Tuning of Chess Evaluation Function by Using Genetic Algorithms

Arda AKDEMIR

Submitted to the Department of Computer Engineering in partial fulfillment of the requirements for the degree of Bachelor of Science
Boğaziçi University, Fall 2016

Supervisor: Tunga Gungor

January 2017

Contents

Chapter 1

Introduction

1.1 Motivation

For centuries chess is claimed to be the ultimate test of intelligence. Chess players are always thought to possess the special brain features and believed to be the most intelligent people on earth. Goethe once said, “Chess is the touchstone of intellect.” [1] That is why at the earlier eras of AI, chess was a very popular area of research. The founding figures of AI like Claude Shannon, Alan Turing Arthur Samuel put in a great effort on creating a strong chess program.

Claude Shannon proposed two types of chess programs [2]. Type A programs calculate all possible move sequences to a certain depth and returns the best possible move by using minimax. Since the average of available moves in the game of chess is around 35 Type A programs can only search up to a certain depth. Type B programs try to guess the important moves and make a deeper analysis by avoiding the moves that it finds inaccurate. This type of programs are able to calculate deeper since they dont make an exhaustive analysis of the game tree. Type B programs only calculate the moves it thinks important and relevant to the position. Even though this type has the advantage of making a deeper analysis, overlooking a move that is actually important may cost losing a game completely. In the earlier eras of chess programs, type B computers dominated the field for a short period of time because of the lack of computing strength of earlier computers. Thus type A programs was only able to search up to a quite small amount of plies. A ply is a term used for denoting a move made by a player. A complete move in chess corresponds to a move by white and a reply by black side. Thus a ply is a half-move.

After the improvements in the hardwares of computers followed by the great increase in the computing speed of computers, type A programs began ruling the chess world. Increase in the computing strength made the usage of brute force method to exploit game trees possible. Current state of the art chess programs are well above the best chess players. Even the world’s best players acknowledge the fact that their laptops can beat them with a perfect score. The human vs computer chapter is over in the area of chess. Now chess players are considering chess programs as teachers rather than rivals.

Even though creating chess programs has been a popular area of research especially with the help of Deep Blue’s victory over Kasparov, little improvement has been made about creating cognitively plausible chess programs. Much of the effort on chess is put on improving the weights of evaluation manually or inventing new search pruning techniques. As mentioned above, state of art chess programs rely upon 1)searching a game tree by using several pruning techniques 2) a detailed evaluation function that is manually tuned by the programmers through years of experience and experiments. The level at which

current chess programs play chess would be an evidence more than enough for Goethe to claim that the chess program is intelligent but one can hardly call the current top chess programs intelligent in the way intelligence is described [3]. The chess programs are completely dependent on the knowledge given by humans and unable to create knowledge, realize patterns or learn from experience which are some of the requirements for an intelligent system.

The aim of this project is to create a cognitively plausible chess program. An application of genetic algorithms for the tuning of the weights of the evaluation function will be demonstrated. I have been playing chess since I was 7 years old at an amateur level and I have a passion for chess. The idea of creating a cognitively plausible chess engine sounded like a really interesting idea and I wanted challenge myself with this project. I believe working on this subject will help me deepen my knowledge on learning algorithms and on how to solve AI problems. Chess is a really good test-bed for artificial intelligence and I hope the ideas in this project will have applications outside the area of chess. The method I am planning to use in this project to tune the weights of the evaluation parameters can be applied to other optimization programs.

Chapter 2

State of Art

I have done an extensive Literature Survey, read more than 20 academic papers about the subject. Out of these papers I picked 6 of them and wrote summaries of the papers. These papers are all about chess programming. Considering the scarcity of academic papers on the subject I have read most of the recent academic papers on the subject. Apart from these academic papers I have done an extensive research on web about previous attempts on chess programming. I analyzed the methods used by state of the art chess programs and combined ideas from different approaches. Since chess programming requires some technical knowledge besides what is written on the academic papers, I learned about Fundamentals of chess programming by reading several sources. These sources explain what to implement on a good chess program besides how to implement them.

2.1 Literature Survey

Rahul [4] used Genetic Algorithms to tune the parameters of the evaluation function.In this paper the dependence of the current top-level chess programs to brute-force search is criticized and implementing a self-improving chess program is pointed out as a solution. Genetic Algorithms are used to evolve the parameters of the evaluation function.

Coevolution is used where the initially randomly created players(chromosomes) play against each other. 15 positional parameters are considered apart from material values to evaluate a chess board. These positional parameters are chosen with respect to their importance and their applicability on Positional Value Tables(PVTs). The evaluation function of the chess program is implemented as PVTs.

A PVT holds a value for all possible 64 squares on a chess board for each piece. Thus intuitively, a PVT for a knight will hold high values for center squares and low values for corner and side squares, because knights are usually stronger on the center squares. This idea of holding a value table for each piece enables the program to learn on its own by evolving the values stored in a PVT. A total of 10 PVTs are stored as a chromosome 6 for all piece types for middle game and 4 for pieces except rook and queen for end game. So a total of 640 genes represent a chromosome in the population the values of which will evolve over time according to the results attained from the games played against other players.

There are some novelties in the way the genetic algorithm is implemented. First, it is suggested that there are many styles of play in chess and none of them can be pointed out as the best. Thus there are many players with different styles of play with equal strength. So the program tries to prevent getting trapped in local optima by keeping several populations which are comparable in strength but different in

terms of weights of the evaluation function. So in a way the program tries to keep different style of play and prevents overvaluing of a certain style of play. This is called Multi-niche crowding. The algorithm maintains stable subpopulations within different niches and converges to different optima. An advanced free open source chess program is used and only the evaluation part of the program is replaced with the suggested PVT approach.

The result of this study is really promising since an ELO rating of 2546 is reached at the end of the coevolution of the PVT values. However, it must be noted that an already advanced chess program is used in the beginning with advanced search pruning techniques.

