A Genetic Algorithm for Chess with a Custom Engine

1st Matthew Belanger
Master of Science
University of Windsor
Windsor, ON, Canada
belangeq@uwindsor.ca

2nd Vlad Tusinean
PhD Student - Computer Science
University of Windsor
Windsor, ON, Canada
tusineav@uwindsor.ca

***Abstract*—This project introduces a chess engine and supporting interface developed entirely in C++, emphasizing performance, modularity, and extensibility. The system is built from the ground up to support both deterministic gameplay and AI-driven experimentation. At its core is a fully functional chess engine capable of enforcing the complete rules of chess. The engine is exposed through a lightweight RESTful API that enables seamless interaction with external clients. The API provides endpoints for retrieving the current board state, querying all legal moves for a given position, submitting moves to the engine, and retrieving general game status. Inputs and outputs are encoded as compact, unseparated strings embedded directly into the request paths, minimizing overhead and simplifying client-side parsing logic.**

A key feature of the project is the inclusion of a genetic algorithm, designed to evolve an individual's chess position evaluation strategies by simulating games and optimizing performance-based fitness criteria. This approach separates itself from other works in that fitness evaluation is based on a chromosome's ability to play an entire game, rather than playing a given position correctly. Our results provide a justification for other approaches, and inform future research in adaptive and self-improving AI behaviour. The modular design of the engine allows the approach to be modified or extended with alternative heuristics or learning techniques without altering core gameplay logic. Algorithms that were used to implement the genetic algorithm are included in this paper.

To support usability and rapid prototyping, we provide a well-documented C++ codebase along with example programs demonstrating how to use the library, interact with the API, and integrate AI agents. This includes sample clients capable of issuing move requests, visualizing engine responses, and benchmarking different configurations. The system has been engineered to support integration into larger applications, whether for interactive gameplay, reinforcement learning environments, or algorithmic competitions. Overall, the project delivers a versatile and efficient foundation for chess engine development, AI experimentation, and educational engagement with game AI and systems programming. Codebase can be found at https://github.com/MattBelanger321/COMP8120-Chess.

I. Introduction

The game of chess dates back to at least 1,500 years ago. It is a complex strategy game that has philosophical implications and is generally seen as a game requiring deep study and enormous effort to reach peak performance. However, with the proliferation of Artificial Intelligence in recent decades, human players are no longer at the pinnacle of chess play. There is an interesting split where competitions exist to determine the best chess AI, leaving human-based competitions in their own category. Nevertheless, chess AI has not been solely destructive on the sport: chess players leverage AI engines to better study the game, and current top players are arguably the best they've ever been due in part to accessibility, and the fact that even the best player in the world can be decisively beaten, allowing them to study the game even further. All sorts of chess AI has been proposed and tested, and now modern-day chess engines generally leverage machine learning to conduct their play. However, there is also an interesting niche of engine development using evolutionary algorithms to attempt to evolve an agent in a similar manner to how humans learn to play the game. Genetic Algorithms (GAs) for chess have been studied for over a decade, and while they do not perform at the level of modern chess engines such as Stockfish [1], some are capable of beating chess masters.

The goal of this work was to implement a GA for chess in a different manner than was described in the literature. This approach was not intended to outperform prior GA approaches or modern chess engines, but simply to see how a different methodology for evolution would perform, and analyze its limitations. As the work for this course was meant to be exploratory, intending for us to research topics we had never explored before, we chose to create not only a GA approach for chess, but also our own engine from scratch.

GAs require enormous processing time, depending on the implementation, and this requires (1) developing an efficient methodology, and (2) developing an efficient engine. We decided that it would be ideal to create a chess engine in C++ (as it is generally an extremely fast language) that was capable of leveraging computation power across a distributed system to accelerate the learning process. The principle behind the approach is that multiple devices could connect to a centralized server and participate in the evolution process. GA evolution involves evaluating the fitness of individuals in (ideally) a large population across a large number of generations. This chess application could leverage client devices to handle this fitness evaluation which would drastically decrease the time between generations. The advantage of leveraging other devices is that we would no longer be limited to just one CPU,

we would have theoretically limitless processing capabilities, bottlenecked only by the I/O to the server device.

There are two key aspects of a GA for chess: the search function and the evaluation function. We focused our efforts on the evaluation function, evolving individuals that assign scores to chess positions. However, unlike other works, we prioritized forcing individuals to play full games of chess against a strong, albeit forcefully limited, chess engine.

