcommand)]
/// The commands for the command line interface
enum Commands {
    /// Compress a single PGN file
    Compress {
        /// Optimization level (0-3)
        #[clap(short, default_value = "3",
        value_parser = |s: &str| match s.parse::<u8>() {
            Ok(n) if n <= 3 => Ok(n),
            _ => Err(String::from(
                "Optimization level must be between 0 and 3")),
        })]
        optimization_level: u8,

        /// Input file path
        #[clap(value_parser)]
        input_path: String,

        /// Output file path
        #[clap(value_parser)]
        output_path: String,
    },
    ...
}
```

Along with making it easy to quickly set up our CLI, "clap" also automatically builds a *'-h'* help menu for each command using the triple-dashed comments above each field, significantly improving the user experience and overall quality of the CLI [6]. An example of this help menu can be found in figure 27. With the parser complete, we can now pattern match against the command line arguments and pass their values to the corresponding Rust module and function. Since the implementation of our PGN compression library has already been covered in section 4, we will focus on the development of specific benchmarking and genetic algorithm modules instead.

```
[~]$ cgn-cli bench -h
Benchmark the compression and decompression algorithms against a Lichess PGN database

Usage: cgn-cli bench <NUMBER_OF_GAMES> <INPUT_DB_PATH> [OUTPUT_PATH]

Arguments:
  <NUMBER_OF_GAMES>  Number of games to benchmark each algorithm on. If set to 'all',
then all games in the database will be used
  <INPUT_DB_PATH>    Input database path (Lichess PGN database format required)
  [OUTPUT_PATH]      Optional output file path for the benchmark results

Options:
  -h, --help  Print help
```

Figure 27: Example help menu for the Bench command

5.2 Benchmarking Suite

To benchmark our PGN compression strategies, we require a source of PGN games to run this evaluation upon. In this case, we choose to rely upon the Lichess Dec2017 PGN database, which we have used previously in our project and contains over 16 million real PGN games [31]. However, this database file is over 32GB in size, meaning that it is too large to load into memory all at once. Instead, we need a custom Iterator that can read these PGN games one at a time from the database file on the disk. Recalling from the official PGN specification, a PGN database is simply a plain-text file of multiple PGN games listed sequentially and separated by a new line character [22]. Therefore, to achieve this parsing functionality we declare a custom PgnDbIter struct and implement the Iterator Interface by providing it with a "next" method. Within this method, we repeatedly concatenate each line read from a PGN database (stored on the disk) until the end of a PGN game is detected, at which point we return the buffered PGN. Hence, we can now iterate over each game in a PGN database without fully loading it into memory.

Next, we need to collect the five chosen statistics required for evaluating our PGN strategies: time to compress (in milliseconds), time to decompress (in milliseconds), compressed size (in bits), decompressed size (in bits), and bits-per-move. We implement this functionality within the "collect_single_metric" function, which takes a PGN file path, and the compression/decompression functions we wish to use as its parameters.

Next, we perform the following steps to compute the five statistics and return them in the form of a custom Metrics struct where each statistic has a unique field:

1. Time how long it takes to compress the PGN file (time to compress)
2. Check how many bits the binary stream is (compressed size)
3. Time how long it takes to decompress the binary stream (time to decompress)
4. Check how many bits the decompressed PGN file is (decompressed size)
5. Compress the PGN file again, but this time with the tag-pairs removed. Divide the size of this binary stream by the number of moves (bits-per-move)

Now to benchmark each strategy, we implement a "collect_metrics" function which iterates over the given PGN database using the PgnDbIter and invokes the "collect_single_metric" function on each game. However, since this iteration is sequential, the original benchmarking process is very slow. To speed this up, we use the "rayon" crate to convert our normal PgnDbIter iterator into a parallel iterator with the "par_bridge" function [36]. Internally, this drains our iterator into a thread pool so that the benchmarking process can use all the available system threads in parallel. This greatly improves performance, and Rust's strong ownership system ensures that no data races occur [43]. After benchmarking over the total number of games specified, the "collect_metrics" function returns a list of all the Metrics structs. Finally, these Metrics are processed within the "metrics_to_summary" function, which simply takes the list of Metrics and computes the average value for each statistic. The averages for each strategy are then printed to the terminal and output file specified via the CLI.