Sharma [5] used Artificial Neural Networks to improve the chess program.An adaptive computing technique on learning from previously played grandmaster games is suggested to hint the possible move. Articial Neural Networks are used to implement this program. The suggested program does not evaluate a given chess position but hints the piece to be played next by learning patterns from previous games.

First a set of positions is given from grandmaster games. An input is given to the ANN and the appropriate output is expected Input is the move played last and the desired output is the next move played by the grandmaster. So by looking at this input-output couples the ANN tries to capture a pattern for finding the next piece to me moved by looking at the previously played move.

Only a string of length 4 is given as an input. As an output the program hints the next piece to be played without specifing the final square for that piece. So this program is used together with a conventional evaluation function to come up with next move. The program is used to prune the search tree by enabling the search to be continued only by considering the possible moves of the hinted piece. It is observed that the program is really good at the opening stage of the game and suggests the optimal piece to be played most of the time. But in the middle stage of the game it sometimes fails to suggest the right piece and blunders.

I think this program is really innovative because it claims there are certain patterns of moves following one another in the game of chess. Conventional way of thinking in chess does not include this type of move search. The conventional way is to look at the current board position and evaluate and come up with a next move regardless of the move played before. Maybe there exists such a pattern but for now it is a mystery.

Koppel and Netanyahu [6] used Genetic Algorithms to tune both the evaluation function and search parameters of the chess engine.Most of the studies in the literature revolve around using genetic algorithms to improve the goodness of the evaluation function. This study also imroves some of the search parameters to prune the search tree more efficiently. Conventionally the parameters of the search algorithms such as null pruning and futility pruning are set manually through years of experience and thousands of trial and errors processes. This study offers using genetic algorithms to auto tune these parameters.

The evolution of the weights of the evaluation is implemented first in a two step approach. Initially the program lacks a search algorithms since the parameters of the search algorithm is yet to be tuned. Thus the program is only capable of making a 1-ply search i.e making a moving and evaluating the resulting board position without looking further ahead. First a set of positions from grandmaster games are taken from the Encyclopedia of Chess Middlegames (ECM). A population is created with the weights of the evaluation function are initialized randomly. The fitness of a chromosome is calculated by the amount of positions solved by that player which means the program is successful at suggesting the right move by looking at a certain board position. The right move in this case refers to the move made by the grandmaster. So

in this first stage of evolution the players try to imitate the grandmasters. After reaching a certain level then coevolution is used to further improve the evoluation functions. Fittest players of the first stage play against each other.

The second part of the study focuses on tuning the search parameters. This time the fitness of a chromosome is calculated by looking at the number of nodes expanded before coming up with the right move. Fittest individuals are those which expand the least amount of nodes to come up with the right move suggested by the grandmaster.

The result of this study points out a great increase at the strength of the players after evolving certain generations. Also the study shows it is feasible to assume that a chess program can come up with the right move by only making a 1-ply search almost 50% of the time which is very contradictory since the modern chess programs make up to 24-ply searches before coming up with the right move. Also tuning of search parameters is very inspring and I am also planning to tune the search parameters.

Fogel [7]used evolutionary algorithms to tune the evaluation function.This study aims to improve the weights of the evaluation function by using neural Networks and coevolution. The weights of the evaluation function are not initialized totally randomly but they are randomly distributed between a lower bound and an upper bound designated manually for each parameter. For example, the weight of the queen can take values between 800-1000 where the value of a pawn is fixed to 100. So the initial chromosomes of the program plays fairly good chess compared to previously described programs. The evolution process tunes the weights even better.The program includes 3 neural Networks for 3 parts of the board namely white area, center and black area. The weights are tuned by having the individuals play against each other. The fitness value is the score attained by playing against other players.

The result of this study shows that the evolved individual can compete against fairly good other chess engines. It is shown that the calculated ELO rating for the final version of the program is 2550 which is a Senior Master level for human players.

This study completely depends on the results of the games played between players. This type of use of evolutionary algorithms is slower compared to other types because a game takes a long time and the result of a game is either 1 or 0 which does not really give much information to tune the parameters. This type of approach either requires to much computation power or too much none of which I can afford considering the limitations I face.

Koppel and Netanyahu [1] used reverse engineering for tuning the evaluation function parameters.This study also aims to tune the weights of the evaluation function by using genetic algorithms. The novelty of the approach is that they use a top-level chess engine as a guide to be mimicked. Rather than trying to mimic the human grandmasters by looking at the next move they played this program tries to mimic the evaluation scored returned by a chess engine. By doing this the program gains much more information about the fitness because the evaluation score is a real number rather than which describes the goodness of the current position for any side. Therefore the program evolves much more faster compared to previous solutions. One drawback of this design is that this program is dependent to another chess engine which is not what we want since the aim is to create an intelligent chess playing program.

The evolution is simulated as follows: First a number of 30 parameters are set manually but their values are initialized randomly. The program also includes top-level search algorithms directly taken from other open source chess programs. So the aim is to improve the evaluation function weights. 5000 chess positions are taken from a database and the evaluation score of a top-level chess engine called

FALCON is calculated for each position. Then every individual of the population evaluates each position and returns a score. The fitness of individuals are calculated by the average differences of evaluation scores of individuals compared with the scores calculated by FALCON. So the fittest individuals are those with the lowest difference. Every individual is represented by a chromosome with 230 bits.

The evolution is run for 300 generations and the results show that the average difference decrease drastically. The results show that this type of approach is really fast for tuning the evaluation function. The paper also suggest combining two top-level chess engines and take the average of the evaluation scores for each position to surpass each of them. If the program manages to converge to the average of the two programs then the program may play better than both chess engines.

This approach is quite straight-forward and easy to implement.However relies on another manually tuned chess engine.