II. Literature Review

To our knowledge, there are five works covering GAs for chess in some form, starting as early as 2005 and as late as 2014. It is worth noting here that while there is a wide breadth of works covering AI approaches to chess, there are very few works on GAs for chess due to the relative limitations in performance as chess is a very complex game. In 2005, [2] explored an evolutionary approach using genetic programming to evolve an agent that was capable of playing endgame positions, which are positions in chess with a low number of overall pieces (such as a rook-and-pawn endgame, which sees a small number of pawns, one or both rooks, and the king for both players). Genetic programming is different from genetic algorithms; rather than each individual being a chromosome (such as a vector of weights), each individual is a program. In this work, these programs are represented as expression trees, where they represent a different chess strategy. The functions in the program are internal nodes (such as boolean AND, OR, NOT, and IF functions), and leaf nodes are evaluations of different features of the chess position (such as the number of pieces attacking another piece). Similar to other genetic algorithms for chess, this approach also iterates over all possible moves and returns the move resulting in the best position according to its scoring system (defined by the program). After evolution, the best individual from this approach was able to make a draw against the second-best chess engine at the time. A key 2008 study [3] attempted to reverse-engineer the evaluation function of a strong, proprietary chess engine using a genetic algorithm. A common issue with chess engines is that the way their reach their position evaluation and subsequent next move is proprietary. Thus, they serve as a ”black box” where decisions are entirely obfuscated and deducing how they came about their decisions becomes nearly impossible. This work was an attempt to see if an individual could be evolved to achieve identical performance to an obfuscated chess engine. The chess engine served as a ”mentor” to individuals in the population, where an individual's fitness was determined based on how close their position evaluation is to the mentor's position evaluation. From random initialization, the mentor engine was able to win 66.3% of games against the mentee, and after the evolution process, it was only able to win 50.5% of games, seeing a significant improvement and a justification of the evolutionary approach. This same group tried a different approach in 2010, where they discussed a methodology for automatically tuning the search parameters of a chess program using an evolutionary approach [4]. Prior to this work, there were a number of search functions used to determine the next best move of a chess engine, and these functions used manually tuned parameters that were considered optimal, determined after years of experiments and manual optimizations. [4] aimed to improve these manually tuned parameters by evolving individuals based on their performance on a dataset of 879 tactical test positions, where they were tasked to find a predetermined ”correct move”. However, the fitness of the individual was not marked by how many times they found the correct move in those 879 positions, but rather if they were able to find the correct move within 500,000 position evaluations, which incentivized the individuals to find the correct move as fast as possible. The resulting performance of these optimized selective searches was comparable to that of the manually tuned parameters of the top chess programs at the time. Extending their work, [5] attempted to also evolve a grandmaster-level evaluation function on top of their search function. Here, they used a dataset of 5,000 chess games from grandmasters (human players), and randomly picked one position from each game. Individuals in the population were evaluated based on whether or not they picked the grandmaster's next move for a given position (which was treated as the correct answer), and fitness was the number of correct moves determined across all 5,000 test positions. They ran 10 instances of the evolutionary algorithm, resulting in 10 individuals that have a slightly different playstyle (due to the random initialization), and these 10 individuals coevolved for an additional 50 generations to attempt to produce an even stronger agent. After the agent was evolved, they evolved their search function in a similar manner as their previous work [], and the result of their approach was an individual that could outperform the best chess engine at the time, which was a monumental increase in performance as this work saw an additional four years of development of the other chess engines. Evolving both a search function *and* an evaluation function overcame assumptions from their prior works which assumed that at least one of the two was *a priori* optimal. Finally, [6] had a similar core approach as [], utilizing a database of 400 tactical chess problems and 400 positional chess problems to evaluate the population. The fitness of the individual was based on whether or not they found the correct move in *all* chess problems, and individuals were mutated if they could not. However, a notable difference is that for a particular problem, only the weights of the chromosome involved in that problem were mutated (such as parameters involving king safety were mutated when failing a problem involving checkmate). The resulting individual was able to win 29/200 games against a grandmaster-level agent, a strong achievement considering the limitations of GAs for chess. While the overall performance of GA-based chess engines still lags behind modern state-of-the-art methods, the reviewed works show clear evidence that evolutionary approaches can yield competitive and adaptable players, particularly when combining search and evaluation optimization. However, it is worth highlighting that these approaches do not evolve agents across entire games, but rather focus on developing position evaluators.

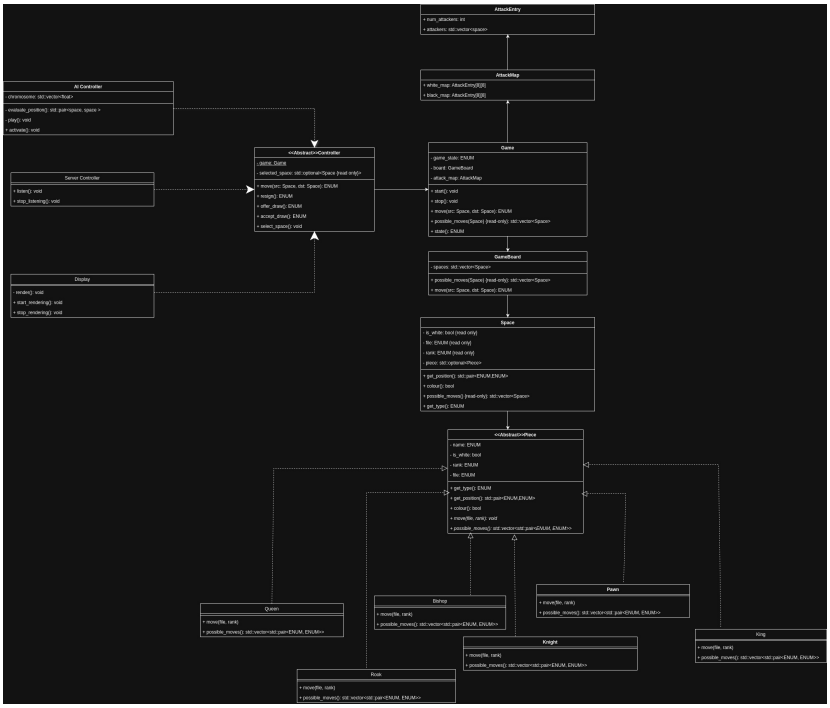

Fig. 1: Chess Engine UML Schematic

III. Methodology

A. The Chess Engine

1) Environment: We built our chess engine from the ground up. The application was written entirely in C++20 [7]. The project is built using CMake 3.31 [8] and Ninja Build 1.10.1 [9]. Conan 2.18.1 [10] was used to manage 3rd party dependencies. DearIMGui 1.90.9 [11] with GLFW 3.4 [12] on the backend was used to render the GUI. IXWebsocket 11.4.5 [13] was used to implement the multiplayer functionality. NlohmannJSON 3.11.0 [14] was used to deserialize the chromosome.

