Applications of Genetic Algorithms to Chess

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Elliot M. Harris

Annandale-on-Hudson, New York
May, 2024

Abstract

This thesis discusses the use of genetic algorithms to tune the parameters of a chess engine, resulting in a significant increase in playing strength. The design of the genetic algorithms builds on the 2008-2011 work of David-Tabibi et al. [13] and Vázquez-Fernández et al. [32]. The overwhelmingly positive result presented in this thesis not only suggests a promising potential for genetic algorithm use to improve computer chess, but also supports the efficacy and potential of applying genetic algorithms to a broader set of use cases.

Contents

Dedication

This thesis is dedicated to the Bard Chess Club.

Acknowledgments

First and foremost, I extend my deepest appreciation to my advisor, Valerie Barr, for her invaluable guidance, encouragement, and support.

I am also thankful to Bard's Computer Science Department for providing the necessary resources and facilities, and specifically to Sven Anderson for lending me unabated access to the Cognitive Systems Lab.

A special thanks to all of those who provided an extra pair of eyes in the process of editing: Michaela, Dasha, Valerie, and my dad.

I want to further include my tribe, the Federated Indians of Graton Rancheria, for their academic support and wisdom during my tenure at Bard.

I would like to thank the expert help that was provided from the online community of engine programmers found on the Discord server, Engine Programming. As with most research, having a community to bounce ideas off and to hear feedback from was invaluable to the finished product.

I would like to give enormous praise and gratitude to my supportive family, who encouraged me when I felt stuck, who listened on long phone calls, and who celebrated all the tiny successes along the way. To my mom Michelle, my dad Greg, Candace, and my brother Ari.

To all my amazing friends, who patiently listened to my rantings, offered encouragement when said ranting became despondent, and who suspended their understanding of the project for the sake of providing a listening ear – your sacrifice did not go unnoticed.

And last but certainly not least, I would like to extend my heartfelt appreciation towards my amazing partner, Dasha, who stuck with me and my craziness day in and day out.

1
Introduction

1.1 Motivation

I first came across Genetic Algorithms in my sophomore year in an introduction to artificial intelligence course. I was struck by their mimicry of nature, and how a seemingly simple set of rules and processes were able to produce extremely fast approximations to difficult problems. The general premise, that one can achieve astonishingly good results simply by letting the laws of nature take their course, was enticing, and stuck with me as I was deliberating on a senior thesis topic. Moreover, I was also deeply troubled by the increasing usage of neural networks and deep learning. While there have been some remarkable outcomes as a result of these machine learning methods, they also have some clear downsides. I was dismayed by both the extreme computing power required of deep learning models, as well as the complete lack of interpretability of their results. Often described as black boxes [8], it is extremely difficult to visualize or describe why or how a model settles on an output [30]. The issues of computing power and interpretability are hardly present in the realm of genetic algorithms, however, and thus a senior thesis idea was born.

1.2 Choosing a Problem Space

For the purpose of choosing a suitable problem space to appropriately test the efficacy of a classic machine learning method, I wanted to explore a classic problem. Many such classic machine learning problems exist, such as optical character recognition (OCR), the host of NP-Hard problems such as the travelling salesperson problem or the knapsack problem [2], or one that I hold a personal affinity towards: the game of chess.

In fact, one of the first tests of a new computing system or intelligence paradigm links back to the two millennium old game [1]. Whether humanity's obsession with chess is linked to its perceived connection to intelligence, or because it so allows for a proxy war between kings, countries, or important figures, the fact remains that chess has held international attention for centuries. In fact, chess popularity burgeoned during the pandemic, and continues its popular trend today, with sites such as Chess.com boasting more than 150 million members [6].

Computer chess was almost synonymous with early AI. Alan Turing, sometimes deemed the father of modern computing, sought to develop a chess program, Turochamp, as one of his first attempts at artificial intelligence [10]. Since then, some of the most successful examples of chess AI breakthroughs have involved specialized chips designed for the game of chess, such as Carnegie Mellon's ChipTest in 1987 [4] or IBM's Deep Blue, which famously defeated Russian Grandmaster Gary Kasparov in 1997. That match is also widely accepted as the first moment AI surpassed human ability in the game [1]. Probably the next most important milestone in computer chess was Google's AlphaZero, in 2017, which learned to play chess at an unprecedented strength by playing games against itself in an unsupervised deep learning model [27]. Not only did these breakthroughs advance collective understanding of how computers might master the game of chess, they also provided an undeniable piece of evidence for the efficacy and power of artificial intelligence. Chess, a game that involves both sides having access to all available data, a nearly infinitely deep search tree of possibilities, and a standardized Elo[1] rating system to objectively measure strength, provides the perfect test for a new artificial intelligence approach.

[1] See Appendix section A.3.4.

Before incorporating a potentially life-altering system to diagnose a patient, or making a large financial investment, an artificial intelligence method must first prove itself on the battleground of 64 squares.

1.3 The General Design of Chess Engines

To understand how exactly a genetic algorithm can be applied to a chess engine, it is first necessary to have an overview of the components of a chess engine. The first component to any chess engine is the internal representation for the chess board. A chess engine must know, at each stage, all the legal moves available, as well as where every piece is at all times. Many different representations are possible, with approaches such as 2D arrays, 1D arrays, string representations, or bit representations known as bitboards. Of these, bitboards have emerged as the fastest, due to the fact that bit operations are among the most efficient operations possible on modern hardware.[2]

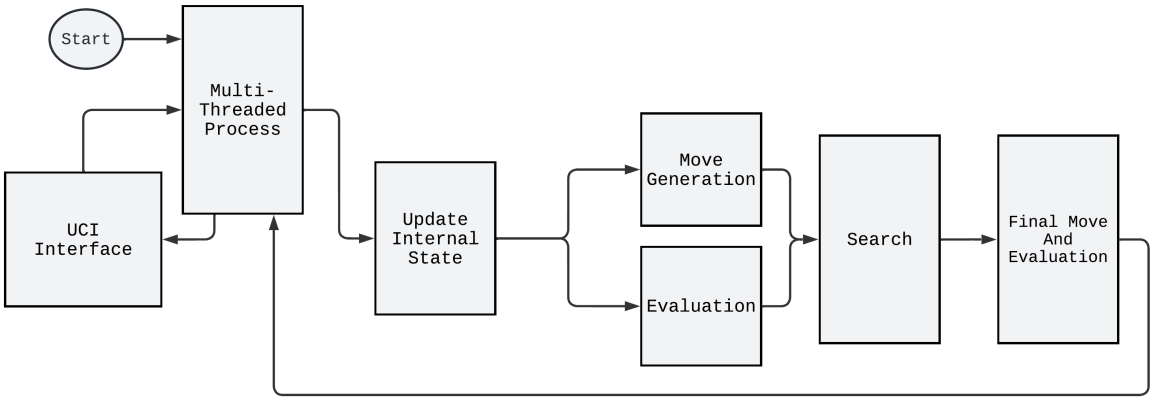

Figure 1.3.1: Generalized Architecture of a Modern Chess Engine

Since many different implementations are used, and internal representations are just that – internal, there needs to be a mechanism by which an engine can actually play against an opponent with a different board-state representation (such as another engine or a human opponent). The ubiquitous solution is called the Universal Chess Interface (UCI), which operates through

[2]For more on bitboards and bit operations, see Appendix A.1.

predefined commands given and received through standard Input/Output [10]. Often an engine will dedicate a separate thread to handle UCI commands, as seen in Figure 1.3.1.

Once the board is accurately represented, and all legal moves for any position can be checked, a move selection search is possible. Because most chess games are timed competitions, this search is limited to a short fraction of a total game clock, thus placing a large emphasis on the efficiency and speed of this search. The best chess engines dedicate all their time to relevant branches of the search tree, and aggressively discard the rest. As outlined in Figure 1, the majority of move selection algorithms consist of two integral parts, a search and an evaluation function.[3] A search function is for examining potential moves and counter-moves within the unfathomably large search tree that results from most chess positions, whereas the evaluation function acts as a heuristic that estimates to the search how far away any given position is from a terminal node such as a checkmate or draw. The two work in tandem to reach a final verdict on the best move to play next. Keep in mind that the game of chess has approximately $2 * 10^{40}$ legal positions, even without accounting for promotions [28]. For context, the number of atoms in the human body is a much lower 10^{27}. Because of the extremely large search space, the search is trying to find the path that will result in the best heuristic score for the engine, and thus the search strength is directly related to the evaluation strength.

The classical architecture uses a highly optimized alpha-beta search, and a heuristic evaluation function consisting of weighted hand-extracted higher level features added together in a linear equation. A much deeper and practical analysis of this specific architecture will be explored in Chapter 3, but the important takeaway is that both the evaluation and search function are parameterized functions. In other words, their performance relies on having an optimal set of weights, margins, and constant values that operate together. Optimizing these values is typically done in a separate tuning process, rather than during a game or match itself, and proves a difficult challenge as the search space is neither uni-modal nor convex. This sets the stage for a genetic algorithm to be applied. In fact, a multi-modal search space without a single obvious

[3]The author is unaware of any highly competitive engine for which this is not the case.

optimal peak is exactly the kind of search space that genetic algorithms are suitable for [19]. Furthermore, a known weakness of genetic algorithms is their inability to find a truly optimal solution. In the case of chess, however, there may not even be one singular 'optimal' solution, but rather a number of possible parameter values that can successfully be found, making this an ideal task for a genetic algorithm.

1.4 The History of Genetic Algorithms and Their Role in Chess

Genetic Algorithms were first formally introduced in 1992 by J.H Holland, and were based on the Darwinian theory of survival of the fittest in nature [19]. Since then, many variations and optimizations have been used to increase their power and adapt them to various problems, including co-evolution or a historical mechanism. They have been employed successfully in many NP-Hard problems (problems that do not have a known polynomial-time solution), including in the sub-space of games and puzzles [19]. Ironically, one of their most prominent use-cases has been in finding better hyperparameters for neural networks, the machine learning method that has spurred major breakthroughs in chess engines and the field of artificial intelligence [11].

Fortunately, there is already a fair amount of research on the application of genetic algorithms in the field of chess engine programming. Perhaps the most successful instance of the crossover between chess and genetically inspired tuning is the chess engine Falcon which was able to place second in the blitz section of the World Computer Chess Championship after employing a genetic algorithm to increase its playing strength [13].

More recently, the same authors proposed a pipeline consisting of two genetic algorithms as well as a co-evolution stage as the basis for tuning a chess engine's parameters. Other research has been completed and improvements have been implemented, for example the modified genetic algorithm architecture of Vázquez-Fernández et al. [32], resulting in a higher rated chess engine.