Kubot [8] tried implementing a learning framework from examples of certain patterns.This study aims to suggest a way to imitate human way of chess playing. Human grandmasters can recall many patterns from previous games and use this pattern recognition ability to help evaluate certain positions without making any look ahead search. For example, it only takes 1 or 2 seconds for a human grandmaster to assess which side has the slight edge without making deep calculations. This is because the players see many similar positions and already know the outcome of such positions. A similar way of learning is aimed to be implemented to computer chess programs. For this problem, this study focuses on a specific pattern that occurs frequently in chess: bishop sacrifice on h-7. This is a fairly famous sacrifice which ends with a mating pattern or a material advantage if done timely. The keyword here is “timely” because a small difference in the position makes the sacrifice a blunder. Aim of this study is to make the program learn this pattern of h-7 bishop sacrifice and be able to differentiate good sacrifices and bad ones. So at the end the program is asked to state whether the bishop sacrifice ends up good or bad in different positions.

For this purpose 200 positions are taken from grandmaster games 115 of which contain bad examples which means sacrificing the bishop will end up losing material and 85 good examples which will either end in a check mate pattern or a material advantage. The available positions with this pattern is fairly rare because the pattern is known by all grandmasters.

In the implementation each square is represented with 71 bits which makes any given chess position represented with 4544 bits. This helps to retain extra knowledge about the position such as mobility. Apart from storing which piece is on that square, each square on the board is checked whether that square can be reached by the piece. Also a 59 binary attributes are initialized manually for other parameters of evaluation of a chess positions such as king safety, mobility etc.

Error rates for this study remain above 30% most of the time which is insufficient to conclude that the program realizes the pattern well. But still the idea of teaching a program a certain pattern by showing good and bad examples is a novelty and cognitively plausible. Increasing the number of test positions can make this approach more efficient. A more frequent pattern can be used for solving this problem.

Chapter 3

Basics of Chess

In this part, I will briefly explain how does a conventional chess program work. Simply a chess program takes a board position as an input and outputs the best possible next move. So a chess program simply is a function that looks as,

$$f(B) = M$$

where B represents input the board position and M is the output move. The strength of the chess program is calculated by the accuracy of the output move. The more accurate the output move is, the better results the program will ultimately have and thus a higher rating. I prefer using the word "accurate" rather than "best move" or "right move" because most of the time in chess it is not possible to define the best move. Since chess is not a solved game and the nodes of the game tree increase exponentially it is not possible to define an objective best move. Many times during a game there are multiple moves that are equally plausible. Players with equal ELO rating but different styles of play often suggest different moves for the same position. So how does a chess program come up with accurate moves?

Conventional chess programs consist of two main parts: Search and Evaluation [9]. If we would have infinite computing power then we wouldn't need any search algorithms and we would simply exploit all the game tree and evaluation all possible chess boards. The finite computing power and the enormous complexity of chess forces to prune the game tree to be able to make a deep search. Applying minimax algorithm alone is simply infeasible because of the complexity of the game. Conventional top-level chess programs rely on several pruning algorithms to narrow down the search tree greatly without overlooking any important moves or at least overlooking important moves with a small probability. These search algorithms try to guess the most accurate move without making a full depth search, try to avoid calculating unnecessary parts of the search tree(alpha-beta pruning) and try several other heuristics such as null-move pruning, futility pruning to narrow down the search tree.

Second part of a chess program is the evaluation function. Evaluation function looks at a given board position and returns a value that predicts the goodness of the position without considering future possibilities. That is, the evaluation function evaluates a chess board statically by taking into account only the current position of the pieces. An elementary straight forward approach would be to count the material of each side and return the calculated value. Unfortunately, the goodness of a chess position depend on many parameters other than the material count and this naive approach would fail on countless situations.

The construction of an evaluation function is a two-step process. First, parameters shoud be defined. Other than piece count, a good chess program have many other carefully picked parameters such as

king safety, mobility, pawn structure etc. to measure the goodness of the position. Second, the weights of the parameters should be determined. For example, most chess books give a value of 1,3,3,5,9 to pawns,bishops,knights, rooks and queens respectively. Other parameters are less straight forward and show a greater variance among programs. This step is the most crucial part in building a chess program because it alone determines the accuracy of the evaluation function for a given position. Assuming there exists a best move or at least several equally good moves for any given position, the weights of the parameters of the evaluation function should be tuned in such a way that it returns the highest values to the positions that occur after making the best moves and lower values for positions after making other moves.

Apart from these fundamental features, state of the art chess programs include a database for openings and endgames. Opening databases are called opening books and most programs rely on the moves written inside the opening books up to 20 moves for some openings. For endgames the computers store special patterns that correspond to solved positions. So in the endgames chess programs try to achieve a position that is known to be winning without making an in-depth calculation of the position.

Chapter 4

Methods

This section will cover all the methods including both the core functionalities of a chess program such as move generation and also the genetic algorithm framework. Core functionalities are those that are always present in every chess program, they make the core structure of the chess engine.

4.1 Core Functionalities

4.1.1 Board Representation

This is the backbone of any chess program and no other feature can be implemented without having a proper, bug-free board representation. In my design I will be using bitboards which basically are 64-bit binary numbers that correspond to a chess board. Bitboards are really fast and work well with 64-bit architecture. Also the usage of bitwise operators make the otherwise complex process of generating possible moves easier for most piece types. Board representation is the way information about the chess board is represented. All other functions of the program such as search, move generation, evaluation use this representation. Besides the places of the pieces on board other information is also stored to fully describe a given chess position. Special moves such as en-passant, castling require additional information to be stored in the program.

Whole board is represented with 12 64-bitboards. Each piece type has a seperate bitboard which makes it very easy to generate moves, detect captures etc.

4.1.2 Move Generation

Move generation is another core functionality because it generates the possible moves from a given board position. The search process is completely dependent on generating possible moves at each step of the search. Move generation does not just generate all possible places each piece can move to but it also makes legality check to prevent generating illegal moves. So the move generator only generates the moves that are legal. Legality check is made by looking at the board position after making a possible move and checking if this position can be seen in a normal game. For example two kings standing at neighboring squares shows that the arrived position is illegal. This functionality can be tricky because of the nature of the game. Discovered checks and pinned pieces make the legality check a bit trickier to implement.