2) Game Logic Implementation: We started by designing the software architecture 1. We decided that the best approach was to separate the application into modules so that we could separate concerns such that game logic was distributed across all classes.

a) Pieces and Spaces: Spaces and pieces are closely coupled when it comes to the game logic, and serve the same purpose. A piece is responsible for knowing how it moves and where it is located. When a piece wants to check its valid moves it returns a list of all of the moves that are valid from its current location. The piece is not responsible for checking collisions.

b) Game Board: The game board is implemented as a 2D vector of spaces. The board therefore has knowledge of piece locations and thus it is responsible for detecting collisions and filtering moves from the piece that are invalid due to collision (or lack of collision in the case of the pawn). The board also manages pawn promotion.

c) Game: The game class is responsible for managing logic related to game state such as castling and checking. The game class will filter out moves that put the king in danger, or fail to get the king out of danger. The game class holds a read-only reference to the king pieces so it can locate the kings in constant time. The game class also implements castling logic including not allowing the king to castle through check and not allowing the king to castle out of check. The final major responsibility of the game class is to maintain the state of the game. The game class knows whose turn it is and if the player is in check.

d) Controllers: The engine supports local players, remote players, and allows for the GA to play the game. Each player type is implemented as a controller. The controller base class has a static class member instantiation of the game class. Due to this, all instantiations of the controller within the same process will share the same game object. The local player is represented by the Display class, which communicates with DearIMGui and is responsible for rendering the board graphically and allowing the user to interact with the render. Remote players can connect to the server controller class, and a sample implementation of a client controller is provided using our API which is described in III-A3. The GA can make moves using the AI Controller which interacts with the Game object directly using the implementation described in III-B.

3) Networking: Multiplayer is implemented via the server controller over our WebSocket [15]-based protocol. We implemented our protocol using endpoints encoded using a directory structure with the following endpoints.

1) `/get/board/`

2) `/get/possible_moves/<src>`
3) `/post/move/<src><dst>`
4) `/get/status/`

a) Get Board: Endpoint 1 allows the client to request a serialized version of the board so that it can view the current state of the game. See Figure 2 for a sample serialization output. Black pieces are lowercase and white pieces are uppercase.

b) Get Possible Moves: Endpoint 2 returns all valid destination squares for a given source square. The source square must be specified using standard algebraic notation (e.g., `e2`), and the response consists of a single unbroken string containing all legal destination squares, each encoded in algebraic notation and concatenated without delimiters. For example, if the piece on `e2` can move to `e3`, `e4`, and `f3`, the endpoint will return the string `e3e4f3`.

c) Post Move: To submit a move, the client calls Endpoint 3 using a request path that concatenates the source and destination square identifiers, again in algebraic notation and without separation. For instance, a move from `e2` to `e4` would be submitted as `/post/move/e2e4`. No separator or special character is used between the source and destination.

d) Get Status: Finally, Endpoint 4 provides high-level information about the game state, including whose turn it is, whether the game has concluded, and the outcome if applicable.

e) On Error: In the case of an error the server will respond with an empty string. A valid response will have the target endpoint as a prefix to the response.

B. The Genetic Algorithm

1) Environment: The genetic algorithm and the chess engine were in development at roughly the same time, so when the genetic approach began being prototyped, a functional chess engine was not available. Thus, the evolution algorithm was developed in python using the python-chess library[], and was subsequently integrated into the C++ chess engine once core functionality was finalized. The Python environment used Python 3.11.8, python-chess 1.999, and numpy 2.2.6. Once translated to the C++ engine, it used the C++ engine environment.

2) The Chromosome: The chromosome was designed as a 1D vector of weights corresponding to different aspects of chess. It consists of 18 chess principles and 6 64-element positional matrices. The 18 chess principles are material score, piece mobility bonus, castling bonus, development speed bonus, doubled pawn penalty, isolated pawn penalty, connected pawn bonus, enemy king pressure bonus, piece defense bonus, bishop pair bonus, connected rooks bonus, king centralization, knight outpost bonus, blocked piece penalty, space control in opponent half bonus, king shield bonus, and king pressure penalty. These principles can be described as a ”bonus” or ”penalty” indicating an ideal that these principles should positively or negatively impact the resultant position evaluation. However, the evolution process may result in the opposite being the case, and it is up to the evaluation function to make a sufficient distinction, The 6 64-element positional matrices represent ideal position placements for each piece type on the board, from white's perspective.

This results in a chromosome containing 403 different weights that are all tuned during the evolution process.

3) The Evaluation Function: The evaluation function computes its own scoring for each of the 18 chess principles described by the chromosome. The evaluation was made to be modular, featuring 18 different scoring functions, which are described in pseudocode in VI. They were designed to return an arbitrary score value, leaving it up to the chromosome to tune the scaling. King pressure is identical to enemy king pressure, but the score is negatively applied, and thus there is no need for two separate functions. Then, for each positional matrix, it determines the rank and file of each piece of a given color and adds the corresponding weight at positional_matrix[rank][file] to the score. The position matrices are defined based on white's perspective, so they need to be mirrored for black. During evaluation, if the board is in a state of checkmate, the score is overridden. If the individual won, the score is 100,000, and conversely -100,000 if the individual lost. This is to ensure the agent prioritizes positions that lead to victory and avoid positions that lose at all costs.