5.3 Genetic Algorithm for Dynamic Huffman Optimisation

As a reminder, our Dynamic Huffman strategy uses a Gaussian Curve to update each player's move frequency distribution after every turn. To optimise for the lowest bits-per-move, we need to find the optimal height and deviation of this Gaussian Curve using a genetic algorithm, as mentioned in section 4.5. Within the Rust ecosystem, there are some pre-existing crates for genetic algorithm simulations, such as the aptly named "genetic_algorithm" crate [51]. However, the API of this crate is not very flexible nor well suited for our goals of optimising the continuous height and deviation parameters. For instance, this crate is limited to 32-bit floating point precision for continuous parameters, which is incompatible with our desire for 64-bit precision given the range and complexity of the search space [51]. Therefore, we believe it is easier to implement our own custom genetic algorithm in Rust.

In a new module, we first declare an Individual struct, which simply contains two 64-bit float fields named "height" and "deviation". We then declare a "fitness_function" that accepts a reference to an Individual struct and uses the "height" and "deviation" fields to run a small Dynamic Huffman benchmark with the "collect_metrics" function from section 5.2. The average bits-per-move of this mini-benchmark is then taken as the fitness score for that individual. Hence, to generate the initial population for our genetic algorithm we first generate a random collection of Individuals and calculate their fitness scores in parallel using the "rayon" crate [36]. For generating subsequent generations, we first select parents using a tournament process where the current generation is split into smaller subgroups and the fittest individual from each is selected. These parents are then randomly paired together and have their height and deviation values averaged to yield a new child Individual. After all the children undergo a small chance of mutation in either of their parameters, they are collected to form the new generation. This process then repeats, eventually yielding Individuals and parameters that produce better results within the Dynamic Huffman strategy.

To grant us additional flexibility, every property of our genetic algo-