Move generation is made for each piece each piece type seperately. Sliding pieces,namely Queen,Rook and Bishop, require special care because they are long range pieces.Obstruction such as enemy pieces or own pieces must be detected and treated seperately.

The move generator is the basis of the search function. Search is all about generating the legal moves and evaluating the resulting positions in the leaf nodes. During the search generating the moves takes the most amount of time because it is done over and over again for each level. So making a small improvement in the move generation return great gains in total. Move generation may sound like a straightforward task but that is not the case. Detecting the all available moves for each piece on the board requires us to make several things. These include, checking if the square is available i.e not occupied by a friendly piece, if the resulting move ends us in a check etc. There are several points one must be careful in order to generate the legal moves correctly. There can be numerous approaches for generating all the possible legal moves. They both differ in complexity and in amount of time needed to execute. Implemented in my Move Generation class, there are 2 different types of move generators. The ones used in the last semester was quite simple but required too much computation. This slowed down the search time greatly, that's why we have decided to change the implementation of the move generator to gain time. This gain in time is critical during the learning phase of our project because simulations take too much time. So a 10x increase in move generation will enable us use a lot bigger population sizes or generation numbers. This in turn improves the learning gains.

Gain in time is also critical in game playing. Since the task that takes the most amount of time is search and move generation, improving the move generator means we can make deeper search during the chess game which is a really good improvement by itself. Increasing the depth of search simply makes the program stronger because it is able to calculate more combinations of moves and see results of the moves better and more accurately.

Now I will explain both designs in a detailed way and state the differences. In the first design, the first move generator does is generating the moves of the opponent without generating the moves of the side to move. This is done because this information is later used to calculate the legality of each move. So all the spots that the enemy pieces can move to are stored in an array. Then for each piece type the respective move generators are called. For each piece type, the respective move generator looks at each square on the board and checks if there is a piece of that type on that square. Then if it finds that piece on the square, it starts generating all possible moves without checking legality. The first issue about this design is checking the same squares on the board several times resulting in a waste of computation. Same square is checked 6 times. Especially during the endgame where most squares are empty this approach is fairly inefficient. After generating all possible moves for each piece, then each resulting position is checked if the side moved last is in check.

If so this means that the move made is illegal so it is not added to the legal moves list. This design is also inefficient because checking the legality of each piece is actually unnecessary. If we examine the game of chess closely, we realize that there are only 2 things to check for in legality check: If king of side to move is under attack and the pinned pieces. If the king is not under attack then only the pinned pieces may cause illegal moves. If the king is under check there are several more options that can cause illegality. So rather than checking legality for each move in the new design we categorize the positions and pieces beforehand during the generation to check only those moves that may cause illegality. For this a new variable is added for pieces that is pinned. So when generating the moves we only check once if the

piece is pinned or not.

Another difference in these two designs is that the former generates the moves of the opponent in the same way the moves of the side to move are generated. This is actually quite inefficient because in our design a lot of additional is appended to each move like the move type, if it is a capture or not etc. However, the only information we need when generating the opponents moves is the span of the enemy. We dont really need the formal moves generated for each piece on the board, rather we only want to know which squares are controlled, namely reached, by the opponent pieces. So in the new design two different move generators are used for generating the moves of diferent sides. The reach of the opponent is calculated in a faster way by excluding unnecessary information. This itself almost speeds up the process 2x because generating the reach of the enemy takes no time at all. In the old design each call for move generator is actually like 2 calls because same task is done for the enemy as well. Generating the reach is enough to calculate the pinned pieces and check if the king is under attack so we do not suffer from loss of information. After generating the reach of the enemy, pinned pieces if any exists, are detected and marked for future use. We can not deduce that pinned piece are immovable because they can move in the direction that they are pinned. So when generating the legal moves for a pinned piece we check if the move makes the piece stay on the same line it is pinned.

If the side to move is under check then there are only a handful of legal moves. So checking all possible moves seperately is a waste of time. Rather detecting the possible moves that can be legal beforehand is a better strategy. When our king is under check by only one piece then there are only three cases: Moving the king, capturing the attacker or putting a piece in between. So all moves that do not belong to these categories are thrown away without any calculation. Remaining moves stil require additional calculations for example we can not capture the attacker with a pinned piece with some exceptions that are rare. By applying these changes and updates to our move generator we received a great speed gain. In the former design we called the move generator function again for each move to test legality. This means that for depth 1 if the average number of moves is 30, then we call the move generator 32 times to do the move generation once for the side to move and 31 times for the opponent to test legality and checkmate. In the new design we call the move generator only once and do some additional work of generating the opponent reach. The net gain is around 30x, so the move generation's speed increased around 30 times. This speed increase is vast and improved the strength of the program greatly without any losses. The new design is less straight forward and a little more trickier to implement.

4.1.3 Search

One of two fundamental aspects of a chess program, Search exploits the game tree and try to find the best move sequence. This part is where much of the optimization algorithms are implemented. The most basic search algorithm is minimax where it checks all possible move sequences and return the move sequence that has the highest evaluation score assuming optimal play by the opponent. This approach is infeasible because of the number of available moves at each position. So I am planning to implement several pruning algorithms to lower the branching factor of the game tree. Below is a list of search algorithms and search features I am planning to implement.