4) The Search Function: The search function was not a key focus of this work, and thus is not involved in any evolutionary approach. It is a basic minimax with alpha/beta pruning:

And in order to pick the best move, 1 is called for every legal move, every turn:

5) Evolution: Our genetic algorithm follows these steps:

Each chromosome is evaluated by playing a full game against a depth-1 stockfish. Stockfish is a famous chess engine known for its incredibly strong performance, and it consistently beats the strongest players in the game. In fact, playing 100 games against the current strongest player in the world, Magnus Carlsen, stockfish won 50 games and drew the other 50 (meaning it never lost). However, stockfish generally operates at depths of 20 or higher, and this is too powerful for individuals in our population. Thus, we limit stockfish by only allowing it to examine the next move. The resultant fitness is (-1000 + move count) for a loss, 1000 for a win, and material score for a draw against stockfish. However, depth-1 stockfish, while limited, still has access to the opening playbook. This means that it knows to make the best possible moves during the opening phase of the game, which negatively impact evolution as the individuals have no such playbook - meaning within the first 5-10 moves of the games it is entirely likely that even a good individual gets a terrible position and subsequently loses the game. In order to mitigate this issue to some extent, games against the stockfish engine begin in the middle game, after all opening moves have been played. This is achieved by having the stockfish engine play itself for 20 moves at depth 4, and then the individuals play against a depth 1 stockfish from that position. To avoid situations where both players infinitely cycle pieces, there is a max move count of 200, and the game is set to a draw if the maximum move count is reached.

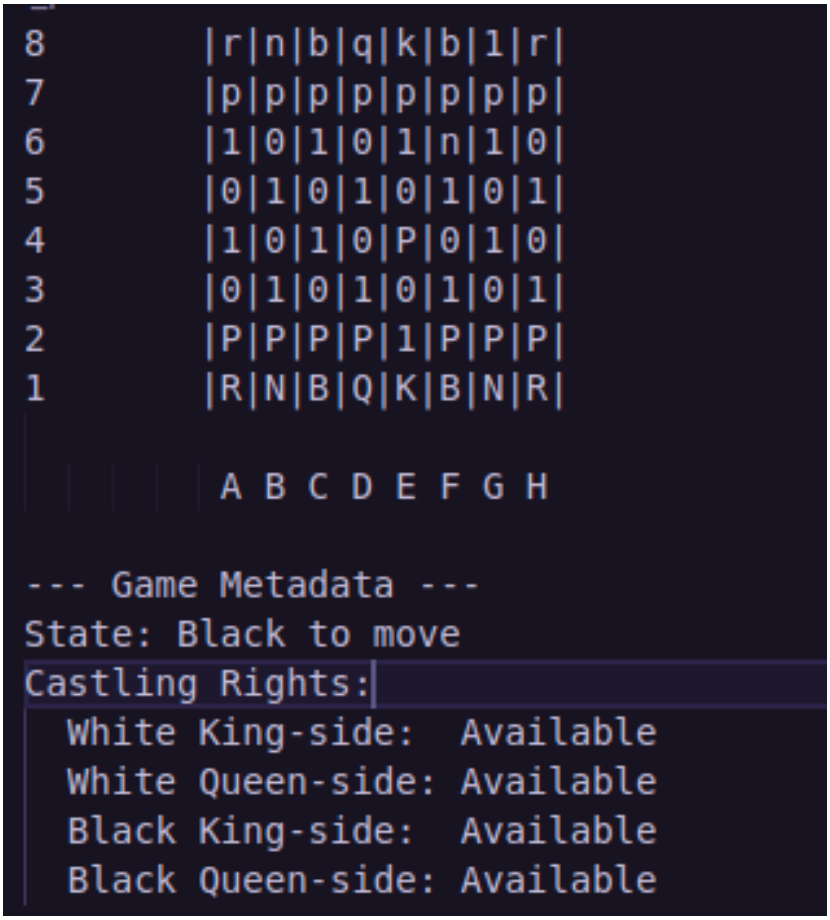

Fig. 2: Sample Board Serialization

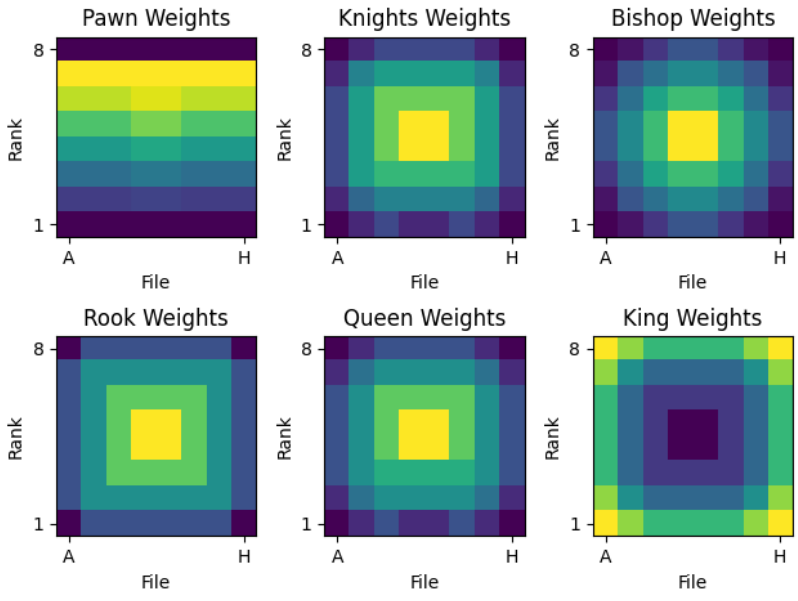

Fig. 3: Piece Positional Weights (8x8 grid)