In both of the above cases, however, the claim of creating a higher rated engine was dubious, since an accurate baseline was not provided. In the development process of many chess engines, the programmer or an expert chess player provides a set of manually chosen parameters. In fact,

automated parameter tuning in chess engines was not very common or successful for many top engines prior to 2008 [13]. Thus, the value of a machine learning algorithm such as a genetic algorithm in the domain of chess is found only in its ability to increase the strength of a chess engine beyond this manually tuned baseline. This thesis aims to provide such a manual baseline that allows for a more robust discussion of results.

In sum, this thesis will explore the use of genetic algorithms for the tuning of a chess engine's parameters in an attempt to more broadly test the efficacy of genetic algorithms as a useful machine learning model. In Chapter 1, a brief but essential background was given on where genetic algorithms fit within artificial intelligence as a whole, as well as where they fit within the chosen problem-space of chess. Chapter 2 describes the chess engine that was built for this research project. Chapter 3 provides an in-depth analysis of the methodology of the two tuning algorithms, followed by a complete look at the results in Chapter 4. Chapter 5 describes further work, followed by an epilogue to fully encapsulate this research on a personal note.

2
The Creation of GERALD

Before delving into the architecture and results of using genetic algorithms for the purpose of chess engine parameter tuning, it is first necessary to build a chess engine. An extensive and detailed description of every consideration and feature of a modern chess engine is beyond the scope of this thesis, but it is still paramount to establish the fundamental architecture and problem space within which the genetic algorithm research will be conducted. Needless to say, the building of GERALD was hardly a small task, and three re-writes were necessary to achieve the final result. An initial Pythonic implementation was discarded due to the infeasibility of using an interpreted and slow language like Python for the compute-intensive task of a chess engine. A second re-write in the much more appropriate C++ was also mostly discarded due to the lack of a robust testing framework as each feature was developed (see Appendix A.3.2). Finally, a chess engine suitable for research purposes was created, GERALD: A Genetically Engineered, Researched, And Lovingly Developed chess engine.

2.1 The Mini-Max Search and its Enhancements

At the heart of GERALD, as well as most turn-based, zero-sum games (such as tic-tac-toe, Connect 4, checkers, chess, etc.) is the mini-max search. The mini-max search takes advantage of the relationship that a positive score for one side is the same score for the other, with the sign flipped (since both sides' scores always sum to zero). In this sense, it is typical in a chess

mini-max search for the side with the White pieces to try to maximize the score, while the side with the Black pieces tries to minimize the score. For terminal positions such as checkmate, a score is theoretically considered to be either positive or negative infinity, although in practice a large number is assigned. Based on this assumption and handling of terminal nodes, one can construct a tree of all possible moves and responses, and choose the current move that leaves the other side with the worst possible outcome. The depth of this search tree is defined as the number of moves and responses searched for each possible branch. For example, the root node will be searched with a depth of 0, all possible moves from that position a depth of 1, and all possible responses for each one of those initial responses a depth of 2. The average chess game is played in about 40 moves from each side, requiring a search depth of 80 to search through an entire average game (not to mention games that last longer than average, for which a much larger depth is needed). Since the search tree grows exponentially,[1] a naive mini-max search is impractical.

The quintessential improvement to the mini-max search is commonly referred to as alpha-beta pruning, which is a completely theoretically correct pruning mechanism to the mini-max algorithm. In this approach, let alpha represent the best possible score the player can achieve on their next move, and let beta represent the best possible score for the opponent. One can forego searching any branch of the search tree that will not improve alpha. In practice, this means that if the opponent has a strong response to a move, that move and all subsequent moves below it will be pruned, saving a huge amount of computation. As commonly used across many engines, including GERALD, a mini-max search with alpha-beta pruning is implemented using the Negamax function. The Negamax function allows one recursive function to work for both sides by constantly flipping the sign and values of alpha and beta. One final consideration of this function is that it is generally more precise to return the score that caused an alpha or beta cutoff, rather than the value of alpha or beta itself. This is known as fail-soft alpha-beta pruning. Pseudo-code is as follows:

[1] See Appendix A.3.3 with an example of this phenomenon.

Algorithm 1 Fail-Soft Negamax

```
1: function Negamax(alpha, beta, depth)
2: if depth == 0 or is terminal node then
3:    return evaluate(position)
4: end if
5: bestScore = -Inf
6: for each legal move in position do
7:    score = -Negamax(-beta, -alpha, depth -1)
8:    if score > bestScore then
9:       bestScore = score
10:      if score > alpha then
11:         alpha = score
            {Alpha-Beta cutoff has occurred}
12:         if score >= beta then
13:            return score
14:         end if
15:      end if
16:   end if
17: end for
18: return bestScore
```

2.2 Enhancements to the Alpha-Beta search

A further improvement to alpha-beta pruning is that the order in which moves are searched in the search tree is directly correlated to the number of branches that can be pruned. If the best move is searched first, most other moves can immediately be seen to not improve the result, since the maximum alpha in the position has already been found. Conversely, if the moves are searched in worst to best order, alpha-beta pruning does not save any time at all. Given this factor, chess engines often spend extra computational resources to find the principal variation, or the sequence of moves that consists of all the likely best moves and replies for both sides.

In GERALD, as with most engines, the process of finding the best moves to search first is done through a combination of intelligent move-ordering and an iterative deepening loop. The iterative deepening loop searches the position at depth k, followed by a search to depth $k+1, k+2, ..., k+i$, until depth N. While there is a significant overhead to searching the earlier depths multiple times, the information gained from these searches allows the overall search to be much faster, and it is more likely the move-ordering and other speed-ups will be optimal.

At each depth, the legal moves available are ordered in terms of likelihood to be the principal variation, and likelihood to speed up search by reducing the distance between alpha and beta. Typical move-ordering consists of placing the principal variation move first, followed by moves that represent captures or promotions as they are likely to be important. For the ordering of these moves, GERALD and many other engines use the Most-Valuable-Victim Least-Valuable-Aggressor heuristic(MVV-LVA for short) [10]. Lastly, moves that were deemed to be important in earlier shallower searches in the iterative deepening loop are examined in order of their calculated importance. GERALD implements a common combination of the killer-move heuristic and a history heuristic to choose the final order [10].

Positions analyzed in previous searches are stored in a hash table along with meta information like depth searched and best move found. This memory-time speedup is commonly referred to as a transposition table, allowing for positions that arise multiple times to not have to be searched again (a position can occur in multiple different orders of moves, and is known as a transposition).

The other essential improvements to the alpha-beta search also revolve around reaching an alpha-beta cutoff as fast as possible. As an overview, methods in this category artificially set alpha and beta before any cutoffs in the hope that this will cause cutoffs to occur at lower depths in the search tree. To do this, many chess engines, including GERALD, use a principal variation search (PVS). In a typical principal variation search, the principal variation move (or move that was found to be best in depth $k-1$) is searched with alpha and beta equal to their normal initial values of negative infinity and positive infinity. Every subsequent move at the initial root depth of zero, however, is assumed to not be able to improve alpha, since it was not found to improve alpha in the previous search of depth $k-1$. Because of this assumption, these moves are searched with what is a known as a zero-window. A zero-window is just when alpha is set to alpha + 1, and beta is set to alpha. Because all possible scores are integers, there is no actual ability for this search to return a score that is between alpha and beta. Instead, all scores are either less than or equal to alpha, or greater than alpha. In the case that they are less than or

equal to alpha, the assumption has held true, and the move did not improve the current alpha. If the search instead finds a score greater than alpha, then a full re-search with normal values of positive and negative infinity is necessary. Although re-searching may appear to be inefficient, this assumption that all moves after the principal variation will not improve alpha holds true enough that the principal variation search is an overall improvement in search speed.

Consider a concrete example, where the iterative deepening loop has just completed a search that reached depth 9 and has found a move that is best at that depth. When a search is initiated at depth 10, the engine wants to know if that move is still the best move. Any other move, however, is less likely to be the best move at depth k, since it was found to not be the best move at depth $k-1$. As the iterative deepening loop searches with a larger and larger depth k, the assumption of the best move at depth $k-1$ being the best move at depth k is more and more likely to be true.

Aspiration windows are sometimes optional but are usually helpful optimizations that sacrifice no 'correctness' in that they never risk returning an alpha value that is not actually the correct alpha, given the scoring system of the engine. Similar to the zero-window search, tightening the distance between alpha and beta causes the search to find more-cutoffs at earlier depths. Rather than do a zero-window search, however, one can do the principal variation search within a much smaller window, usually $[score + constant, score - constant]$, where $score$ is the best score found in the previous search. Once again, if a search returns a score outside of these bounds, then a re-search is necessary. GERALD, for the sake of simplicity, simply defaults to a setting alpha and beta to the conventional negative and positive infinity if the aspiration window's bounds are found to be too narrow.

The last type of optimization, often resulting in a more efficient search, is applying extensions and reductions to the search process. The most common extension is searching an extra depth beyond the current depth for all moves that put the opponent in check. The most common reduction, usually called a late-move reduction, is reducing the depth of searches conducted on moves late in the move order, as these moves are likely to be bad given that they were sorted

to the end in the move-ordering scheme of the engine. Once again, these moves are searched in a zero-window, and only in the case that they return a score greater than alpha is a re-search necessary.

The most essential extension mechanism is the quiescence search, which extends all searches once they reach a depth of 0. This search is intended to negate the horizon effect, where an engine may stop the search at a critical check or capture, leading it to be unaware of the ensuing danger. The quiescence search uses all the same speedups and optimizations of the main alpha-beta search, but only considers captures or, in some cases, promotions. GERALD employs more or less standard implementations of these extensions and reductions [10].

The important takeaway from this section is that using intelligent parameters in setting the various alpha and beta bounds will drastically increase the speed of a chess engine's search. The same can be said about finding optimal conditions on the extensions and reductions, so the search can quickly converge on a couple of promising branches of the search tree. On the contrary, bad values will result in many re-searches which is extremely detrimental to the chess engine's overall playing strength.

2.3 Pruning the Search Tree

Even with all of these search techniques, the search tree is still immense. As a result, more pruning of the tree is necessary. While the above methods for optimizing the search function are able to consider every possible move, these following methods sometimes sacrifice correctness for the sake of pruning branches of the search tree. However they result in a much stronger engine with a more powerful search.

Many pruning methods are based on the null-move observation. This observation states that, for most positions, playing a move is almost always better than passing the turn. Thus, if a player is to pass the turn without playing any move (the null move), and the other side is still unable to improve beta, then beta can be returned from the search with no further searching or calculations. This is the basis of one of the most powerful pruning methods commonly referred

to as null-move pruning. Of course, in order to pass the turn, it is necessary that the player not be in check, that null-move pruning was not applied in the previous move, and that the current static evaluation is already lower than beta. The most common pitfall of fully pruning a move rather than reducing the depth is that the null-move observation fails to consider a position in Zugzwang, or when the best move is to not move at all. These positions mostly occur in the endgame, however, and GERALD disables null-move pruning for all positions that have only kings and pawns left.