- Alpha-Beta Pruning
- Principal Variaton Search

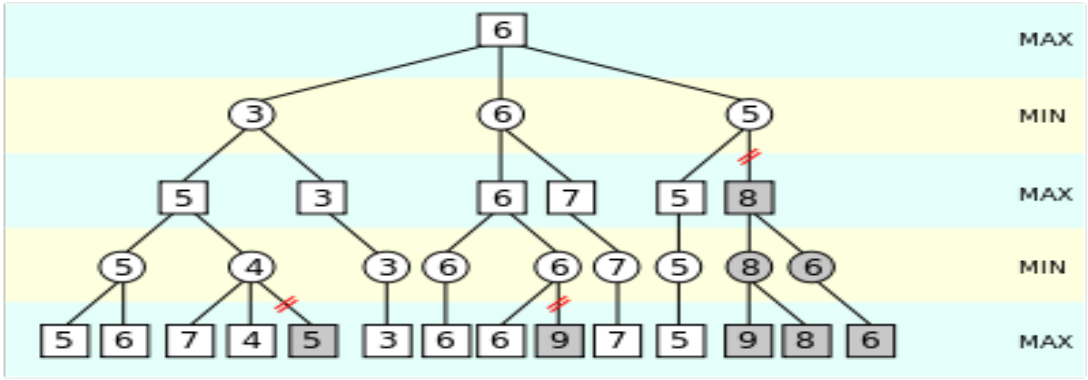

Figure 4.1: Alpha cutoff example

- Transposition Table
- Killer Move Algorithm
- Quiescence Search
- Null move Pruning
- Futility Pruning
- Iterative Deepening

This list covers some features I havent implemented since I am planning to implement them in the future. In the scope of this project, I have implemented some of them: Alpha-Beta Pruning, Iterative Deepening, Quiescence and Transposition Table.

Alpha-Beta pruning, speeds up the search process by pruning the search tree without affecting the solution find [10]. The idea behind this algorithm is pretty simple. It assumes that if a move is too good to be true then it will not be played. To be more specific, alpha corresponds to the value that white player is guaranteed to receive for the current position. It is like a lower bound. Beta is the upper bound in the same sense that, any move with a value higher than beta will not be possible because black will simply choose the other path that results with a value of Beta [11]. So a move is accepted only if it is between the boundaries alpha-beta. Depending on the side to move alpha and beta values are updated as the new moves are calculated. Pruning a branch because it has a lower value then alpha is called an alpha cutoff. Opposite is called beta cutoff.

Below is an example of alpha cutoff. We can see on the right of the figure that it is possible to prune a branch completely which gains us a lot of computing time.

However for alpha-beta to be effective the move order must be good. So the good moves should be checked first and inferior moves are checked last. Then the question is how to determine a move is good. One approach is checking captures and checks which are in general the most important moves in the game. Another heuristic, which is what I applied in my program is using iterative deepening.

Iterative deepening statically evaluates the board positions at all levels rather than just evaluating the leaf nodes. This approach may sound like a waste of time but thinking that the complexity of chess is exponential evaluating all inner nodes does not change the overall calculation work [12]. It brings however a major advantage. By evaluating the boards we have an idea about the goodness of that position

so the positions with high values are given high priorities and searched by alpha-beta algorithm first. This is a good heuristic for having a moderate move ordering without increasing the computational work.

Quiescence Search:

This search algorithm is appended at the end of the regular alpha-beta algorithm to further search the position if there are captures or checks in the end position. Search continues but this time only the moves that are not "quiet " considered as an option. So available moves are far less compared to normal search which makes it feasible. "Quiet" move is any move that is not a capture, check or a pawn push to higher ranks. These moves suggest that no immediate threats are available and evaluating the board position will not return a wrong result in most cases. The issue about this algorithm is that the definition of the quiet move is rather hard to make. A move that looks innocent may be really dangerous and can cause a major threat but in most cases they are captures and checks. Still even if the algorithm can not capture all of the threats it accurately finds them in most cases which makes it useful. In our program we used a simplified version of quiescence search which checks the available captures at a fixed depth of 2. This approach is useful but not enough to have a strong chess program.

Transposition Table:

Search algorithms that speed up the search time is crucial for game of chess. Increasing the search depth by a half move increases means a considerable increase in the playing strength so in chess programming the main focus is on making the search faster. One of the features we can add to a chess program to increase the search speed is the transposition table. This table saves the already calculated evaluation scores for positions occured in the search previously. In chess we can arrive at the same position by different move orderings. This causes to calculated the same position many times during the search. Also, we can arrive at a position at different times of the search from different positions during the game. The evaluation score may be calculated without checking the position any further but still helps us gain time. So if we store all calculated scores for each position. When we generate a new position we check the table to see if this position occured in the search previousy. If it does then we do not need to evaluate the position again.

Transposition tables are proven to be useful especially during the opening phase of the game where move ordering is not really crucial. That is why it is decided to be implemented in our design as well. However, there are several issues that should be addressed. How to store a position and what to store about a position should be addressed properly. In order to store and quickly check if the position is included in the table hash tables are decided to be used. The keys for each table should be chosen in a way to avoid collision because collisions would be really dangerous for chess. For this zobrist keys are used. Zobrist keys will be explained in detail in this section. For each square on the board a 64-bit binary number is generated randomly for each 12 piece type. Also we generate random numbers for castling and enpassant rights and side to move. Then we calculate the key for a given position as follows: Check each square on the board and if a piece exists in that square we XOR the square value of that piece with the zobrist key. This is done for castling rights, side to move and enpassant rights as well. By applying the XOR operation we can be sure that all numbers in 64 bit can occur with same probability. This is crucial because we want to avoid collisions. So we do not want approaches that generate certain keys more than the others.

Then the next issue is what to store in the hash table. There are many information for a given position and we can store any of them for future. The most important information for a given position is the best move for the position, the evaluation score associated for that move and age of the position. First to are

straightforward and age is used when cleaning positions from the hash table. Since there are too many positions in chess game and we want to avoid collisions we should be cleaning the old positions regularly before the table gets too crowded.

Simple calculations show that for a table of 10^9 entries by using 64bit zobrist keys the probability of having a collision is $1/1000$ so we can confidently have a table of size 10^9.

4.1.4 Evaluation

Evaluation is the core function of a chess engine. Without a good evaluation function a chess engine has basically no chess knowledge besides the rules of the game. Without an evaluation function we would need to search all possible moves of chess until finding a winning position which is of course infeasible considering the enormous complexity of the game.