Algorithm 1: Minimax(*board*, *chromosome*, *depth*, α, β, *maximizing*)

```
if depth = 0 or board.is_game_over() then
    score ← evaluate_position(board, chromosome, white = maximizing)
    return score
if maximizing then
    max_eval ← −∞
    foreach move ∈ board.legal_moves() do
        board.move(move)
        eval ← Minimax(board, chromosome, depth − 1, α, β, False)
        board.undo(move)
        max_eval ← max(max_eval, eval)
        α ← max(α, eval)
        if β ≤ α then
            break
    return max_eval
else
    min_eval ← ∞
    foreach move ∈ order_moves(board) do
        board.move(move)
        eval ← Minimax(board, chromosome, depth − 1, α, β, True)
        board.undo(move)
        min_eval ← min(min_eval, eval)
        β ← min(β, eval)
        if β ≤ α then
            break
    return min_eval
```

Algorithm 2: SelectBestMove(*board*, *chromosome*, *depth*)

```
best_move ← None
best_score ← −∞ if board.turn = WHITE, else ∞
foreach move ∈ board.legal_moves do
    board.move(move)
    is_maximizing ← ¬board.turn
    score ← Minimax(board, chromosome, depth − 1, −∞, ∞, is_maximizing)
    board.undo()
    if board.turn = WHITE and score > best_score then
        best_score ← score
        best_move ← move
    else if board.turn = BLACK and score < best_score then
        best_score ← score
        best_move ← move
return best_move
```

Algorithm 3: Genetic Algorithm

```
population ← generate_chromosomes(POPULATION_SIZE)
for gen ← 0 to NUM_GENERATIONS − 1 do
    fitnesses ← []
    foreach chromosome ∈ population do
        fitness ← evaluate_fitness(chromosome)
        fitnesses.append((chromosome, fitness))
    fitnesses.sort()
    next_generation ← [fitnesses[0][0], fitnesses[1][0]]
    while ‖next_generation‖ ≠ POPULATION_SIZE do
        parent1 ← tournament_selection(fitnesses)
        parent1 ← tournament_selection(fitnesses)
        child ← crossover(parent1, parent2
        child ← mutate(child), 1.0, 0.3 × (1 − (gen/NUM_GENERATIONS))
        next_generation.append(child)
    population ← next_generation
```

IV. Discussion

A. Limitations

Ultimately, even the best agent struggles to defeat a depth-1 stockfish consistently. This likely highlights the key limitation with our approach and justifies the approach taken by other works in the literature - fundamentally, evaluating fitness based on an entire game is not ideal for evolving a strong chess engine. At a low-level, sufficiently evaluating an arbitrary chess position is all that is necessary for a chess engine, and thus requiring the engine to be performant across an entire game adds too much variability that is difficult to capture adequately in the fitness evaluation function. A potential approach to mitigate the negative impacts of evaluating over the entire game would be to have three values for each gene in the chromosome: opening, middle game, and endgame. This would enable the chromosome to transition its weights to values more appropriate for the current state of the game. For example, king centralization was a score intended to incentivize the individual to prioritize centering the king, which is extremely valuable in the end game as with so few pieces on the board the king is safely able to play an offensive role. However, centralization of the king in the early game is a surefire way to instantly lose, so this score not distinguishing between the game state is not correct. It would be better to have a king centralization score for the opening that effectively forces the king avoid the center at all costs, and transition to a score that encourages centralization in the endgame. This

Algorithm 4: tournament_selection($fitnesses$)

```
k ← POPULATION_SIZE × 0.05
selected ← k random elements from fitnesses
selected.sort()
return selected[0][0]
```

Algorithm 5: Crossover($parent1$, $parent2$)

```
child ← [];
for i ← 0 to ‖parent1‖ do
    alpha ← random_number(0.4, 0.6);
    child_gene ←
      alpha × parent1[i] + (1 − alpha) × parent2[i];
    child.append(child_gene);
return child
```

logic applies to all genes in the chromosome, enabling more dynamic play.

Additionally, the evolution process does not rely on any prior knowledge about the game other than valid moves, such as strong opening move sequences or grandmasters moves in certain positions. As such, it has no real way to optimize for incredibly strong play, it mainly attempts to optimize for reasonable play, but lacks a more fundamental understanding of the board. It is theoretically possible for the agents to become strong, but would require significantly more generations and simulated games in order to eventually reach that point, assuming the evaluation function can adequately reach that level. Optimizing for a strong opening sequence, or for the best move in a given position using a known grandmaster move, similar to what was done in the literature, would greatly help the performance of these individuals.

With no restrictions on the piece positional matrices, the matrices quickly lose their structure as well, so the value of a piece on a square loses meaning, and basically becomes noise. It is clear from this that the individual cannot optimize for key squares per piece type without additional guidance from an external source. Comparisons between the initialized position matrices and their corresponding evolved version are in VI.

Algorithm 6: Mutate($chromosome$, $mutation\_rate$, $mutation\_strength$)

```
for i ← 0 to ‖chromosome‖ do
    if random_number(0.0, 1.0) < mutation_rate
     then
        chromosome[i] ← chromosome[i] +
          random_number(−mutation_strength,
                        mutation_strength);
return chromosome
```

B. Results

Three tests were run with different agents at depth 4 playing 100 games against the depth 1 stockfish, starting from a midgame position. The first test was with a completely randomly initialized chromosome, the second was with an agent resulting from 150 generations of the GA, and the third was with that same agent, but with the original positional matrices. Based on the fact that the results improve, it's clear to us that incentivizing the agent to preserve the distribution of a positional matrix is to its benefit, and a future step would be to enforce that preservation in the mutate function.