GERALD employs other pruning methods that also take advantage of the null-move observation such as razoring, reverse-futility pruning, and stand-pat pruning done in the quiescence search [10].

Further essential pruning methods are based not on the likelihood to decrease beta, but on the likelihood of a move to improve alpha. One example of this is delta pruning which is done within the quiescence search. If a capture plus the value of the piece being captured plus an arbitrary margin is unable to increase the current best score found, then the subsequent move and its branches are pruned.

By using these additional pruning techniques, GERALD is able to drastically reduce the effective branching factor of the search tree and, at many low depths, keep it from exponentially increasing.

2.4 Search Features of GERALD

For full transparency, the specific search techniques of GERALD are listed in Table 2.4. Because search features can be implemented differently in different chess engines, direct examination of the code as given in Appendix B will show GERALD's implementation. Furthermore, due to time constraints, some implementations were simplified, while other search features that are common in modern engines were omitted. These features are meant to be representative of a typical set, but not meant to represent a full state of the art approach.

Table 2.4.1: Search Techniques Used in GERALD

Fail-soft Alpha-Beta Negamax Search
Principal Variation Search
Aspiration Windows
Iterative Deepening
Transposition Table with Zobrist Hashing
Quiescence Search with Stand-Pat Pruning
Delta-Move Pruning
MVV-LVA Move Ordering
Killer-Move Heuristic and the History Heuristic
Null-Move Pruning
Razoring
Reverse Futility Pruning
Late Move Reductions
Late Move Pruning
Check Extensions
Simplified Time Management Scheme

Before discussing the evaluation function, it is important to credit the following open source chess engines which inspired or provided a basis for GERALD's implementation: SmallBrain [15], Raphael [23], Rice [26], Stormphrax [9], Sunfish [31], and Stockfish [29].

2.5 The Evaluation Function

Once the quiescence search is done sifting through the various potential moves, it returns a static evaluation of the resulting position. In this sense, an engine is never directly evaluating the position that one sees, but rather finding the maximum value of the all nodes on the frontier of the generated search tree. These leaf nodes are unlikely to contain critical captures or checks, thanks to the quiescence search and check extensions. Because of these qualities, it is easier to apply an accurate heuristic on a static position, informing the engine of the likelihood a side is going to win. If an engine searches one million nodes, then one million evaluations will be done, thus emphasizing the efficiency and speed with which this function must calculate its result.

As mentioned above, GERALD's evaluation function is a classical evaluation function done through a set of weighted handcrafted features. The final evaluation for any given position p for a color, c, is found with the following summation, where i is the index of the current feature, $w_{i,c}$ is the weight of the feature for c, x_i is the value found for that feature for p given c, and k is the length of the feature list:

$$Evaluation(p_c) = \sum_{i=0}^{k} (w_{i,c} \cdot x_{i,c} - w_{i,!c} \cdot x_{i,!c}) \tag{2.5.1}$$

A further essential detail is that GERALD uses a tapered evaluation, meaning each weight has two distinct values, one for a middle game position, and one for an end game position. Rather than assigning a hard cutoff to decide when to use which weight, the weights are 'tapered' in between the game states to allow for a more nuanced approach. Let g be a combined material score, and let m be the middle-game weight for feature f, and e be the end-game weight. The game phase ϕ is calculated from the tapered endgame score by:

$$\phi = \max(0.0, \min(1.0, \frac{g - 24}{24.0})) \tag{2.5.2}$$

where ϕ is constrained to be between 1 and 0 to ensure a smooth transition between the middle and end games. The weights for the middle game (m) and end game (e) are then combined based on the current game phase ϕ to compute the final score s for feature f as follows:

$$s = m \cdot \phi + e \cdot (1 - \phi) \tag{2.5.3}$$

Here, $m \cdot \phi$ represents the contribution of the middle-game weight scaled by the game phase, while $e \cdot (1 - \phi)$ represents the contribution of the end-game weight scaled by the inverse of the game phase. This tapered evaluation is inspired by the tapered evaluation found in the chess engine PeSTO [16]. For a full example, two positions are given in Figures 2.5.1 and 2.5.2, with corresponding values in a table in Figure 2.5.3.

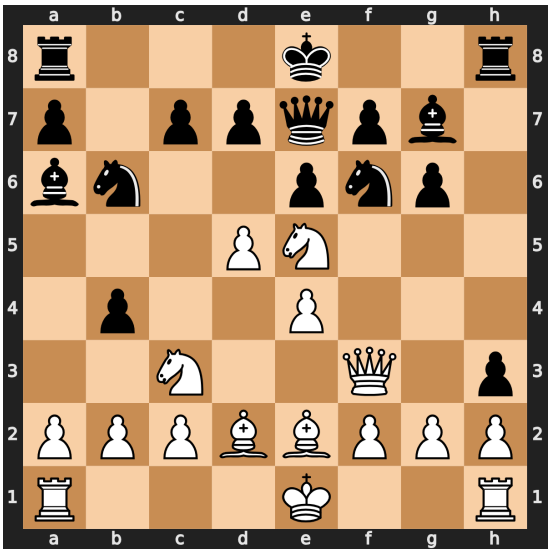

Figure 2.5.1: Position 1: A famous test position for chess engines known as kiwipete [10]

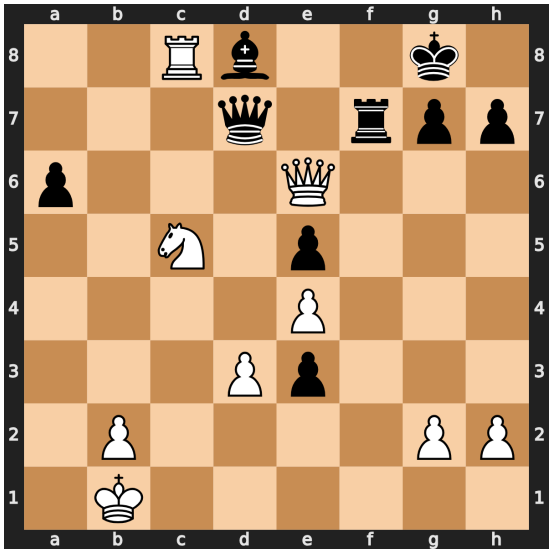

Figure 2.5.2: Position 2: A dynamic and imbalanced position

Parameter	Description	Pos. 1	Pos. 2
Game Phase	Based on number of pieces left 0 = endgame	1	.833
Pawn Value	Value of one pawn	0	0
Knight Value	Value of one knight	0	1
Bishop Value	Value of one bishop	0	-1
Rook Value	Value of one Rook	0	0
Queen Value	Value of one Queen	0	0
Passed Pawn Multiplier	Pawn with no other pawns in its way	0	-1
Doubled Pawn Penalty	Pawns stacked on the same file	0	-1
Isolated Pawn Penalty	Pawn with no adjacent pawns	0	-2
Backward Pawn Penalty	Pawn with all adjacent pawns advanced	0	1
Weak Square Penalty	Square in middle 4 ranks no friendly pawn can reach	5	-7
Passed Pawn Enemy King Square	Can King prevent passed pawn promotion	0	-1
Knight Mobility	Number of squares a knight can move to	2	5
Knight Outpost Mult.	Knight is defended by pawn on opp. weak square	0	0
Bishop Mobility	Number of squares a bishop can move to	3	-7
Bishop Pair	Possessing both colored bishops	0	0
Rook Attack King File	Rook is on same file as enemy king	0	0
Rook Attack King Adj File	Rook is on adjacent file to enemy king	0	0
Rook 7th Rank	Rook is on second to last rank	0	0
Rook Connected	Rooks can guard each other	0	0
Rook Mobility	Number of squares a rook has	-4	-4
Rook Behind Passed Pawn	Rook is behind a passed pawn	0	0
Rook Open File	Rook has no pawns in its file	0	0
Rook Semi Open File	Rook has no friendly pawns in its file	0	0
Rook Atck Weak Pawn Open Column	Rook can directly attack a weak enemy pawn	0	0
Queen Mobility	Number of squares a queen can move to	5	5
King Friendly Pawn	How many and how close friendly pawns are to king	-4	-3
King No Enemy Pawn Near	King is far away from enemy pawns	0	0
King Pressure Mult	Number of opp. attacks in kings zone	-6	20

Figure 2.5.3: Table of Position Features with Descriptions (for only White)

Since the example values are all from White's perspective, a negative value corresponds to Black having an advantage in a category, and a positive value corresponds to a White having the advantage. Note that, much like search features, handcrafted evaluation features differ from engine to engine. The source code provides a more complete and definite look into how each feature is calculated and contributes to a final evaluation.

I have now defined in more granularity the architecture and parameters that make up GERALD. With these definitions, it is now possible to discuss the process of 'learning' better sets of parameters through genetic and co-evolutionary algorithms.

3
Tuning GERALD

Chapter 2 provided an extensive description of GERALD, describing the kinds of parameters that can be tuned in the search and evaluation functions. Now we turn to the use of genetic algorithms to improve the search and evaluation processes. The search function, bounds, conditions, and margins will be tuned via a genetic algorithm to operate together and optimize GERALD's search function. In the evaluation function, the set of weights that correspond to the evaluation features from Table 2.5.3 will make or break the accuracy of GERALD's ability to estimate the score of a static position. These weights will also be tuned and optimized with a genetic algorithm.

Initially, during system development, baseline values were programmed into GERALD for each one of these parameters. These baseline values were determined mostly from my research of chess engine programming as well as my own intuition as an experienced chess player. They were also each tested individually and incrementally to make sure they contributed to the overall playing strength of the engine.[1] In this sense, the baseline values might not be optimal, but they are all positively contributing to the engine, suggesting they are at least somewhat reasonable. These baseline values were then subjected to two genetic algorithms, one for the search function, and one for the evaluation function. The final objective, and main focus of this project, was to

[1] See Appendix A.3.2 for the testing method used.

find a set of more optimal values that result in a better chess engine, as measured by increases in Elo rating.

Given the natural separation between the search parameters and the evaluation feature weights, a separate genetic algorithm was developed for tuning each. Since the parameters of the search function are dependent on the accuracy of the evaluation function, the evaluation function was tuned first, followed by the search. In the following section, an overview of genetic algorithms will be presented, followed by the specifications and differences of the two genetic algorithms used.