In my design, I have chosen in total 33 evaluation function parameters. A position is evaluated as follows:

$$\sum_{n=1}^{33} Weight[n] * F[n] = Val$$

Val represents the result of the evaluation. It is the summation of all the parameters multiplied by their weights.

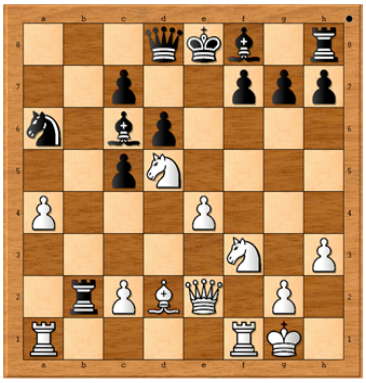

Figure 4.2: Example diagram to illustrate feature extraction.

Figure4.3 shows all the lower and upper bounds and obtained values from the simulation.Below is the full list of the 33 evaluation function parameters and their explanations:

- Weakcount is the number of squares in Player A's area that cannot be protected by Player A's pawns. For example, in Fig.4.2 the white player has 8 weak squares.
- Enemyknightonweak is the number of knights of Player B that are in the weak squares of Player A. These weak squares are called outposts and best squares for knights because they can not be pushed back by enemy pawns. For example, in Fig.4.2 the white player has 1 knight on outposts on d5 because d5 is a weak square for black.
- Centerpawncount is the number of pawns of Player A that are in squares e4,e5,d4 or d5. Center pawns are important for controlling the center and decreasing enemy mobility. For example, in Fig.4.2 the white player has 1 pawn in the center.

- Kingpawnshield is the number of pawns of Player A adjacent to Player A's king. Pawn shield is an important parameter for evaluating the king safety. For example, in Fig.4.2white king has 1 adjacent paw.

- Kingattacked is the material value of the pieces of the enemy that are acting on ones king's adjacent squares. For example, in Fig.4.2 the material value of the pieces of the black player's pieces that are acting on the white player's king's adjacent squares is 0.

- Kingdefended is the material value of the pieces of the Player A that are acting on Player A's king's adjacent squares. For example, in Fig.4.2 the material value of the pieces of the white player's pieces that are acting on the white player's king's adjacent squares is the sum of the values of Queen, 2 Rooks and Knight.

- Kingcastled returns 1 if Player A is castled to measure king safety. For example, in Fig. 4.2white king is castled.

- Bishopmob is the number of squares that a bishop can go to. This type of parameters are calculated seperately for each bishop. For example, in Fig. 4.2the number of squares that the white bishop on d2 can go is 5.

- Bishoponlarge parameter returns 1 if the bishop on the given square is one of the two large diagonals of the board. Bishops are stronger on the large diagonals because they have higher mobility and they are reaching the two central squares simultaneously controlling the center. For example, in Fig. 4.2the black bishop on c6 is on a large diagonal.

- Bishoppair returns 1 if Player A has two or more bishops. Bishop pairs are generally considered an advantage as to bishops can together cover all possible squares regardless of the color of the square. Bishop pairs are especially strong in open positions where there are no central pawns and the bishops can move freely to create threats. For example, in Fig. 4.2black player has the bishop pair.

- Knightmob is the number of squares that a specific knight can go to. For example, in Fig. 4.2the number of squares that the white knight on f3 can go to is 5.

- Knightsupport returns 1 if a given knight on a given square is supported by ones own pawn. Supported knights are strong because they can only be backfired by pawns and since they are supported they can stay in an important position for a long number of moves making it harder for the opponent to play around it. For example, in Fig. 4.2the knight on f3 is supported by a white pawn.

- Knightperiphery0 returns 1 if a given knight is on the squares a1 to a8,a8 to h8,a1 to h1 or h1 to h8. This is the outest periphery and most of the times knights on these squares are weaker. For example, in Fig. 4.2the black knight on a6 is on periphery0.

- Knightperiphery1 returns 1 if a given knight is on the squares b2 to b7,b7 to g7,b2 to g2 or g2 to g7. For example, in Fig. 4.2no knights are on periphery1.

- Knightperiphery2 returns 1 if a given knight is on the squares c3 to c6,c6 to f6,c3 to f3 or f3 to f6. For example, in Fig. 4.2the white knight on f3 is on periphery2.
- Knightperiphery3 returns 1 if a given knight is on the squares e4, e5,d4 or d5. For example, in Fig. 4.2the white knight on d5 is on periphery3.
- Isopawn returns 1 if a given pawn has no neighboring pawns of the same color. Isolated pawns are generally considered as a weakness since they cannot be protected by pawns so they should be protected by other more valuable pieces. For example, in Fig. 4.2the pawn on e4 is isolated.
- Doublepawn returns 1 if a pawn is doubled pawn. Doubled pawns are considered a disadvantage as they blocked each other and they are vulnerable to attacks. For example, in Fig. 4.2the black pawns on the c-column are doubled.
- Passpawn returns 1 if for a given pawn there are no opposing pawns of the enemy on the neighboring columns and on the given pawn's column ahead of the pawn. If a pawn is passed it is big threat for the opponent because the are no pawns on the way to prevent it from promoting. For example, in Fig. 4.2white pawn on a4 is a passed pawn.
- Rookbhdpasspawn returns 1 if a rook of the same color is behind the passed pawn. If there is a rook behind a passed pawn it is easier to push to pawn forward as it is always protected by the rook and rook never gets in the way. For example, in Fig. 4.2the white pawn on a4 is passed and defended by the rook on a1.
- Backwardpawn returns 1 if the neighboring pawns of a pawn are ahead of it. Backward pawns are the last pawn of a pawn chain and even though they are not isolated they can not be defended easily. So they are considered a disadvantage. For example, in Fig. 4.2the white pawn on g2 is a backward pawn.
- Rankpassedpawn is the rank of the passed pawn. A passed pawn on rank 7 which means the pawn is one move away from promoting is a lot more dangerous compared to a passed pawn on its initial square. Passed pawns with higher ranks have higher priority thus they are an advantage. For example, in Fig. 4.2the a4 pawns rank is 4.
- Blockedpawn returns 1 if a central pawn on column e or d on its initial square is blocked by its own piece which severely decreases the mobility of the pieces. In Fig. 4.2there are no blocked pawns.
- Blockedpassedpawn returns 1 if a passed pawn of Player A is blocked by a piece of Player B which prevents it from moving closer to promotion. This is an advantage for the blocking side. The passed pawn on a4 is not blocked by any black pieces. If it is pushed to a5 however it will be considered blocked because there is a black knight on a6.
- Rookopenfile returns 1 if a given rook is on a file with no pawns from either side. Rooks are stronger on open columns because they can move freely. For example, in Fig. 4.2the black rook on b2 in on an open file.
- Rooksemiopenfile returns 1 if a given rook is on a file with no pawns from its own side. Rooks are strong on semi-open files as well. For example, in Fig. 4.2the white rook on f1 is on a semi open file.