Overall, we see the flaw in designing an approach when the chromosome is evaluated based on its ability to play an entire game, and our results justify other research where the fitness function evaluated an individual's ability to make the correct move for one given position. Chess is not inherently a time-series game, positions can be evaluated in a vacuum, and thus the over-reliance on finishing an entire game leads to worse results. This approach is likely powerful enough to defeat Stockfish at higher depths, but the evolution approach needs to be modified to work for specific positions rather than games.

V. Contributions of Team Members

Vlad Tusinean - Responsible for the genetic algorithm and most of its translation to C++, additionally worked on optimizations to the chess engine.

Matthew Belanger - Implemented chess engine, designed the application, helped with GA translation. Lead C++ development.

References

[1] The Stockfish Developers, "Stockfish 17.1," https://github.com/official-stockfish/Stockfish, 2025, open-source chess engine; version 17.1 released March 302025.

[2] "GP-EndChess: Using Genetic Programming to Evolve Chess Endgame Players," in *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 120–131, iSSN: 0302-9743, 1611-3349. [Online]. Available: https://link.springer.com/10.1007/978-3-540-31989-4_11

[3] O. David-Tabibi, M. Koppel, and N. S. Netanyahu, "Genetic algorithms for mentor-assisted evaluation function optimization," in *Proceedings of the 10th annual conference on Genetic and evolutionary computation*. Atlanta GA USA: ACM, Jul. 2008, pp. 1469–1476. [Online]. Available: https://dl.acm.org/doi/10.1145/1389095.1389382

[4] ——, "Optimizing Selective Search in Chess," Sep. 2010, arXiv:1009.0550 [cs]. [Online]. Available: http://arxiv.org/abs/1009.0550

[5] O. E. David, H. J. van den Herik, M. Koppel, and N. S. Netanyahu, "Genetic Algorithms for Evolving Computer Chess Programs," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 5, pp. 779–789, Oct. 2014. [Online]. Available: https://ieeexplore.ieee.org/document/6626616/

[6] E. Vázquez-Fernández and C. A. C. Coello, "An adaptive evolutionary algorithm based on tactical and positional chess problems to adjust the weights of a chess engine," in *2013 IEEE Congress on Evolutionary Computation*, Jun. 2013, pp. 1395–1402, iSSN: 1941-0026. [Online]. Available: https://ieeexplore.ieee.org/document/6557727/

[7] International Organization for Standardization, *ISO/IEC 14882:2020: Programming Languages — C++*, 2020, https://isocpp.org/std/the-standard.

[8] Kitware, Inc., *CMake: Cross-Platform Make*, 2024, https://cmake.org/.

[9] E. Martin, *Ninja: a small build system with a focus on speed*, 2024, https://ninja-build.org/.

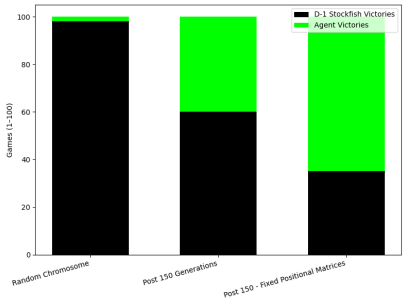

Fig. 4: Agent vs. Depth-1 Stockfish

[10] JFrog Ltd., *Conan: C/C++ Package Manager*, 2024, https://conan.io/.

[11] O. Cornut, "Dear imgui: Immediate mode gui," 2024, version 1.90.9, https://github.com/ocornut/imgui.

[12] C. Löwy, *GLFW: A multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop*, 2023, version 3.4, https://www.glfw.org/.

[13] B. Sergeant, "ixwebsocket: Websocket client and server for c++," 2024, version 11.4.5, https://github.com/machinezone/IXWebSocket.

[14] N. Lohmann, "Json for modern c++," 2022, version 3.11.0, https://github.com/nlohmann/json.

[15] I. Fette and A. Melnikov, "The WebSocket Protocol," RFC 6455, December 2011, https://datatracker.ietf.org/doc/html/rfc6455. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6455

VI. Appendix

Appendix

Algorithm 7: ComputeMaterialScore($board$, $white$)

$score \leftarrow 0$
piece_values $\leftarrow$ {
    PAWN: 1,
    KNIGHT: 3,
    BISHOP: 3,
    ROOK: 5,
    QUEEN: 9
}
foreach *(piece_type,* $value$*)* $\in$ *piece_values* **do**
    $score \leftarrow$
     $score + \|board.pieces(piece\_type)\| \times value$
    $score \leftarrow$
     $score - \|board.pieces(piece\_type)\| \times value$
return score

Algorithm 8: ComputePieceMobility($board$, $white$)

$score \leftarrow 0$
foreach $move \in board.legal_moves$ **do**
    $rank \leftarrow move.rank$
    $file \leftarrow move.file$
    $attackers \leftarrow board.attackers((file, rank), color$
    $defenders \leftarrow$
     $board.attackers((file, rank), \neg color)$
    if $attackers \leq defenders$ **then**
        $score \leftarrow score + 1$
return $score$

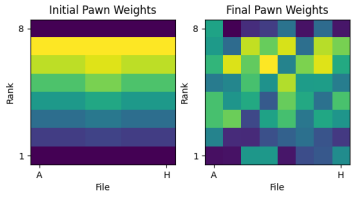

Fig. 5: Pawn Positional Weights (8x8 grid)