3.1 The Genetic Algorithm Loop

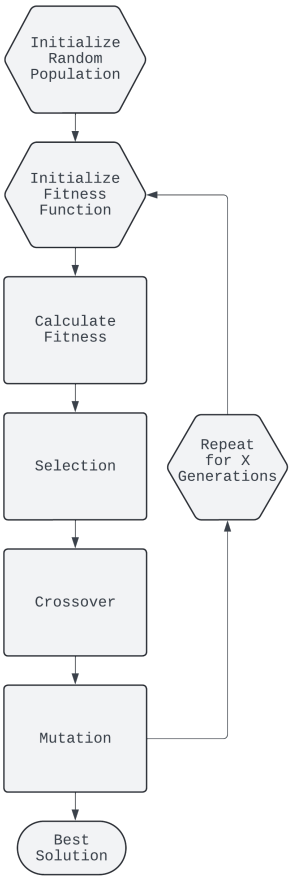

Figure 3.1.1: The Genetic Algorithm Loop

At the core of the genetic algorithm is the genetic algorithm loop, which usually runs for a set number of generations. This loop is presented in Figure 3.1.1. First, an initial population is created. Typically, a member of this population, which will be referred to as a chromosome, represents one possible solution to the environment of the genetic algorithm. The solution is directly encoded in the chromosome, where unpacking the genes of the chromosome gives a valid solution to the problem. The larger the population, the larger the number of possible solutions tested, and the more a genetic algorithm can maintain 'genetic diversity.' The chromosomes are ranked based on a fitness function, the function that more or less decides if a chromosome is 'fit' for its environment. While the fitness function of an actual environment might be the subject of philosophical debate, the fitness function of a genetic algorithm can be thought of as a scoring mechanism– the higher score, the better. It is based on this score that the other genetic operators, selection, crossover, and mutation, can function.

Over the course of one generation, the current population will be scored based on the fitness function, and then a criteria will be applied to select the best organisms. Sometimes this even involves duplicating high scoring chromosomes to increase their genetic representation in the next generation. Next, a crossover mechanism is applied to create new offspring that combine the genetic makeup of their parents. Not all selected chromosomes end up being parents, some pass directly to the next generation, while others are neglected entirely. Finally, the next generation's newly founded population is subjected to a mutation phase, where random bit flips in each chromosome's genetic makeup are applied to try to stumble upon random improvements. Every chromosome in the new generation is evaluated by the fitness function once more, and the loop repeats. In the subsequent sections, each stage of this process will be covered more extensively, and will refer to the exact implementation of the genetic algorithms used for GERALD's parameter tuning.

3.1.1 Chromosomal Encoding

Since both the evaluation feature weights and the search parameters of GERALD are no more than a set of positive integer values, they can be easily encoded in a binary string of ones and zeros. To discern which set of ones and zeros correspond to which parameter or weight, predefined boundaries are used. For example, in one algorithm the first seven bits are used to represent the value of a pawn in the middle game, while the last four are used to encode the king-pressure multiplier in the end game. Identical to the genetic algorithm used for the tuning of Falcon [13], chromosomes were encoded using Gray encoding. Gray encoding is done to attempt to solve what is know as the 'Hamming-cliff' problem in chromosome mutation [5]. Imagine, for instance, the ideal value for a specific 4-bit subsection of a chromosome is eight, represented in binary as 1000. Suppose the current value is seven, which has a binary representation of 0111. Four separate flips are necessary to reach the more optimal encoding of 1000, which is highly unlikely to occur in one generation in most genetic algorithms. It is also unlikely to occur over many generations because each intermediate value that results from any single bit-flip might be

worse for the overall fitness of the chromosome. Gray encoding attempts to alleviate this problem by ensuring the number of bit mutations between any two numbers one integer value away (i.e 7 and 8) is exactly one. Pseudo code for Gray-to-Binary and Binary-to-Gray conversions are provided below, and are taken directly from Chakraborty and Janikow's paper [5].

Algorithm 2 Binary-to-Gray Conversion

```
1: g_1 ← b_1
2: for i = 2 to n do
3:     g_i ← b_{i-1} ⊕ b_i
4: end for
```

Algorithm 3 Gray-to-Binary Conversion

```
1: b_1 ← bitvalue ← g_1
2: for i = 2 to n do
3:     if g_i = 1 then
4:         bitvalue ← COMPLEMENT(bitvalue)
5:     end if
6:     b_i ← bitvalue
7: end for
```

3.1.2 Selection

Once each chromosome in a population is evaluated based on a fitness function, a selection mechanism is performed, an approximation of Darwin's natural selection. The selection operation aims to choose the most fit individual chromosomes, while maintaining a reasonable amount of genetic diversity. Both genetic algorithms used for tuning GERALD employed proportional selection.[2] Proportional selection is done by summing up the total fitness of an entire population, and then randomly choosing individuals weighted by their 'proportion' to the total fitness. If one is trying to minimize the fitness function, this process can be inverted.

Elitism

Additionally, it is in the selection stage that elitism is implemented, where the top x individuals are automatically chosen without going through proportional selection. This is done so that the best solutions are not lost from generation to generation, and their high scoring chromosomal

[2]Proportional selection is also known as roulette-wheel selection.

representations are more likely to impact future populations. Choosing a good value for x is about finding the delicate balance between keeping good solutions whilst maintaining genetic diversity.

Historical Mechanism

On top of elitism, the genetic algorithms used for tuning GERALD implemented a historical mechanism, as inspired by Vázquez-Fernández et al. [32]. The historical mechanism is essentially just a dynamically-sized array (in C++ it is implemented as a Vector object) that keeps track of the top k solutions in the entire training process. The difference between a historical mechanism and elitism is that the historical mechanism maintains solutions between generations. This is done so that the most promising solutions are preserved, even if they end up being mutated or combined with less promising solutions later. Similar to elitism, the top y chromosomes from the historical mechanism are passed on to crossover without undergoing proportional selection.[3] Both the historical mechanism and elitism aim to help speed up the genetic algorithms ability to converge upon a good solution.

3.1.3 Crossover

After selection is finished, pairs of individuals are randomly chosen from the new population for crossover. A crossover rate is assigned prior to training, and each pair is given a random chance for crossover, or for directly passing on to the new generation based on this rate. For example, a crossover rate of .8 signifies that 80 percent of the time two randomly selected individuals will indeed combine their chromosomes to create two new offspring, while 20% of the time both parents will directly pass on to the next population. This allows for less genetic drift between generations, since a certain percentage of the population will stay the same. While there are many different methods available for crossover, both genetic algorithms employed in the tuning process use single-point crossover, a fairly common and straight-forward crossover method. Single-point crossover refers to the process of creating a new chromosome with 0-k

[3]Both x and y are hyperparameters, are were found to perform best when they were very small compared to the overall population size.

genes from parent one, and k-l genes from parent two, where k is a random number, and l is the length of the encoded chromosome.

3.1.4 Mutation

The final step in the genetic algorithm loop is the mutation step. Mutation tries to simulate the seemingly random occurrence of genetic mutation that occurs in nature and in the passing on of genes. Much like in nature, most mutations are neutral or harmful, but some allow novel or more optimal solutions to be found. Once again, a mutation rate hyperparameter, p_m is defined at the start of training, and each bit in each chromosome that is not elite is flipped with a p_m probability. Typically, the mutation rate is a very small probability to allow for the overall 'goodness' of a chromosome to likely remain intact. It is in this step that the aforementioned Gray encoding is helpful as mutations are less likely to drastically alter a chromosome's encoded solution.

3.2 The Evaluation Genetic Algorithm

The first genetic algorithm applied to GERALD is designed to tune the weights of the hand-crafted features in the evaluation function. The genetic algorithm follows the shell from the previous section, with proportional selection, elitism, a historical mechanism, single point crossover, and a constant mutation rate. Table 3.2.1 gives the exact breakdown of the bits in chromosomal encoding, totalling to chromosomes of 356 bits.

This section will focus on the aspect least covered previously, the fitness function. The issue with many genetic algorithms is that the fitness function can act as a bottleneck to speed. Using genetic algorithms in the problem-space of chess presents the same problem, since it is difficult to properly estimate the fitness of two similar chromosomes without playing thousands of games. Simulating thousands of games for each chromosome, for each generation, results in an explosion of necessary computing resources and is quite unrealistic for most researchers. The chess application of genetic algorithms can benefit from the ways this problem was tackled in

Feature	Bits	Feature	Bits
Pawn MG (fixed)	0	Knight MG	9
Pawn EG	8	Knight EG	9
Bishop MG	9	Rook MG	10
Bishop EG	9	Rook EG	10
Queen MG	10	Passed Pawn MG	6
Queen EG	10	Passed Pawn EG	6
Doubled Pawn MG	6	Isolated Pawn MG	6
Doubled Pawn EG	6	Isolated Pawn EG	6
Weak Pawn MG	6	Central Pawn MG	6
Weak Pawn EG	6	Central Pawn EG	6
Weak Square MG	6	Passed Pawn Enemy King Sq MG	6
Weak Square EG	6	Passed Pawn Enemy King Sq EG	6
Knight Outposts MG	6	Knight Mobility MG	5
Knight Outposts EG	6	Knight Mobility EG	5
Bishop Mobility MG	5	Bishop Pair MG	6
Bishop Mobility EG	5	Bishop Pair EG	6
Rook Attack King File MG	6	Rook Attack King Adj File MG	6
Rook Attack King File EG	6	Rook Attack King Adj File EG	6
Rook 7th Rank MG	6	Rook Connected MG	6
Rook 7th Rank EG	6	Rook Connected EG	6
Rook Mobility MG	5	Rook Behind Passed Pawn MG	6
Rook Mobility EG	5	Rook Behind Passed Pawn EG	6
Rook Open File MG	6	Rook Semi-Open File MG	6
Rook Open File EG	6	Rook Semi-Open File EG	6
R Attack P Open File MG	6	Queen Mobility MG	3
R Attack P Open File EG	6	Queen Mobility EG	3
King Friendly Pawn MG	6	King No Enemy Pawn Near MG	6
King Friendly Pawn EG	6	King No Enemy Pawn Near EG	6
King Pressure Score MG	4	King Pressure Score EG	4

Table 3.2.1: Number of bits for each evaluation feature weight (MG = middle game, EG = end game

the tuning of other classical evaluation functions via genetic algorithms, discussed next. Then a full process of the genetic algorithm that was applied to GERALD will be described, followed by the final result.

3.2.1 Previous Research

The solution presented in David-Tabibi et al. [13] was to use an "expert" evaluation function known to be of high caliber, and base the fitness function on the difference between the value

of the expert and the value of a given chromosome. This value is a 'centi-pawn' value, a unit meant to represent one one-hundredth of a pawn's value. A set of chess positions were given to the expert, a score was recorded, and each chromosome's fitness was simply the average loss between the expert's centi-pawn score and the chromosome's score over a subset of positions. Since the evaluation step takes milliseconds at most, and the expert can pre-process a set of chess positions prior to any run, this approach is extremely efficient and fast.