rithm can be easily configured and passed as CLI arguments. This includes the initial population size, the number of games to benchmark an Individual against, the number of generations to run for, the child mutation rate, the tournament group size, and the minimum/maximum values allowed for the height and deviation parameters. To aid in visualising these results, we print each generation of Individuals into a plain-text file for that simulation run, which can then be parsed and displayed within a separate Jupyter notebook. This ability to visualise each simulation run is critical, as it allows us to readjust the minimum/maximum parameter range for each run and focus on promising areas of the state space with greater precision. Examples of these visualisations can be seen in figures 28 to 33, with yellow and blue corresponding to that run's newest and oldest generations, respectively.

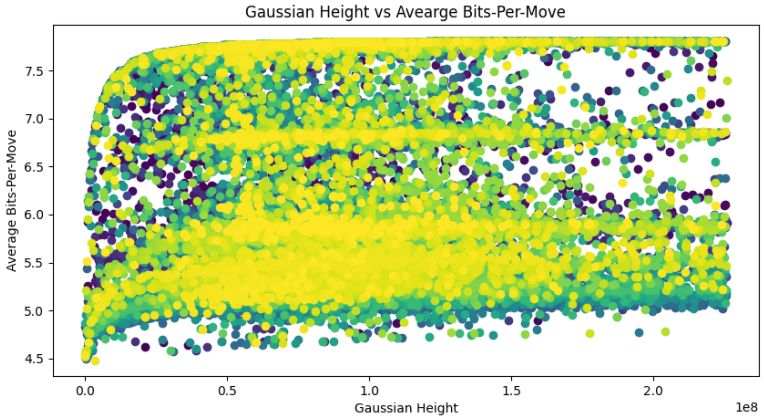

Figure 28: Gaussian Height vs Bits-Per-Move from 1st run

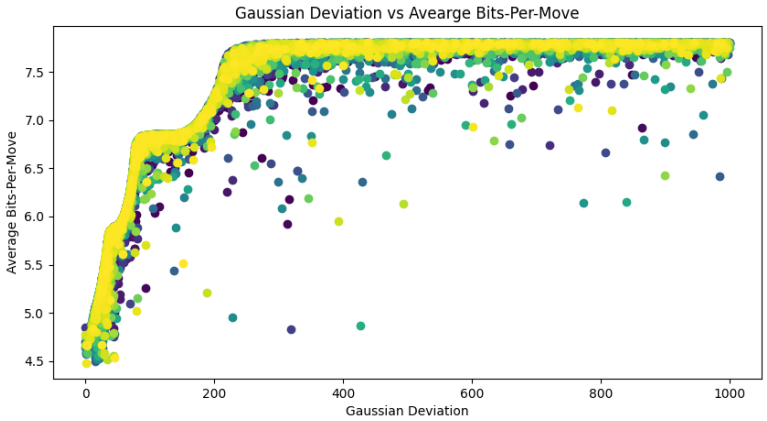

Figure 29: Gaussian Deviation vs Bits-Per-Move from 1st run

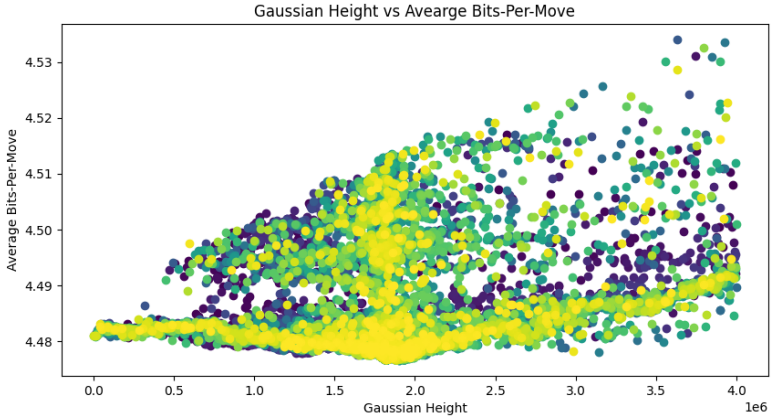

Figure 30: Gaussian Height vs Bits-Per-Move from 9th run

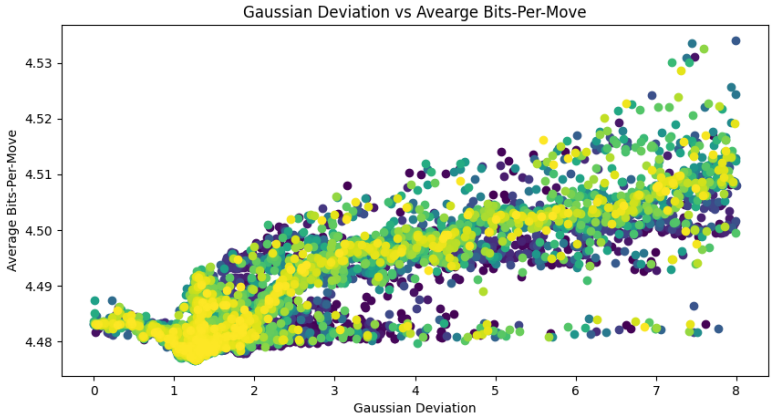

Figure 31: Gaussian Deviation vs Bits-Per-Move from 9th run

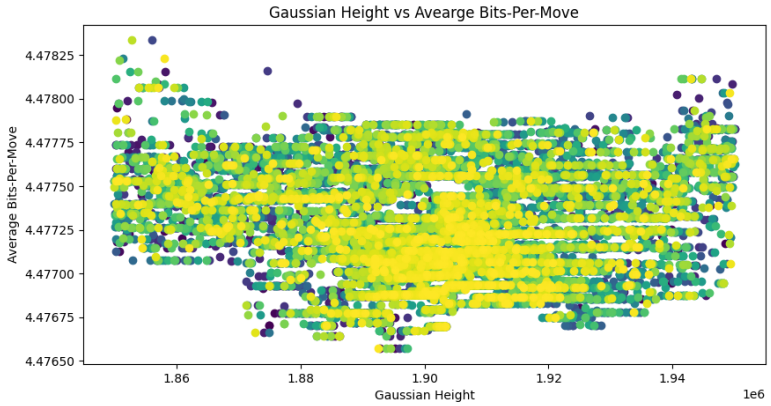

Figure 32: Gaussian Height vs Bits-Per-Move from 17th run

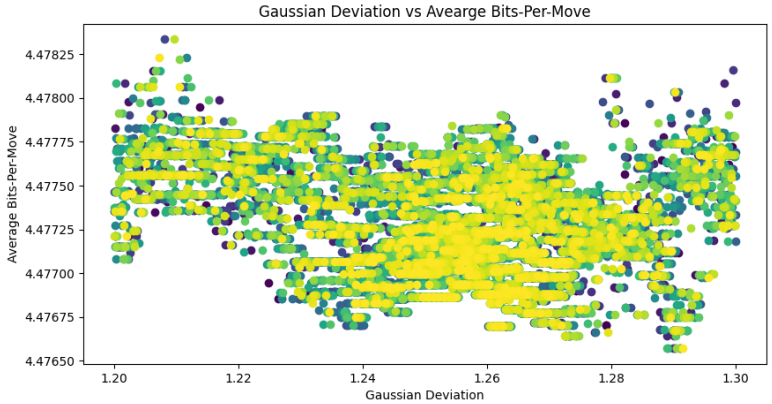

Figure 33: Gaussian Deviation vs Bits-Per-Move from 17th run

We repeat this process of simulating and readjusting parameter ranges for a total of 17 genetic algorithm runs, with each run containing hundreds of generations. However, on the 17th run of the genetic algorithm, we start to notice that the average bits-per-move of our individuals are settling into discrete horizontal levels and that it is becoming exceedingly difficult for new optimal individuals to emerge. This is a sign that we are reaching a precision limit and that continuing to shrink the parameter range is unlikely to yield significant improvements. For this reason, we terminate our search here and use the best Individual parameters of $height = 1,895,192.82$ and $deviation = 1.29$ within our Dynamic Huffman strategy.