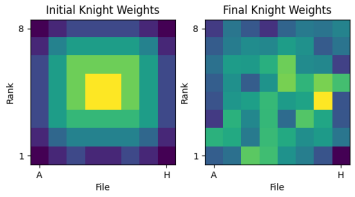

Fig. 6: Knight Positional Weights (8x8 grid)

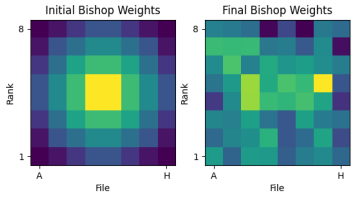

Fig. 7: Bishop Positional Weights (8x8 grid)

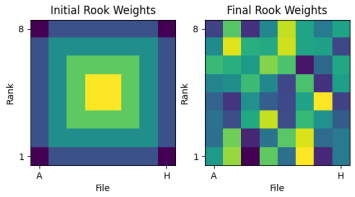

Fig. 8: Rook Positional Weights (8x8 grid)

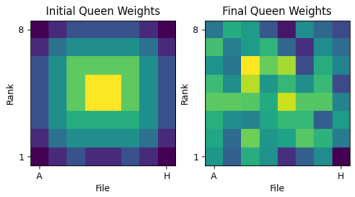

Fig. 9: Queen Positional Weights (8x8 grid)

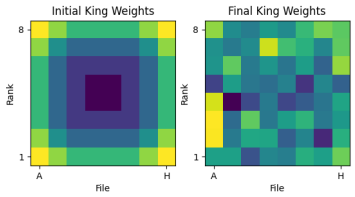

Fig. 10: King Positional Weights (8x8 grid)

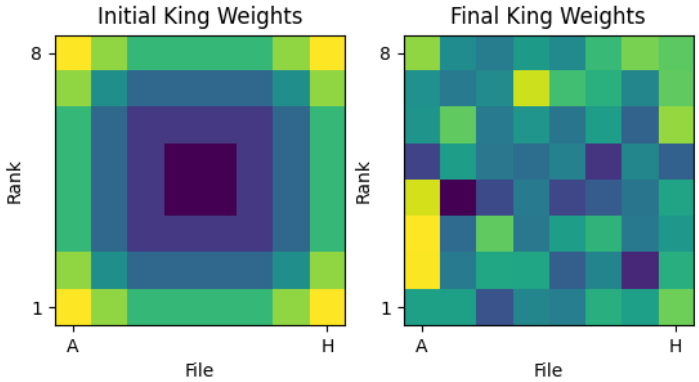

Fig. 11: King Positional Weights (8x8 grid)

Algorithm 9: ComputeCastlingBonus($board$, $white$)

```
score ← 0
if board.castling_rights(color) then
    score ← score + 1
else
    if board.last_move(color) == castling then
        score ← score + 2
return score
```

Algorithm 10: ComputeDevelopmentSpeed($board$, $white$)

```
score ← 0
color ← WHITE if white, else BLACK
foreach piece ∈ pawns of color do
    if piece.rank ≠ board.start_rank(piece) then
        score ← score + 0.5
foreach piece_type ∈ {KNIGHT, BISHOP, ROOK, QUEEN} do
    foreach piece ∈ board.get_pieces(piece_type, color) do
        if piece.rank ≠ board.start_rank(piece) then
            score ← score + 1
return score
```

Algorithm 11: ComputeDoubledPawnScore($board$, $white$)

```
score ← 0
color ← WHITE if white, else BLACK
file_counts ← [0, 0, 0, 0, 0, 0, 0, 0]
foreach piece ∈ board.pieces(PAWN, color) do
    file_counts[piece.file] ← file_counts[piece.file] + 1
foreach count ∈ file_counts do
    if count > 1 then
        score ← score + (count − 1)
return score
```

Algorithm 12: ComputeIsolatedPawnScore($board$, $white$)

```
score ← 0
color ← WHITE if white, else BLACK
files_with_pawns ← []
foreach piece ∈ board.pieces(PAWN, color) do
    files_with_pawns.append(piece.file)
foreach piece ∈ board.pieces(PAWN, color) do
    isolated_left ← (piece.file = 0) or (piece.file − 1 ∉ files_with_pawns)
    isolated_right ← (piece.file = 7) or (piece.file + 1 ∉ files_with_pawns)
    if isolated_left and isolated_right then
        score ← score + 1
return score
```

Algorithm 13: ComputeConnectedPawnScore($board$, $white$)

```
score ← 0
color ← WHITE if white, else BLACK
direction ← −1 if white, else 1
pawns ← board.pieces(PAWN, color)
foreach pawn ∈ pawns do
    behind_rank ← pawn.rank + direction
    if 0 ≤ behind_rank ≤ 7 then
        if pawn.file − 1 ≥ 0 and (pawn.file − 1, behind_rank) ∈ pawns then
            score ← score + 1
        if pawn.file + 1 ≤ 7 and (pawn.file + 1, behind_rank) ∈ pawns then
            score ← score + 1
return score
```