In 2010, David-Tabibi et al. tried a slight variation of their original idea. The researchers maintained the use of an expert but, instead of finding a difference in centi-pawn value, they based the fitness function around the number of identical moves found for the same position. A similar set of positions was given to an expert system, but this time the expert chose the best move rather than a score. Each chromosome evaluated the possible moves from the root position, and the move that resulted in the highest evaluation score was compared to the expert's move. If they matched, the fitness score of the chromosome was incremented. Vázquez-Fernández et. al [32] used an identical method in their paper. The advantage of this method was that the expert did not need to provide an exact score, and thus the choice of moves from human Grandmaster players or any high level chess game could be used. The disadvantage, however, was that this was a significantly slower process, since each chromosome needed to iterate through all the legal moves of each position. Furthermore, the fitness function was able to determine only if a move was found that was identical to an expert's choice, rather than have a granular 'distance' between centi-pawn scores.

In all of the above research, the set of positions that were used in the fitness function were taken randomly from a database of human Grandmaster games. While there is an argument for using naturally occurring positions in human games, this dataset is not necessarily representative of the set of positions a chess engine's evaluation function will be applied to. For one, chess engine's have a main search and a quiescence search, so they almost never evaluate a position with relevant captures or tactics. Furthermore, games between chess engines vary from games played between humans, so the set of positions may not be fully representative.

The following subsections describe how these concerns were addressed in the evaluation function of GERALD, as well as provide a full overview of the training process.

3.2.2 Choosing an Expert Fitness Function and Building the Data Set

Similar to the importance of the quiescence search which provides the evaluation with a quiet position, static evaluation tuning is typically done on quiet positions. As mentioned above, most research thus far used a set of random positions from a database of Grandmaster games. As an improvement, a set of quiet positions used for the tuning of a stronger engine, Zurichess [24], were used. These positions are the leaf nodes of a strong engine's search, and so are more representative of the kind of positions GERALD's evaluation function would be applied to.[4]

Because a set of quiet positions were used, and because these positions did not occur in natural play, the more granular comparison of centi-pawn evaluations was deemed to be more suitable. To acquire this expert evaluation, many experts were tried, including the more modern neural network based evaluation functions present in Smallbrain [15] and Stockfish 16 [29].[5] However, I found that a neural network evaluation does not act at all like a handcrafted evaluation. In fact, most modern neural networks are able to detect moves that would otherwise require a deep search to find, making them not ideal 'experts' for a handcrafted evaluation function.

After much trial and error, I chose Stockfish 11 as the expert [29]. The reason for an older version of Stockfish is that Stockfish 11 was the last version of Stockfish to use a classical handcrafted evaluation function. Along with seamless integration with python-chess [25], Stockfish 11 is one of the strongest classical chess engines ever made, and thus serves as a great expert (and it is open-source). A set of 8,000 quiet positions from Zurichess' data set were used for training. Each position was given to Stockfish 11, and Stockfish 11 was limited to just a one node search (effectively just the evaluation function). The resulting centi-pawn evaluation was stored in an adjacent column to a character string representing the position in Forsyth-Edward

[4]These positions were actually used to tune a neural network evaluation function in Zurichess, but were found to be satisfactory for the purposes of handcrafted evaluation tuning as well.

[5]Modern Chess Engine's dwarf even the best human chess players in playing strength, so they were preferred experts.

Notation [7] (See Appendix Section A.1.1 for more on Forsyth-Edward Notation). The fitness of any single chromosome, c_i, in the population is found by first selecting 1,600 random entries, k, from the 8,000 total, and finding the difference between actual evaluation e_a and calculated evaluation e_c for each entry, and then finding the average difference among the given k entries. A succinct summation captures this:

$$\text{Fitness}\, c_i = \frac{\sum_{i=0}^{k} |e_a - e_c|}{k} \tag{3.2.1}$$

3.2.3 Training Results

With the fitness function fully finished, the full genetic algorithm was run. After a great number of experiments with various hyperparameters and training sizes, the final hyperparameters that resulted in the lowest training loss were as follows: Figure 3.2.1 shows the fitness of

Mutation Rate	: 0.002
Crossover Rate	: 0.75
Total Generations	: 300
Population Size	: 1000
Training Size	: 1600
Elitism	: 0
Historical Mechanism	: False

the average and best chromosome over the course of the 300 generations. The exact time for a parallelized run of the genetic algorithm was only 222 seconds, and even a single core would likely complete a training run of 300 generations with the above hyperparameters in under an hour. The baseline fitness was found by having the manually tuned evaluation feature weights perform on a set of 1,600 positions from the data set. Noticeably, after less than 50 generations the best individual is 25 centi-pawns better from the baseline, although even the all time best fitness value is still in the ballpark of 160 centi-pawns average loss from Stockfish 11. This is still reasonable, since 28 features are unlikely to reproduce the same value consistently as the hundreds of parameters and features used in Stockfish 11's evaluation function.

The set of final tuned evaluation parameters are outlined in Figure 3.2.2. It is interesting to note that the tuning process learned traditional piece-values from a random initialization, and also produced many values that vary highly from the original guesses.

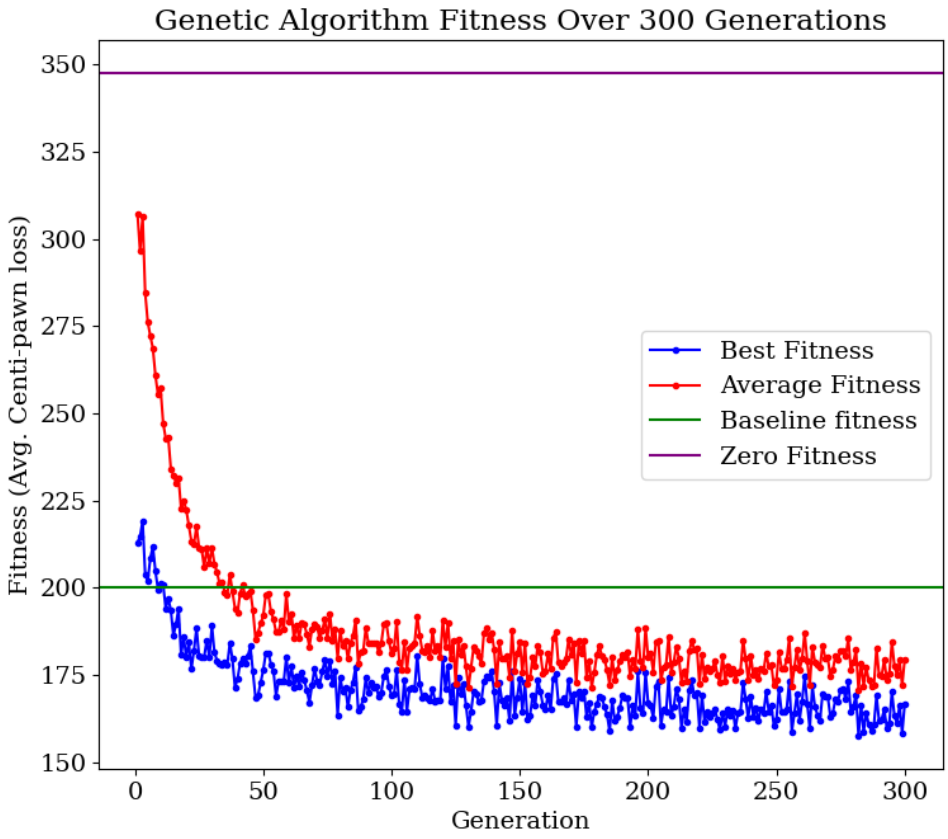

Figure 3.2.1: Training Results

Feature	Base Midgame	Tuned Midgame	Base Endgame	Tuned Endgame
Pawn	100	100	100	89
Knight	305	330	300	226
Bishop	315	359	320	307
Rook	480	506	520	479
Queen	910	1021	910	991
Passed Pawn	40	4	80	42
Doubled Pawn	10	20	20	45
Isolated Pawn	10	20	20	15
Weak Pawn	40	3	40	18
Central Pawn	10	14	10	3
Weak Square	5	1	2	3
Passed P Enemy K Sq.	0	0	50	2
Knight Outposts	40	3	35	28
Knight Mobility	1	12	2	15
Bishop Mobility	3	6	1	5
Bishop Pair	15	39	20	60
Rook Attack K File	15	22	5	1
Rook Attack K Adj File	10	10	5	10
Rook on 7th Rank	25	2	35	27
Connected Rooks	20	22	10	41
Rook Mobility	2	1	1	6
R Behind Passed Pawn	10	1	35	14
Rook Open File	15	45	5	7
Rook Semi-Open File	20	26	10	10
R Attacks Weak P	35	45	30	61
Queen Mobility	1	1	1	4
King Friendly Pawn	4	11	0	4
K No Enemy Pawn	5	4	0	0
King Pressure Score	3	2	1	0

Figure 3.2.2: Comparison of Base and Tuned Values

Based on these values, some observations can be made: 1) the original value of a bishop pair advantage was grossly underestimated; 2) the queen is worth the value of about ten pawns compared to the conventional wisdom of nine. Still, the true test of whether this tuning process was successful is if it improves the playing strength of GERALD as a whole. Table 3.2.2 outlines the results of playing 1,000 games between an instance of GERALD with manually tuned evaluation feature weights, and an instance with the newly tuned ones: The result of self-play is that the

Category	Results
Score of tuned eval vs base	538 - 236 - 226 [0.651]
Tuned eval playing White (500 games)	287 - 108 - 105 [0.679]
Tuned eval playing Black (500 games)	251 - 128 - 121 [0.623]
White vs Black (1000 games)	415 - 359 - 226 [0.528]
Elo difference	108.3 ± 19.6
LOS	100.0
Draw Ratio	22.6%

Table 3.2.2: Comparison of Tuned and Base Evaluations in Chess

newly-tuned GERALD is undoubtedly better, by a margin of about 108 Elo. In practice, this means that the new set of values beat the old a staggering 65% of the time. Closer examination of a couple of the games played between the two instances of GERALD show clearly why such a dramatic result was achieved. In most positions, the two instances disagree about the centi-pawn score of the position. And, in most cases, the newly tuned instance is closer to the 'truth'. The next task, after a much better set of evaluation feature weights have been found, is to tune the search parameters of GERALD given this update.