5.4 Evaluation

Despite not originally being planned for development, this CLI is now an extremely useful tool in our project portfolio as it provides an interface for demonstrating the PGN compression library to interested parties. This should help promote the wider adoption of our compression strategies and allow users to experiment with the library before committing to a full integration. Additionally, by adhering to Separation of Concerns, we ensure both the library and CLI ship with the minimal set of dependencies required by their unique functionality. For instance, if we built the CLI within the same Rust project as the library, we would have to include additional crates such as "clap" which are not required by the compression functions themselves. Users would then have to download this additional dependency when using the library, regardless of whether they want to use the CLI or not. By creating two separate Rust projects, users can now choose the exact code they want to use, whether it be the main library or the additional CLI wrapper.

As for the benchmarking and genetic algorithm, both these modules have been designed in an efficient and configurable manner to improve their utility to users of the CLI. For example, if the user wants to benchmark the strategies against a different PGN database or number of games, this can be facilitated by passing different command line arguments to the CLI. Likewise, every aspect of the genetic algorithm can be configured without making changes to the source code, allowing others to replicate our Dynamic Huffman optimisation process or make additional improvements. Overall, the implementation of this CLI subproject was critical to the success and evaluation of our universal PGN compression library, and a beneficial proof-of-concept illustrating how our API may be integrated into other applications and projects.

6 Web Browser Extension Implementation

6.1 Design and Development

Having completed the library and CLI, we can now turn our attention to compiling the PGN compression strategies to a WASM module to fulfill our objective of creating a universal library. To do so, we will develop a small proof-of-concept subproject outside of the Rust ecosystem that uses this WASM module. This will help demonstrate the universal nature of our work, and how the library can be easily integrated across multiple platforms and languages. Given WASM's support by web browsers, we have chosen to implement a simple browser extension that can use the Opening Huffman strategy to compress or decompress PGN files. It is worth noting that while the utility of this WASM module could have been highlighted in a wide range of other languages (e.g. Python, C/C++, .NET, Go, and Java), building a JavaScript-based browser extension allows us to use other familiar web technologies like HTML and CSS to rapidly build a UI for the application, and distribute it to non-technical users via the Chrome Extension store if desired [52].

To start this process, we first need to compile the Rust library into WASM. Due to Rust's excellent support of WASM as a compilation target, we can either directly use Rust's "cargo build" command, or a command line tool such as "wasm-pack", which also generates the necessary JavaScript files for importing the WASM module into the browser [8]. However, as mentioned in section 4.1, there are some restrictions on what functions within our library can be compiled using "wasm-pack". Primarily, functions exported to WASM must only use primitive types in their type signature, as these are then converted by "wasm-pack" to the corresponding WASM primitive types at compile time [8]. This means that functions that accept or return structs cannot be exported to WASM since "wasm-pack" is unable to convert these structs into a WASM primitive type. To avoid this issue, we can declare a new Rust function that only uses primitive types, but then builds the required struct and calls the original Rust function from within its

body. This works well since the WASM function body does not need to exclusively use primitive types and is free to use the stack or heap as usual.