- Rookclosedfile returns 1if a given rook is on a file with pawns from both sides. Rooks on closed files are considered a disadvantage as they have lower file mobility and no access to the important squares of the game especially in the middlegame and endgame.

- Rookonseventh returns 1 if a given rook is on seventh rank from the Players perspective. For white that would be the rank 7 for black rank 2. Rooks on seventh rank are dangerous and a classical theme in chess for creating major threats at once. For example, in Fig. 4.2the black rook on b2 in on the seventh rank.

- Rookmob returns the number of squares a rook can move to. For example, in Fig. 4.2the black rook on b2 has a mobility value of 9.

- Rookcon returns 1 if there are no pieces between to rooks of the same color and they are on the same file or on the same rank. Connected rooks defend each other to create threats in the opposition area because they cannot be captured by queen or king. For example, in Fig. 4.2the whites rooks are connected.

- Queenmob returns the number of squares a queen can move to.

- Pawn value is the number of pawns of the Player A. These parameters are the simple material count parameters.

- Knight value is the number of knights of the Player A.

- Bishop value is the number of bishops of the Player A.

- Rook value is the number of rooks of the Player A.

- Queen value is the number of queens of the Player A.

These evaluation parameters try to catch the subtleties of chess besides basic factors.

4.2 Genetic Algorithms

Parameter Name	Lower Bound	Upper Bound	Obtained Value
Pawn Value	100	100	100
Knight Value	200	400	342
Bishop Value	220	420	374
Rook Value	400	600	530
Queen Value	800	1000	911
Center Pawns	-30	70	-8
King Pawn Shield	0	100	35
King Attacked	-100	0	-
King Castled	0	100	60
Bishop on Large	0	100	74
Bishop pair	0	100	5
Bishop Mob	0	15	13
Knight Mob	0	15	14
Knight Support	0	100	40
Knight Perip0	-100	0	-51
Knight Perip1	-60	40	-18
Knight Perip2	-40	60	45
Knight Perip3	-30	70	-1
Isp pawn	-50	50	-3
Double pawn	-50	0	-7
Pass pawn	-20	100	62
Rook behind pawn	0	100	30
Backward pawn	-70	30	-14
Rank-pass pawn	-40	40	5
Blocked pawn	-50	0	-23
Blocked pass pawn	-50	0	-10
Rook on openfile	0	100	27
Rook on semi open	0	100	57
Rook on closed	-60	30	-46
Rook on seventh	0	100	41
Rook mob	0	10	9
Rook connected	0	100	12
Knight on Weak Squ	0	100	-39
Queen Mobility	0	10	3

Figure 4.3: Table for boundaries and obtained values

In this part, I will explain how I will use genetic algorithms to tune the weights of the evaluation function. The methodology used here is quite similar with the work done by Koppel and Netanyahu [6]. To keep the process simple and fast I will not include any search algorithms when using evolutionary algorithms. So the program will only generate the possible board positions by making a 1-ply search, i.e it just checks the possible board position 1 step ahead.

In my design a chromosome represents the weights of the evaluation function parameters. The parameters of the evaluation function is largely taken from the same work by Koppel [6]. Apart from those parameters, several other parameters will be included and several parameters from the work will be excluded. The evaluation function parameters are explained above. I will have 33 parameters form y evaluation function. So a chromosome can be thought as an array of length 33 where each element corresponds to a parameter. An individual with this chromosome will try to guess the

next move played in positions taken from grandmaster-level chess games by making a 1-ply search and using its own parameter weights. If the individual succesfully predicts the next move of the game then 1 point will given. This will be done for 1000 different board positions and the total number of moves predicted correctly will be calculated divided by 1000. This will be the fitness value for an individual with a chromosome.

$$f(N) = (numberofmovespredictedcorrectly)/1000$$

Same procedure will be made for 10 individuals and fittest chromosomes will be calculated. Fittest individuals will preserve their chromosome by transmitting them to the next generation through crossover and stucking in a local point will be avoided through implementation of mutation. The parameters used by the program is as follows:

- Population size=10-20
- Crossover rate= 0.25
- Mutation rate = 0.007
- Number of generations = 200

This process tries to mimic the behavior of the grandmasters assuming that every move made by a grandmaster is optimal or at least very close to the optimal move. This assumption is pretty accurate considering the fact that super grandmasters have CAPS(Computer Aggregated Precision Score) ratings over 97%. CAPS is a recent new way of measuring the strength of a chess player by calculating the accuracy of moves with the help of state of the art computer programs. A typical grandmaster with a rating of 2400 have a CAPS score of 96.20% on average. This means that for 96.20 percent of the board positions which corresponds to 9620 positions out of 10000 games I use in my project, there are no better moves known as yet other than the one played in the game. This CAPS rating system of course assumes that for a given chess position there are several moves that are equally plausible. So for most of the time it is safe to assume that the move made in the game in the database is the correct move. At each generation the individuals who successfully found the correct moves the most will go to the next generation. At the end of 200 generations I am planning to see an increase in the average number of moves solved by best players of that generation compared to the initial individual. In the beginning, the genes in the chromosomes of all 100 individuals will be initialized randomly from an interval initialized manually. This is to avoid weird initialization of weights such as giving a higher value for a pawn than a queen etc. Thus we make here the assumption that our chess program starts with a very basic chess knowledge. This approach will help to fasten the process of tuning the weights as there wont be a huge offset from the optimum values.