3.3 The Search Genetic Algorithm

While a myriad of learning techniques have proven successful for the task of tuning the parameters of an evaluation function, it is much more difficult to tune the parameters of a search function. There are many reasons for this increase in difficulty: First, searches take much longer than single evaluations, and thus there is an exponential time increase to receive a score or a result.[6] Second, search parameters are a lot more finicky, and bad values can sometimes even result in infinite or broken searches. Imagine, for example, if a margin of zero was applied to reverse-futility pruning [10]. The majority of all moves and the branches below them in the search tree would be immediately pruned, preventing any valid move from being searched. Lastly, search parameters are extremely co-dependent, and changing one value can have a cascading effect on the entire engine. Still, chromosomes were encoded in an identical Gray-encoded fashion as the evaluation genetic algorithm, as outlined in Table 3.3.1. The main difference is that the chromosomal encoding use significantly fewer bits, only 64, since the search parameters are fewer and smaller overall. In general, the shorter the chromosomal encoding, the better a genetic algorithm will perform, as there are fewer possible solutions to explore.

[6] As mentioned previously, millions of evaluations are calculated in one single search.

Parameter	**Bits**	**Parameter**	**Bits**
Aspiration Window Initial Delta	6	Use Lazy Eval Static	1
Use Aspiration Window Depth	3	Futility Margin	8
Razoring Margin	8	Delta Margin	10
Killer Move Score	10	Initial Depth LMR	3
Initial Move Count LMR	3	LMP Move Count	4
Null Move Pruning Initial Reduction	3	Null Move Pruning Depth Factor	5

Table 3.3.1: Allocation of bits for each search parameter

3.3.1 Previous Research

The only research that was found where a genetic algorithm was implemented for the purposes of tuning a chess engine's search function was done by some of the same authors who pioneered the method for the evaluation function in 2008. In their following chess engine research, they proposed a full pipeline for tuning a chess engine's search and evaluation functions from a random initialization [12]. The genetic algorithm they proposed was problematic, however, as even less evidence of its efficacy was given, and the fitness function was based on how many middle game puzzles the engine solved. Specifically, the fitness made a couple of key assumptions: 1) that solving more middle game puzzles was directly correlated to a stronger search and, 2) that solving such problems with fewer nodes examined per search is better than more nodes examined per search. The first assumption has been disproved, as shown in a Wiki page in RubiChess' engine GitHub page [21]. Most likely this is because a search is conducted on every imaginable position, and not just positions where one move is really good and the rest are sub-optimal as is the case with chess puzzles. The second assumption also fails for a similar reason, since fewer nodes can sometimes mean good moves are too quickly being pruned away. Thus, for the creation of the genetic algorithm for tuning the search function of GERALD, only fitness functions that considered overall play of the engine were considered.

3.3.2 Adding Co-Evolution

One known mechanism to enhance genetic algorithms is the introduction of co-evolution. In a typical genetic algorithm scheme, the only 'evolving' entities are the individual chromosomes in the population. A better reflection of nature, as well as a better machine learning algorithm, is possible when the fitness function is evolved along with the population, resulting in a more accurate convergence. Co-evolution also allows there to be unsupervised learning in a genetic algorithm, since perhaps the optimal strategy may not be known before hand.

For the game of chess, as well as any competitive game with a clear outcome in the end, there is an obvious and theoretically 'pure' implementation of co-evolution. For this we use as the fitness function the performance of each chromosome playing the rest of the population in a giant round-robin tournament consisting of hundreds of thousands of games. The fitness function will inevitably evolve as the field produces better players, and the exact fitness that is determined of each chromosome is inscrutable. Unfortunately, the main downside to this approach is the immense computing power required. Imagine a population of one hundred chromosomes – each chromosome would need to play each other chromosome in a deterministic match consisting of many games. This would require possibly thousands if not millions of games each generation.

To combat this issue, but maintain its theoretical soundness, I designed a modified version that I call King-of-the-Hill co-evolution. Rather than have each chromosome play every other chromosome, they all just play the current best chromosome. If any are successful in proving that they are better, they become the new 'King-of-the-Hill' for the next generation. This greatly reduces the number of games required per generation, while still allowing the fitness function to consider the overall playing strength of each chromosome.

The question still remains, however, as to how one can know exactly when two very similar chess engines differ significantly in playing strength. For this, a metric known as likelihood of superiority (or LOS for short) was used. LOS is used in almost every instance of chess engine testing. For example, a popular chess engine testing site, CCRL [3] posts a LOS value in-between every adjacent chess engine in their standings. Luckily, a LOS table can be found at the Chess

Programming Wiki [10], and so the math did not need to be implemented. LOS works on a net scoring system and assumes a constant rate of draws (32% draw-rate in this case). A win scores one point, and loss negative one, a draw zero. The final net-score is meant to represent how many wins over losses one side achieved. The higher the net-score, the higher the likelihood of superiority.

After some experimentation, a LOS value of 99.99% for a net score of +43 in 200 games was found to be the perfect balance between ensuring that a new chromosome was better than the current King-Of-The-Hill, while still allowing for the difference in playing strength between two engines to not be large. In fact, the more games played, the more one can be confident that a small margin of victory is significant with the LOS metric.

Despite the King-of-the-Hill speedup, time and computing power is still a factor. Great care was taken to fully utilize the computing resources that were available. The research computer that was used for this thesis had twenty cores, a population size of twenty was assigned, and each chromosome played the King-of-the-Hill chromosome in a match of 200 games in parallel, each generation. In the games, a hard limit of 200 total moves was set (before a game is adjudicated as a draw), and a search time of 50 milliseconds was given to each move for both sides. To introduce a wider variety of positions and get rid of repeat games, a random eight move opening was assigned to the starting position of each game from an opening book. The creation of this testing setup was not trivial, and command-line tools such as Valgrind were invaluable in ensuring no memory leaks occurred in the running of the algorithm.

The hyper-parameters used are listed in Table 3.3.2. Note, that to cater to the drastically smaller population size of 20, elitism was decreased to a value of 2 elites per generation, and the mutation rate was increased to have a larger impact on each chromosome.

From start to finish the genetic algorithm took 227038 seconds, or 2.6 days. The king of the hill was initially set to be the base manually chosen search parameters, and the king of the hill was updated twice in the training process, on generation 77 and 97.[7] Although colossally slower

[7]It was ultimately the values from the generation 97 update that were used in final testing. The algorithm could have stopped in half the generations since no more king of the hill updates were done.

Mutation Rate	: 0.05
Crossover Rate	: 0.75
Total Generations	: 200
Population Size	: 20
Elitism	: 2
Historical Mechanism	: True
Hist. Individuals Re-introduced per Gen.	: 1

Table 3.3.2: The Hyper-Parameters of the Search Function Genetic Algorithm

than the evaluation function, the ability for the algorithm to be parallelized means that the speed of the algorithm can scale with computing resources. The graph presented in Figure 3.3.1 tracks the best, average, and worst fitness of the population over the 200 training generations. Unlike the evaluation genetic algorithm, the best and average fitness was sporadic. This is most likely attributed to the much smaller population of 20 (versus 1,000 previously), as well as the more drastic effect that bad search parameters have on the playing strength of GERALD. This effect is better seen in the stochastic nature of the worst fitness graph, where the worst fitness jumps from -30 to -200 in just one generation. This is also a result of the extreme effect that bad crossovers or mutations can have on the overall playing strength of any single solution.

A table of the base values compared to the tuned values is also provided below. As with the evaluation feature weights, the best way to understand what each one pertains to is to see them in the code (found in Appendix B).

Feature	**Base Search**	**Tuned Search**
Aspiration Window Initial Delta	25	53
Aspiration Window Initial Depth	5	0
Use Lazy Eval for Static Pruning	True	False
Futility Margin	100	75
Razoring Margin	150	167
Delta Pruning Margin	300	194
Killer Move Score	100	841
Initial Depth for Late-Move Red.	3	3
Initial Move Count for Late-Move Red.	3	5
Move Count for Late-Move Pruning	10	8
Initial Reduction for Null-Move Pruning	2	2
Null-Move Pruning Depth Factor	10	9

Figure 3.3.2: Comparison of Base and Tuned Values

Some notable observations are that the tuned values are by and large more aggressive in pruning and reducing the depth of most branches in the search tree. For example, the delta

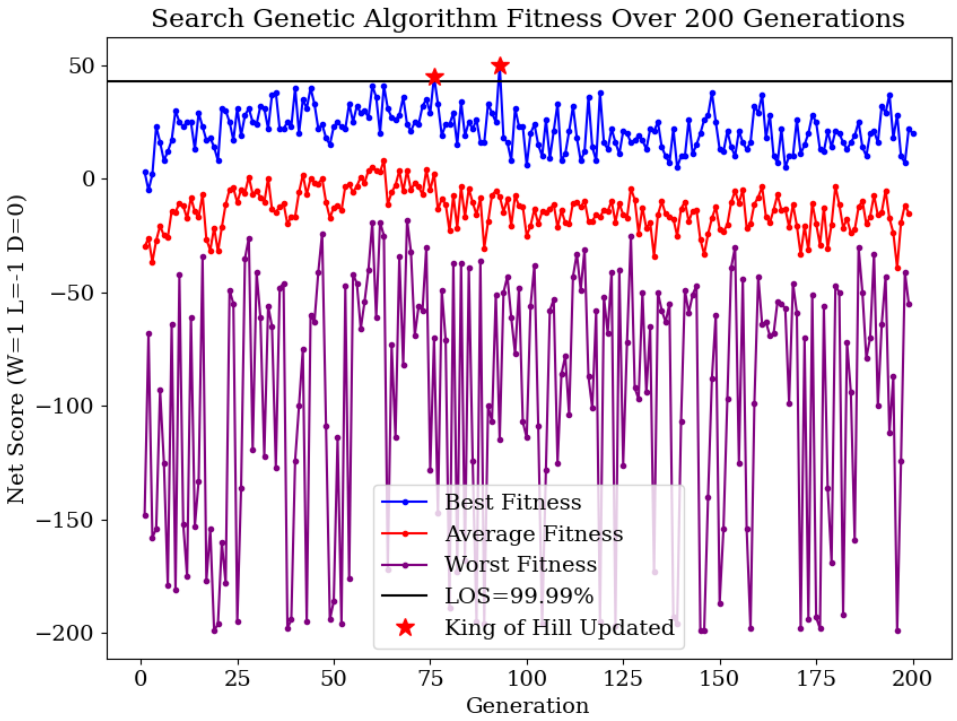

Figure 3.3.1: Training Results

pruning margin was reduced by more than 100 centi-pawns, the futility margin was reduced by 25%, and late move-count pruning was applied two moves earlier in. Also, the killer move score is much larger than initially anticipated, probably resulting in a higher placement of 'killer moves' in the move ordering schema of GERALD. The other huge difference is the tuned values opt to apply an aspiration window right away, but make that window twice as large. This is not as typical with other chess engines, but it seems to work well for GERALD. Once again, the true merit of this tuning process will be found only in actual games, so a round-robin tournament was conducted of 6,000 total games between instances of GERALD with the original manually-tuned values, the evaluation tuned values, and both the search and evaluation functioned tuned values. The results of this tournament are shown in Table 3.3.3. Based on the Elo increase of approximately 62 points from the evaluation tuned instance, and an increase of 230 points from the base instance, the genetic algorithm with King-of-the-Hill co-evolution is effective. The

Rank	Name	Elo	+/-	Games	Score	Draw
1	Tuned Search & Eval	96	14	2000	63.5%	24.2%
2	Tuned Eval	34	13	2000	54.9%	25.7%
3	Base	-134	14	2000	31.6%	21.8%

Table 3.3.3: Performance in Self-Play

increase was not as dramatic from the base instance to the evaluation instance as it was for the first expert-driven genetic algorithm, but it is nonetheless significant.