The compression and decompression functions of the four strategies use `&PgnData -> Result<BitVec>` and `&BitVec -> Result<PgnData>` as their respective type signatures. While the primitive versions of these functions could be handwritten, given their identical signature across all strategies a custom Rust macro can be used instead to automatically generate the primitive versions at compile time. As such, we declare a custom "export_to_wasm" macro, which takes a pair of compression/decompression functions, and wraps them in the primitive type signatures of `&str -> Vec<u8>` and `&[u8] -> String`. In addition, each of these newly generated functions is marked with a "wasm_bindgen" macro, so that the "wasm-pack" tool can identify what functions to compile into the final WASM module [8]. After writing some unit tests to ensure that the primitive versions demonstrate the same results as their original counterparts, we compile to WASM using "wasm-pack" and the *'-web'* command line flag to get the required WASM and JavaScript files for the browser extension [8].

Following Google's online documentation on creating a browser extension, we first create a "manifest.json" file for declaring the name, version, and permissions of our extension, along with the necessary HTML, CSS, and JavaScript files [28]. The UI is split into two sections as seen in figure 34, one for compressing PGN files and the other for decompressing. To compress, the user can enter a PGN string manually or drag-and-drop a *'.pgn'* file onto the textbox. They then can copy the compressed bytes (represented by a shorter hexadecimal string) to their clipboard or download a *'.cgn'* file. This custom file format, an abbreviation of "Compressed Game Notation", simply contains the compressed PGN bytes. Likewise for decompression, the user can enter a hexadecimal string or drag-and-drop a *'.cgn'* file onto the textbox. The original PGN string can then be copied back to the user's clipboard or downloaded as a standard *'.pgn'* file.

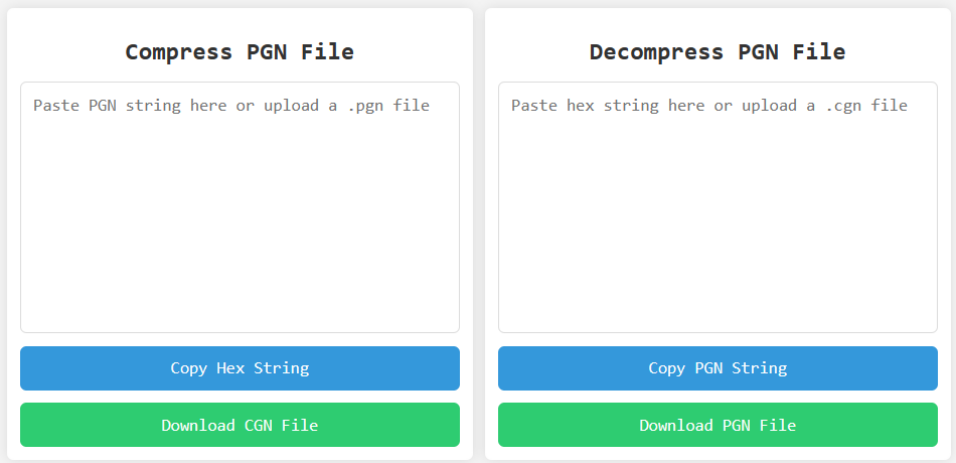

Figure 34: User interface of the Web Browser Extension

To implement this, within the "index.js" file we import the Opening Huffman strategy from the WASM module and attach the corresponding compression and decompression functions to the on-click handlers of the buttons, taking the content of the relevant textbox as input. These handlers are then wrapped in "try-catch" blocks to ensure that any errors from within the WASM module are caught and that the user is notified by changing the corresponding button text. Finally, these files are zipped together and imported into a Google Chrome browser to verify that the extension works as expected and produces the same results as our original PGN compression library.

6.2 Evaluation

This subproject serves as a great proof-of-concept for how our universal PGN compression library can be distributed and run on a multitude of different devices, despite being originally written in Rust. The browser extension also makes few sacrifices when it comes to runtime efficiency, executing the Opening Huffman strategy natively within the browser at nearly identical speed to its Rust equivalent. This suggests that similar client-side integrations of our library by Chess.com and Lichess.org could save them considerable amounts of server-side resources by leveraging the power of the user's device instead. Moreover, there are additional use cases for this browser extension, mostly

regarding the sharing and communication of PGN files online. For instance, at the time of writing, Twitter's tweets are limited to only 280 characters for free accounts. The content of most PGN files exceeds this limit, meaning that users must resort to other external methods to share their chess games. However, by using this browser extension, users can compress their PGN content to a shorter hexadecimal string that will fit within the character limit. An example tweet sharing the move-text section of the PGN file in figure 2 can be seen in figure 35.