Algorithm 14: ComputePassedPawnScore($board$, $white$)

```
score ← 0
color ← WHITE if white, else BLACK
foreach pawn ∈ board.pieces(PAWN, color) do
    passed_pawn ← True
    if white then
        foreach enemy_pawn ∈ board.pieces(PAWN, BLACK) do
            if abs(pawn.file − enemy_pawn.file) ≤ 1 and pawn.rank < enemy_pawn.rank then
                passed_pawn = False
        if passed_pawn then
            score ← score + 1
    else
        foreach enemy_pawn ∈ board.pieces(PAWN, WHITE) do
            if abs(pawn.file − enemy_pawn.file) ≤ 1 and pawn.rank > enemy_pawn.rank then
                passed_pawn = False
        if passed_pawn then
            score ← score + 1
return score
```

Algorithm 15: ComputeKingPressureScore($board$, $white$)

```
score ← 0
color ← WHITE if white, else BLACK
king ← board.pieces(KING, color)
directions ← [(-1,-1), (0,-1), (1,-1), (-1,0), (1,0), (-1,1), (0,1), (1,1) ]
foreach (df, dr) ∈ directions do
    f ← king.file + df
    r ← king.rank + dr
    if 0 ≤ f ≤ 7 and 0 ≤ r ≤ 7 then
        if (f, r) is attacked by ¬color then
            score ← score + 1
temp_board ← board.copy()
foreach square ∈ all squares do
    piece ← temp_board.piece_at(square)
    if piece exists and piece.color = color and piece.type ≠ KING then
        remove piece from temp_board at square
score ← score + ||temp_board.attackers(king, ¬color)||
return score
```

Algorithm 16: ComputePieceDefenseScore($board$, $white$)

```
score ← 0
color ← WHITE if white, else BLACK
foreach piece_type ∈ {PAWN, KNIGHT, BISHOP, ROOK, QUEEN} do
    foreach piece ∈ board.pieces(piece_type, color) do
        if ||board.attackers(piece, ¬color)|| > 0 then
            score ← score + 1
return score
```

Algorithm 17: ComputeBishopPairScore($board$, $white$)

```
score ← 0
color ← WHITE if white, else BLACK
if ||board.pieces(BISHOP, color)|| ≥ 2 then
    score = 1
return score
```

Algorithm 18: ComputeConnectedRooksScore($board$, $white$)

```
score ← 0
color ← WHITE if white, else BLACK
rooks ← board.pieces(ROOK, color)
for i ← 0 to ‖rooks‖ − 1 do
    for j ← i + 1 to |rooks| − 1 do
        if (rooks[i].rank = rooks[j].rank or
          rooks[i].file = rooks[j].file) and no pieces
          between rooks[i] and rooks[j] then
            score ← score + 1
return score
```

Algorithm 19: ComputeKingCentralizationScore($board$, $white$)

```
score ← 0
color ← WHITE if white, else BLACK
king ← board.pieces(KING, color)
center_squares ← [(3, 3), (4, 3), (3, 4), (4, 4)]
distances ← [ ]
foreach (cf, cr) ∈ center_squares do
    d ← max(|king.file − cf|, |king.rank − cr|)
    distances.append(d)
if white then
    score ←
      − min(distances) − 0.25 × (7 − king.rank)
else
    score ← − min(distances) − 0.25 × king.rank
return score
```

Algorithm 20: ComputeKnightOutpostScore($board$, $white$)

```
score ← 0
color ← WHITE if white, else BLACK
knights ← board.pieces(KNIGHT, color)
foreach knight ∈ knights do
    if (white and knight.rank < 4) or (¬white and
      knight.rank > 3) then
        continue
    if knight is protected by friendly pawn then
        attack_squares ← []
        rank ← knight.rank + 1 if white else
          knight.rank − 1
        if ¬(0 ≤ rank ≤ 7) then
            continue
        if 0 < file − 1 and enemy pawn on (file - 1,
          rank) then
            blocked_left = True
        if 0 < file + 1 and enemy pawn on
          (file + 1, rank) then
            blocked_right = True
        if ¬blocked_left and ¬blocked_right then
            score ← score + 1
return score
```

Algorithm 21: ComputeBlockedPieceScore($board$, $white$)

```
score ← 0
color ← WHITE if white, else BLACK
foreach piece_type ∈ {KNIGHT, BISHOP, ROOK,
  QUEEN} do
    foreach piece ∈ board.pieces(piece_type, color)
      do
        if ‖board.moves(piece)‖ = 0 then
            score ← score + 1
return score
```

Algorithm 22: ComputeSpaceControlScore($board$, $white$)

```
score ← 0
color ← WHITE if white, else BLACK
if color then
    ranks ← [4, 5, 6, 7]
else
    ranks ← [0, 1, 2, 3]
foreach rank ∈ ranks do
    for file ← 0 to 7 do
        if board.attackers((file, rank), color) then
            score ← score + 1
return score
```

Algorithm 23: ComputeKingShieldScore($board$, $white$)

```
color ← WHITE if white, else BLACK
king ← board.king(color)
forward ← 1 if white else −1
shield_rank ← king.rank + forward
score ← 0
foreach df ∈ {−1, 0, 1} do
    file ← king.file + df
    if 0 ≤ file ≤ 7 and 0 ≤ shield_rank ≤ 7 then
        piece ← board.piece_at(file, shield_rank)
        if piece exists and piece.color = color then
            if piece.type = PAWN then
                score ← score + 1
            else
                score ← score + 2
return score
```

Algorithm 24: ApplyPositionMatrix($board$, $white$, $piece\_type$, $matrix$)

```
score ← 0
color ← WHITE if white, else BLACK
foreach piece ∈ board.pieces(piece_type, color do
    if white then
        score ←
          score + matrix[piece.rank][piece.file]
    else
        (file, rank) ← board.mirror_square(piece)
        score ← score + matrix[rank][file]
return score
```