4
Results

To further test the full results of both tuning methods on GERALD, a match was played between a third party chess engine and the three GERALD versions – the base version, the version with a tuned evaluation function, and the version with both tuned evaluation and search functions. The third party chess engine used was Stash-bot [18], which proved a perfect opponent because its 12th release has an official rating in close proximity to GERALD's estimated rating (Stash-bot is rated as 1886 Elo on CCRL at the time of writing [3]).

Rank	Name	Elo	+/-	Games	Score	Draw
0	stash-12.0	21	8	6000	53.0%	12.3%
1	Tuned Search & Eval	42	14	2000	56%	10.7%
2	Tuned Eval	2	14	2000	50.2%	14.0%
3	Base	-108	15	2000	34.9%	12.4%

Table 4.0.1: Performance against Stash-bot

As seen in Table 4.0.1, the hierarchy of the three GERALD instances holds up well even against an external opponent. This further indicates that the two tuning processes helped strengthen GERALD's playing strength overall, and not just in self-play or testing. From this we can conclude that the two proposed genetic algorithms provide serious and promising alternatives to existing methods of chess engine tuning. In the first genetic algorithm, there was tremendous ease and speed in the training process. In the second, a more complex and lengthy training

process was required, but was nonetheless effective. The second search parameter tuning genetic algorithm is also especially relevant amidst the sparse alternatives found in research.

More importantly than chess, however, the results of this thesis provide evidence for the overall effectiveness of the two proposed genetic algorithms in similar problem spaces. In fact, many search spaces might be able to benefit from or apply one or both of the genetic algorithms. In many applications of artificial intelligence, a computer is meant to recreate human expert results.[1] In fact, any supervised learning model has a built-in 'expert' labelling data or judging results. The evaluation genetic algorithm effectively reverse-engineered an expert's results without having access to the same architecture as the expert. Furthermore, using a King-of-the-Hill co-evolutionary algorithm is plausible in any problem where various solutions can be directly compared against one another. Realistically, if two solutions cannot be directly compared, a genetic algorithm probably will not be ideal anyways since it is unlikely a good fitness function exists for that type of problem.

In sum, GERALD, a research engine built in C++, was greatly enhanced in both the evaluation and search components through the tuning of two genetic algorithms applicable to a large number of use-cases. The Elo rating increase was estimated to be about 150 Elo in both self-play and in external testing, putting an engine built by an inexperienced chess programmer in less than four months well above the majority of human play in the world (see Appendix A.3.4).

[1] Such examples include diagnoses to patients, natural language processing, or recommendation algorithms to name a few [8].

5
Further Work

While two genetic algorithms were successfully applied to improve GERALD's playing strength, it remains to be seen how these techniques would service a much stronger engine. Further, other state of the art tuning methods were not applied to the base version of GERALD, and thus there is not an accurate analysis of whether the genetic algorithms were an improvement over existing non-genetic tuning solutions. In future research, testing multiple tuning techniques on a set of several stronger engines would provide a more definitive view of the potential of genetic algorithms in the context of chess engine parameter tuning.

More broadly, this work does not explore how expert-driven genetic algorithms or co-evolution methods compare to other known methods for the broad use cases of artificial intelligence. Applying genetic algorithms to a variety of similar problems would give much more insight into their overall power as a machine learning tool.

As for the development of GERALD, the process of developing a chess engine is endless: new features, tweaks, better tuning, and code optimizations have potential to increase the engine's playing strength. As such, There are many optimizations and potential changes for the increase of GERALD's baseline playing strength. Essential improvements include the optimization of extracting position features for a position (known improvements include optimizing code, pawn hash tables, caching data, and selective updating based on move played), and the addition of

search techniques such as multi-cut pruning or internal iterative deepening. This was partially neglected because accuracy was prioritized over speed in the short development period of GERALD. Further, the game of chess is solved for all board states with seven pieces or fewer, and strong engines use lookup methods to play out known endings. Access to such tables would certainly increase GERALD's strength in the endgame. Further, GERALD does not use any piece square tables whatsoever which are usually a hallmark of effective handcrafted evaluations.[1] Including these, as well as perhaps incorporating them into the existing tuning mechanism, would likely result in a significant increase in playing strength and positional understanding. Future readers may see some of these changes implemented, and they may not. Perhaps the lessons learned from this thesis will aid the development of a new chess engine. Either way, the creation of GERALD marks the beginning of my foray into computer chess, not the end.

[1]The main reason for their absence is to keep the chromosomal encoding of the evaluation features minimal.

6
Epilogue

There are many aspects of this project that deserve their own section or chapter. I have tried to provide resources and references wherever possible so the curious reader can get answers to burning questions that were not directly addressed here. One of the aspects of this thesis that did not fit within the traditional thesis framework was a discussion of the challenges faced, and the lessons that were needed to overcome them. I will break this discussion into three categories: 1) the challenges of writing good bug-free software, 2) the challenges of explaining an immensely complicated algorithm or process in layman's terms, and 3) the existential challenge of writing and coming to terms with artificial intelligence.

Writing code that works has many challenges. It is not an accident that the Appendix has its own section for chess engine testing.[1] The first full version of GERALD had almost no automated testing. I simply would make a change to the chess engine's code, and then manually play against the engine and somewhat subjectively judge if the change was beneficial or harmful. Not only was this bad practice, but it became increasingly difficult to debug once all the obvious improvements were made. The most important automated tests are outlined in the Appendix, but the fundamental lesson of having a foundation of good testing was harshly learned.

The other challenge with writing bug-free code for this project was that both GERALD and the five tried genetic algorithms utilized multi-threading. Race-conditions needed to be dealt

[1] See Appendix A.3.

with correctly, and it was important to make sure the immense amount of memory required didn't raise any SEGFAULTs (20 cores running in parallel means a lot of memory being used). In one instance, I spent a week debugging a memory leak that turned out to just be an uninitialized variable. Still, learning to better debug multi-threading or memory issues with tools such as Valgrind is an essential skill for any low-level programmer, and I am grateful I had the opportunity to develop my knowledge in this area.

Challenge number two was actually writing this thesis, not because writing a senior thesis is generally arduous (it is), but because it is difficult to write about highly specific and technical niches. In many cases, using niche-specific terms can alienate a reader, or prevent general understanding. Finding a balance between brevity and clarity requires great care, especially when neither are particularly easy to achieve. Most people do not have an extensive knowledge of programming, and, of the people who do, very few have a strong foundation in classical chess engine architecture. An even smaller slice of this population might also be aware of the nuances of genetic algorithms. As a result, there was a huge emphasis on just trying to explain what was done in this thesis. A general background of computer science and a basic understanding of the game of chess was assumed, but I have no doubt that friends and family who do not meet this assumption and are reading this might be profoundly confused by most sections. Nonetheless, I take great issue with research papers that hide between esoteric words and verbose descriptions. A sign of true mastery, is the ability to explain complex subjects with plain and simple language.

The final challenge became apparent to me in the middle of a long road trip to one of my basketball team's away games. I was adding a new feature to an early version of GERALD, and began testing it on the team bus. Thirty-five moves later, I resigned. I felt excited, accomplished, but mostly... depressed. In just a month of development, the executable file I compiled surpassed my own ability in a game I take great pride in playing. Of course, this was not the first time artificial intelligence has forced me to take a hard look in the mirror. It was however, the first time something I had personally created did. Over the course of writing and programming this thesis, I have been forced to rethink what it is that makes my intelligence as a human worthwhile.

Genetic algorithms drastically outperformed my manually tuned baseline, and GERALD is now much much better than I am at chess. Not to mention, artificial intelligence in the form of Chat-GPT 4 [22] and GitHub's Co-Pilot [17] provided massive help in the form of a virtual assistant. Perhaps this human replacement is an example of the broader existential threat many people feel with artificial intelligence – what happens when artificial intelligence is just simply better than us at our jobs, at solving problems, and at making decisions? Isn't that what we, as humanity pride ourselves in?

I believe that, along with raising these questions, the process of creating this thesis was able to point me in a promising direction. The fundamental disadvantage of most genetic algorithms is that they are very unlikely to converge upon an optimal solution. I believe this is by design, however. The process of evolution is an inherently imperfect one. It creates organisms that are 'good-enough' to survive their environments and adapt to their surroundings. But to say that these organisms are perfect would be ludicrous: just look at the average human, and try to argue that we are examples of perfectly evolved organisms. Our humanity was created through evolution – and if genetic algorithms have taught me anything, it is that our strange mutations, our even stranger choice in mates, are not guided towards anything perfect. Hopefully, artificial intelligence will teach us not to fear our lackluster abilities, but take pride in our unique 'humanness'. It is our inefficiencies, our flaws that make us who we are, and what keeps games like chess so interesting to witness and play.

Appendix A
The Current State of the Art for Chess Engine

A.1 Board Representation

The fastest method by far for handling the internal logic and representation of chess in a chess engine is to use bitboards. Bitboards, in essence, use unsigned 64 bit integers to represent the sixty four squares of a chess board, and the occupancy of those squares by various pieces. One can, for instance, use a logical AND on the 64 bits representing the squares of each piece, and get an occupancy bitboard for the entire board. Bitboards can represent possible legal moves, attacks, or even specific evaluation features, such as passed pawns or knight outposts. For the purposes of time and efficiency, GERALD uses an extremely fast C++ library for board representation made by GitHub user Disservin [14].

A.1.1 Forsyth-Edwards Notation (FEN)

Forsyth-Edwards Notation (FEN) is a standard notation for describing a particular board position of a chess game. The purpose of FEN is to provide all the necessary information to restart a game from a particular position. It is also used to store databases of chess positions in a consise way. FEN is also used in the Extended Position Description (EPD) format, which is often used in the testing of chess engines [10].