Figure 35: Example tweet sharing a compressed game in under 280 characters

7 Conclusions

Out of the four PGN compression strategies developed within this project, the Opening Huffman strategy provides the best compression ratios and the lowest number of bits-per-move. In fact, this strategy represents a new SOTA solution within the field, achieving an average of 4.37 bits-per-move in comparison to Lichess's 4.4 bits-per-move [37]. Moreover, there is reason to believe that Lichess rounded down their result from the true value of 4.44 bits-per-move since this is the outcome of our Huffman strategy and represents a direct implementation of Lichess's approach in Rust. In this case, our Opening Huffman strategy produces 1.6% fewer bits-per-move than the current SOTA, a meaningful improvement when it comes to the bulk compression of PGN files. For example, given that Chess.com generates approximately 10TB of uncompressed PGN move-text data annually, adopting our strategy would yield an additional 160GB of free space if used instead of Lichess's approach. This is a considerable amount of savings, improved by the fact that the Opening Huffman strategy is only 2ms slower than the Huffman strategy. Therefore, given that we have beaten the previous SOTA and demonstrated the language-agnostic utility of our library in section 6, our original objectives for this project have been successfully met.

In addition to meeting these goals, we mentioned in section 3 that we hope to see some natural usage of our library and tools by the wider chess community. This would not only add credibility to the technical quality of our library but also show that our additional efforts into having 97% test coverage and 77% documentation coverage are appreciated by like-minded developers. To verify this, on February 13th 2024, our library and CLI were published to Rust's "crates.io" repository under the names "cgn" and "cgn-cli" respectively. These names are an abbreviation of "Compressed Game Notation", a moniker used through this project and inspired by the similarly named "Portable Game Notation". A JavaScript-compatible version of the library named "cgn-js" which uses WASM was also uploaded to the "npmjs.com" repository on the same date. Since then, our code has seen a total of 2214 downloads,

split between 1304 downloads for "cgn", 742 for "cgn-cli", and 168 for "cgn-js" [39] [40] [41]. Given this rapid accumulation of downloads with no marketing or promotion, this suggests that there is a sizeable demand for PGN compression tools and that our project satisfies this niche with its performance and quality.

In terms of the limitations of this project, it is disappointing that our solution cannot be quantitatively compared against Chess.com's implementation of PGN compression, given that they are closed source and disclose no details about their internal infrastructure. However, provided that the Opening Huffman strategy surpasses all publicly known PGN compression algorithms, it is probable that it also outperforms Chess.com as well. A further limitation to consider, particularly from a technical perspective, is the undesirable inefficiency of the Dynamic Huffman strategy. While it performed on par with the Huffman strategy in terms of compression ratio and bits-per-move, it was more than twice as slow for both compressing and decompressing PGN files. While this was not the desired result, it illustrated that attempting to incorporate personalised information (such as player's skill) into our strategies was not a viable approach and that more generic properties (such as chess openings) should be explored. Hence, it is due to the inferior performance of the Dynamic Huffman strategy that we were motivated to search for more effective solutions to this problem, and eventually derive the Opening Huffman strategy that proved to be only 2.1% away from the theoretically optimal 4.28 bits-per-move.

8 Future Work

While this project is a comprehensive investigation into the topic of PGN file compression, there are of course other aspects that could not be covered. One such idea is related to the discovery that the Huffman strategy achieves an average of 4.44 bits-per-move, while the theoretical entropy limit is only 4.28. Given that this entropy limit was calculated by using the approximated probability distribution from the Huffman strategy itself, it is expected that this strategy would approach 4.28 bits-per-move more closely. The fact that the Huffman Coding does not reach this optimal limit suggests that the move strength indices are not truly "Independent and Identically Distributed" (IID) as required [16]. In other words, it appears that the strength of one move can influence the strength of later moves. For example, say your opponent plays strong moves in the early turns of a match and captures many of your pieces. Since you now have a smaller set of legal moves to select from, it becomes impossible to play a move with a large strength index (e.g. with 10 legal moves, the maximum playable strength index is 9). Therefore, as you lose pieces during the chess match and your set of legal moves decreases, you must inherently play moves with smaller strength indices. This introduces a relationship between past and present strength indices, which violates the IID condition required for Huffman Coding to produce optimal encodings [16]. Because of this, future research could be conducted into the feasibility of other Entropy Codings, such as Arithmetic Coding and Asymmetric Numeral Systems [16]. These techniques could provide more flexibility and approach the 4.28 entropy limit without the IID condition for optimality.