A FEN record contains six fields. The fields are:

1. **Piece Placement (from White's perspective)**: Each rank is described, starting from rank 8 and ending with rank 1. Within each rank, pieces are noted from file "a" to file "h". Following symbols are used to represent the pieces:

   - **P** – White pawn
   - **N** – White knight
   - **B** – White bishop
   - **R** – White rook
   - **Q** – White queen
   - **K** – White king
   - **p** – Black pawn
   - **n** – Black knight
   - **b** – Black bishop
   - **r** – Black rook
   - **q** – Black queen
   - **k** – Black king

2. **Active Color**: Indicates whose move it is. It is "w" if White's move, "b" if Black's move.

3. **Castling Availability**: Indicates the castling ability. If neither side can castle, this is "-". Otherwise, this has one or more letters: "K" (White can castle kingside), "Q" (White can castle queenside), "k" (Black can castle kingside), and "q" (Black can castle queenside).

4. **En Passant Target Square**: If there is a pawn move that makes it possible for an opponent to capture this pawn en passant, this field shows the square. If there is no such square, this is "-".

5. **Halfmove Clock**: This is the number of halfmoves since the last capture or pawn advance. This is used to determine if a draw can be claimed under the fifty-move rule.

6. **Fullmove Number**: The number of the full move. It starts at 1, and is incremented after Black's move.

Here is an example of a FEN record, which is the FEN of the starting position of every chess game:

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

A.2 Architecture

As mentioned in Chapter 2, GERALD's architecture follows what is now considered to be a classical chess engine architecture where the search is conducted with a highly optimized alpha-beta enhanced mini-max algorithm, and the evaluation (or heuristic function) is done through a set of 'handmade' features. This architecture is now considered 'classical' because it has proved to be less competitive than deep learning or neural network approaches to the evaluation function. In fact, as is the case with AlphaZero and its predecessors, a Monte-Carlo tree search is used instead [20]. The most competitive state of the art approach, however, is to use the enhanced alpha-beta search, but to replace the handcrafted evaluation with a specialized neural network called an NNUE [20]. Nonetheless, the classical architecture is still quite popular in the field of chess engine development, and is more than capable of super human results, as demonstrated by older versions of Stockfish, the world's leading open-source chess engine [29].

A.3 Chess Engine Testing

A.3.1 Evaluation Function Testing

One of the best ways to test a handcrafted evaluation function is to test its symmetry on a large set of positions. While this does not guarantee correctness for each field, it does at least ensure there is consistency for both sides. The easiest way to ensure this symmetry is to programmatically invert a position (switching the colors of the pieces, and then flipping them across the fourth rank). Pseudo code to do this transformation given a position in FEN format is as follows:

```
Function mirrorFen(fen: String) -> String
    Initialize mirroredFen as empty String
    Initialize fenParts by splitting fen using splitFen function

    // Step 1: Flip the board and swap colors
    Initialize board with fenParts[0]
    Reverse the order of characters in board
    For each character c in board
        If c is lowercase
            Convert c to uppercase
        Else if c is uppercase
            Convert c to lowercase

    // Step 2: Swap the side to move
    If fenParts[1] is "w"
        sideToMove <- "b"
    Else
        sideToMove <- "w"

    // Step 3: Adjust castling rights
    Initialize castlingRights as empty String
    For each character c in fenParts[2]
        If c is lowercase
            Append uppercase c to castlingRights
        Else if c is uppercase
            Append lowercase c to castlingRights
```

```
    // Step 4: Adjust the en passant target square
    Initialize enPassant with fenParts[3]
    If enPassant is not "-"
        If the second character of enPassant is '3'
            Replace it with '6'
        Else if the second character of enPassant is '6'
            Replace it with '3'

    // Step 5: Reassemble the FEN string
    Combine all parts to form mirroredFen
    Return mirroredFen
End Function
```

This testing was mostly used as an initial check after a feature was implemented or modified.

A.3.2 SPRT

The sequential probability ratio test (or SPRT for short) is by far the most robust test for determining if minor changes to an engine are indeed significant enough to keep. One advantage of SPRT testing is that a sample size is not picked in advance [10], but rather games are continuosly played until it can be determined with a certain confidence level whether one version is better or wose by a pre-determined Elo margin. SPRT testing was initially formalized by Abraham Wald, and his initial paper is listed in this thesis' bibliography [33]

A.3.3 Perft

A 'perft' test functions as both a test of both correctness and speed for the move generation component of a chess engine. Chess engines will generate every legal move and reply up to a certain depth and record the nodes and nodes per second (NPS) generated. If the total nodes do not match a pre-determined known value (i.e, from the starting position, there are exactly

119060324 nodes that must be generated to reach every possible position 6 plies deep), then there must be a bug in the move generation.

Table A.3.1: Perft Table provided by Disservin's Chess Library [14]

Depth	Time(ms)	Nodes	NPS	FEN
6	539	119M	220M	START POS
6	64	11M	169M	`8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - -`
5	53	15M	293M	`r3k2r/Pppp1ppp/1b3nbN/nP6/BBP1P3/q4N2/Pp1P2PP/R2Q1RK1 w kq`
5	267	89M	335M	`rnbq1k1r/pp1Pbppp/2p5/8/2B5/8/PPP1NnPP/RNBQK2R w KQ`

A.3.4 Elo Rating

Although GERALD's rating is only an estimate, and is also estimated based on a CCRL rating, it is still somewhat comparable to the ratings of human rating organizations such as FIDE or USCF (citations needed). As such, GERALD would most likely be somewhere in the Class A Player range, marking the engine as close to a very good experienced player, but falling short of any master or expert titles.

Table A.3.2: Chess Rating Titles and Thresholds

ELO Range	Title or Class	Description
2500 and above	Grandmaster	Usually 2500 or higher
2400-2499	International Master	Usually between 2400 and 2500
2300-2399	FIDE Master	Usually between 2300 and 2400
2200-2299	FIDE Candidate Master / National Master	Usually between 2200 and 2300
2000-2199	Expert / National Candidate Master	Between 2000 and 2200
1800-1999	Class A Player	Advanced club player
1600-1799	Class B Player	Intermediate club player
1400-1599	Class C Player	Basic club player
1200-1399	Class D Player	Casual player
1000-1199	Class E Player	Novice

Appendix B
Source Code

All code is available publicly in my personal GitHub repository at `https://github.com/eharris733/Senior-Project-Chess-AI`. If for some reason this link stops working in the future, please email elliotmharris@gmail.com to request access to all code.

Bibliography

[1] Akdemir, A. Tuning of chess evaluation function by using genetic algorithms. Bachelor's Thesis, January 2017.

[2] Alridha, A., Salman, A. M., and Al-Jilawi, A. S. The applications of np-hardness optimizations problem. *Journal of Physics: Conference Series 1818*, 1 (2021), 012179.

[3] Banks, G. Ccrl - computer chess rating lists. `https://www.computerchess.org.uk/ccrl/`. Accessed: 2024-04-28.

[4] Carnegie Mellon University. The iconclast.

[5] Chakraborty, U. K., and Janikow, C. Z. An analysis of gray versus binary encoding in genetic search. *Information Sciences 156*, 3 (2003), 253–269. Evolutionary Computation.

[6] Chess.com. Chess.com members, December 2023.

[7] Chess.com. Forsyth-edwards notation (fen). `https://www.chess.com/terms/fen-chess`, 2024. Accessed: 2024-04.

[8] Christian, B. *The Alignment Problem: Machine Learning and Human Values.* W. W. Norton & Company, October 2020.

[9] Ciekce. Stormphrax. `https://github.com/Ciekce/Stormphrax`, 2024. Accessed: 2024-04.

[10] contributors, C. P. W. Chess programming wiki, 2019. Accessed: October 1, 2023.

[11] Cowen-Rivers, A. I., Lyu, W., Tutunov, R., Wang, Z., Grosnit, A., Griffiths, R. R., Maraval, A. M., Jianye, H., Wang, J., Peters, J., and Ammar, H. B. Hebo pushing the limits of sample-efficient hyperparameter optimisation, 2022.

[12] David, E., van den Herik, H. J., Koppel, M., and Netanyahu, N. S. Genetic algorithms for evolving computer chess programs. *CoRR abs/1711.08337* (2017).

[13] David-Tabibi, O., Koppel, M., and Netanyahu, N. S. Expert-driven genetic algorithms for simulating evaluation functions. *Genetic Programming and Evolvable Machines 12*, 1 (2010), 5–22.

[14] Disservin. Disservin's Chess Library. C++ library for chess-related operations, 2023.

[15] Disservin. Smallbrain. `https://github.com/Disservin/Smallbrain`, 2024. Accessed: 2024-02.

[16] Friederich, R. Rofchade. `https://rofchade.nl/`. Accessed: 2023-04-27.

[17] GitHub. Github copilot. Software tool, 2024. Accessed: 2023.

[18] Houppin, M. Stash-bot chess engine. `https://github.com/mhouppin/stash-bot`, 2024. Accessed: 2024-04-28.

[19] Katoch, S., Chauhan, S., and Kumar, V. A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications 80* (2021), 8091–8126.

[20] Klein, D. Neural networks for chess, 2022.

[21] Matthies, C. Madness in computer chess. `https://github.com/Matthies/RubiChess/wiki/Madness-in-computer-chess#destroying-the-faith-to-testsuite-results`, n.d. Accessed: April 2024.

[22] OpenAI. Chatgpt-4. OpenAI, Chatbot, 2024. Accessed: 2023.

[23] Orbital-Web. Raphael. `https://github.com/Orbital-Web/Raphael`, 2024. Accessed: 2024-02.

[24] Programming, C. Zurichess. `https://www.chessprogramming.org/Zurichess`, 2024. Accessed: 2024-01.

[25] Python-Chess Contributors. Python-Chess: A Chess Library for Python, 2023. GitHub repository.

[26] rafid dev. Rice. `https://github.com/rafid-dev/rice`, 2024. Accessed: 2024-02.

[27] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[28] Steinerberger, S. On the number of positions in chess without promotion. *Int. J. Game Theory 44*, 3 (2015), 761–767.

[29] Stockfish. Stockfish Chess Engine, 2023. GitHub repository.

[30] Suh, A., Appleby, G., Anderson, E. W., Finelli, L., Chang, R., and Cashman, D. Are metrics enough? guidelines for communicating and visualizing predictive models to subject matter experts. *IEEE Transactions on Visualization and Computer Graphics* (2023), 1–16.

[31] Thomas Ahle. Sunfish Chess Engine, 2023. GitHub repository.

[32] Vázquez-Fernández, E., Coello, C. A. C., and Troncoso, F. D. S. An evolutionary algorithm for tuning a chess evaluation function. In *2011 IEEE Congress of Evolutionary Computation (CEC)* (2011), pp. 842–848.

[33] Wald, A. Sequential tests of statistical hypotheses. *The Annals of Mathematical Statistics 16*, 2 (1945), 117–186.