Alongside research-intensive work, there are also more practical and software-based projects that could be developed. Firstly, there is significant potential for the library's WASM module to be used within other languages and published in their respective package repositories. Provided that a JavaScript-compatible implementation is already available via the "cgn-js" package, Python would be another good language to support. This would bring our project to millions more developers and enhance its credibility as a universal library for PGN file compres-

sion. Finally, there is also the potential for better PGN tag-pair management solutions. For instance, instead of including the compressed tag-pairs in the final binary stream, a local key-value database could be used. Pointers could then be included within the compressed PGN file to identify the correct tag-pairs within the database, reducing the amount of data duplication as each unique tag-pair would only need to be stored once to be referenced across many compressed PGN files. This idea can be further expanded into a domain-specific database application for PGN files, allowing for more rapid access and efficient querying for online chess platforms.

References

[1] R. C. Academy. Unveiling the average number of moves in a chess game. https://chess-teacher.com/the-average-number-of-moves/, August 2023.

[2] S. Akinyemi. Awesome webassembly languages. https://github.com/appcypher/awesome-wasm-langs?tab=readme-ov-file, January 2024.

[3] Baeldung. Simulated annealing explained. https://www.baeldung.com/cs/simulated-annealing, March 2024.

[4] Chess.com. Chess.com reaches 100 million members! https://www.chess.com/article/view/chesscom-reaches-100-million-members, December 2022.

[5] Chess.com. Chess is booming! and our servers are struggling. https://www.chess.com/blog/CHESScom/chess-is-booming-and-our-servers-are-struggling, January 2023.

[6] clap rs. clap-rs/clap. https://github.com/clap-rs/clap, April 2024.

[7] C. P. W. Contributors. Steven edwards. https://www.chessprogramming.org/index.php?title=Steven_Edwards&oldid=22309, November 2020.

[8] M. Contributors. Webassembly. https://developer.mozilla.org/en-US/docs/WebAssembly, April 2024.

[9] R. U. Contributors. Functional usage of rust. https://rust-unofficial.github.io/patterns/functional/index.html, March 2024.

[10] W. Contributors. Bitboard. https://en.wikipedia.org/w/index.php?title=Bitboard&oldid=1182200881, October 2023.

[11] W. Contributors. Lichess. https://en.wikipedia.org/w/index.php?title=Lichess&oldid=1187623082, November 2023.

[12] W. Contributors. Portable game notation. https://en.wikipedia.org/w/index.php?title=Portable_Game_Notation&oldid=1178763636, October 2023.

[13] W. Contributors. Algebraic notation (chess). https://en.wikipedia.org/w/index.php?title=Algebraic_notation_(chess)&oldid=1215300450, March 2024.

[14] W. Contributors. Chess opening. https://en.wikipedia.org/w/index.php?title=Chess_opening&oldid=1217697349, April 2024.

[15] W. Contributors. Gaussian function. https://en.wikipedia.org/w/index.php?title=Gaussian_function&oldid=1215049701, March 2024.

[16] W. Contributors. Huffman coding. https://en.wikipedia.org/w/index.php?title=Huffman_coding&oldid=1209911693, February 2024.

[17] W. Contributors. Information theory. https://en.wikipedia.org/w/index.php?title=Information_theory&oldid=1216573942, March 2024.

[18] W. Contributors. Rust (programming language). https://en.wikipedia.org/w/index.php?title=Rust_(programming_language)&oldid=1218037672, April 2024.

[19] W. Contributors. Trie. https://en.wikipedia.org/w/index.php?title=Trie&oldid=1197148112, January 2024.

[20] W. Contributors. Zlib. https://en.wikipedia.org/w/index.php?title=Zlib&oldid=1216098790, March 2024.

[21] A. Crichton, S. Thiel, and J. Triplett. rust-lang/flate2-rs. https://github.com/rust-lang/flate2-rs, March 2024.

[22] S. J. Edwards. Standard: Portable game notation specification and implementation guide. http://www.saremba.de/chessgml/standards/pgn/pgn-complete.htm, 1994.

[23] N. Fiekas. niklasf/rust-huffman-compress. https://github.com/niklasf/rust-huffman-compress, December 2023.

[24] N. Fiekas. lichess-org/chess-openings. https://github.com/lichess-org/chess-openings, February 2024.

[25] N. Fiekas. niklasf/rust-pgn-reader. https://github.com/niklasf/rust-pgn-reader, April 2024.

[26] N. Fiekas. niklasf/shakmaty. https://github.com/niklasf/shakmaty, April 2024.

[27] GeeksForGeeks. Geneticalgorithms. https://www.geeksforgeeks.org/genetic-algorithms/, March 2024.

[28] Google. Hello world extension. https://developer.chrome.com/docs/extensions/get-started/tutorial/hello-world, October 2022.

[29] R. H. Inc. What is ci/cd? https://www.redhat.com/en/topics/devops/what-is-ci-cd, December 2023.

[30] J. Levine. Hi reddit! i'm josh levine, cto at chess.com. ama! https://www.reddit.com/r/chess/comments/111940a/hi_reddit_im_josh_levine_cto_at_chesscom_ama_10am/, February 2023.

[31] Lichess. Lichess.org open database. https://database.lichess.org/, March 2024.

[32] D. Moshkovitz and B. Tidor. Lecture 19: Compression and huffman coding. https://ocw.mit.edu/courses/6-046j-design-and-analysis-of-algorithms-spring-2012/388115265a456321c4a5d19dc9e05281_MIT6_046JS12_lec19.pdf, 2012.

[33] N. Naidu. Agile software development – software engineering. https://www.geeksforgeeks.org/software-engineering-agile-software-development/, March 2024.

[34] S. Nakatani and S. Celis. laysakura/trie-rs. https://github.com/laysakura/trie-rs, April 2024.

[35] T. Overby, Z. Riordan, and Trangar. bincode-org/bincode. https://github.com/bincode-org/bincode, March 2024.

[36] rayon rs. rayon-rs/rayon. https://github.com/rayon-rs/rayon, April 2024.

[37] revoof. Developer update: 275 https://lichess.org/blog/Wqa7GiAAAOIpBLoY/developer-update-275-improved-game-compression, March 2018.

[38] J. V. Stone. Information theory: A tutorial introduction. https://arxiv.org/pdf/1802.05968.pdf, June 2019.

[39] J. Strudwick. cgn. https://crates.io/crates/cgn, April 2024.

[40] J. Strudwick. cgn-cli. https://crates.io/crates/cgn-cli, April 2024.

[41] J. Strudwick. cgn-js. https://www.npmjs.com/package/cgn-js, February 2024.

[42] subraman. Time and space complexity of huffman coding algorithm. https://www.geeksforgeeks.org/time-and-space-complexity-of-huffman-coding-algorithm/, February 2024.

[43] R. Team. Data races and race conditions. https://doc.rust-lang.org/nomicon/races.html, April 2024.

[44] R. Team. Modules. https://doc.rust-lang.org/rust-by-example/mod.html, April 2024.

[45] R. Team. Tests. https://doc.rust-lang.org/cargo/guide/tests.html, April 2024.

[46] R. Team. Vectors. https://doc.rust-lang.org/stable/rust-by-example/std/vec.html, April 2024.

[47] R. Team. Webassembly. https://www.rust-lang.org/what/wasm, 2024.

[48] J. Unadkat. What is test driven development (tdd)? https://www.browserstack.com/guide/what-is-test-driven-development, June 2023.

[49] L. Watson. Chess boom hits new heights with 1 billion games played in february. https://www.chess.com/news/view/chess-boom-1-billion-games-played-in-february, March 2023.

[50] M. Weeks. The rise of internet chess. https://www.mark-weeks.com/aboutcom/aa07e19.htm, May 2007.

[51] B. v. Westing. basvanwesting/genetic-algorithm. https://github.com/basvanwesting/genetic-algorithm, August 2023.

[52] A. Williams. Webassembly runtimes compared. https://blog.logrocket.com/webassembly-runtimes-compared/, April 2021.