Position after 11.Bg5

For example, given the position on the left as an input, the program should evaluate all the possible board positions resulting from all legal moves for black. Then the move with the highest evaluation score should be given as an output. If the output matches with Bobby Fischer's 11... Na4!! then that individual receives 1 point. Actually This move may have some bonus points considering that the game is considered as The Game of The Century. This game was between 13-year-old Bobby Fischer and Donald Byrne and

the black's 11th move is accepted as "one of the most powerful moves of all time." For those who are interested Byrne didn't accept the sacrifice and answered with Qa3 as taking the knight on a4 would put him in great trouble.

But this approach alone lacks the true notion of intelligence as it totally depends on grandmaster games. So without guidance this approach is unable to generate new ideas or improve its strength which are some basic requirements for an intelligent system. Thus after implementing the genetic algorithms on solving grandmaster positions, I am planning to further improve the weights of the evaluation function by applying coevolution method. Coevolution means the strongest chromosomes created during the first part of the process will play against each other in a tournament style. The strongest individuals will be the ones with highest points and they will transmit their genes to the next generation. This process of coevolution will continue for several generations. The aim of the second part is to have even stronger individuals than the individuals created with mimicking the grandmasters alone. Second part is also important in the sense that it is not dependent on any other intelligent system as it improves its parameters only by playing against itself. The details of this part of the project will be determined later according to the progress of the project.

We followed the same approach with Netanyahu and Koppel [6] about the genetic algorithm framework. In that approach, the best move suggested by the program is compared with the move played by the grandmaster. If they match then we increase the fitness of the individual suggested the move. So fitness of an individual is calculated as the number of positions that the best move suggested by the program matches the grandmaster. This approach is quite straightforward and is a really good way to measure fitness. However, it has several drawbacks. First, in chess there can be multiple moves that are equally plausible at a certain position. Since the game of chess is too complex to calculate all possibilities until the end we can not be sure exactly sure about the goodness of moves. Of course in most cases we dont need to calculate each position until the end of the game to compare two moves suggested at a position. In most of the cases one of the moves are a lot better than others like capturing a piece, attacking the king etc. However, in some cases especially in the middlegame there are a lot of cases where different grandmasters go for different approaches and select different moves.

The games in our databases are games played by super-grandmasters, they all are from the top 50 grandmasters. However, even for them there are several equally plausible options. This feature of the game of chess makes us reconsider the definition of fitness of an individual. Even if the move suggested by the individual does not match with the move played by the grandmaster this does not necessarily mean that this move is a bad move. The databases we have consist only of the moves played by the grandmasters and we dont have all the moves that grandmaster think that are almost as good as the ones played by them. So learning the other plausible moves is impossible with this database. So what we change in our approach is rather than having the program return the best move calculated, the program returns all the moves that it thinks that are almost as good. Then it checks if any of the moves matches with the one played by the grandmaster. This way, surely we can calculate the fitness of an individual more accurately. However, we need to be careful about several things. What exactly means equally plausible or as good? In chess programs the score of a move is calculated based on evaluation scores. The move with the highest evaluation score is returned as the best move. So in our approach we use mathematical approximity to decide if there are any other moves equally plausible. The second best move is not always equally plausible because it may have a really lower evaluation score compared to the first move suggested by the

program. Conversely, for example the 5th best move is not necessarily a bad move if it has a really close evaluation score. This latter case is rather rare but occurs nevertheless. For these reasons, we suggest a new approach. Rather than comparing the best move, we define "almost equally plausible" moves that are inside a lower bound calculated by considering the evaluation score of the best move. For example, say we have the evaluation score 100 for the best move. Then we mark all the moves with a value above 90 as equally plausible. We do not consider how many moves if any are included in this range. Then we compare all these returned moves with the move played by the grandmaster. If any of them matches, then we increase the fitness. Another issue, is the amount we increase the fitness because we need to give a higher fitness if the first move matches than if the second move is the one that matches. This is handled in a straightforward way. We decrease the amount by 0.2 as we check the next best move suggested. This number can be updated but results show that it works fine.

Comparing multiple moves helps us accurately calculate the fitness of individuals. So we are able to better sort the individuals which is very crucial for genetic algorithms. The most important part of a genetic algorithm is to define the fitness of individuals precisely. By applying this subtle change we show that we gain an additional 5 percent increase in the strength of the individuals. This is a meaningful increase considering the overall fitness went up from 21% to 26% as further explained in the results section.

4.2.1 Depth Increase

Without the proper implementation of some search algorithms we could only make the genetic algorithm simulation using a search depth of 1 because of the time it takes to simulate a depth higher than 1 took too much time. However, after updating the move generator, the search algorithms and the way we run the simulation the program got a lot faster enabling a search of depth 2. This means the program searches all possible moves at a depth of two and decides on the best move. This coupled with the capture routine which is a simplified version of the quiescence search makes more robust decisions free from simple blunders. This helped us gain an additional improvement of 3%. So we move from 25% to 28%. So the best move suggested by the program matches the grandmasters move in almost 1 out of 3 cases. This ratio converges to 1 out 2 if we consider the top three moves suggested by the program. These results will be further discussed in results section.

4.3 Critical Decisions

Because of the nature of this project we have many variables to decide about. In the beginning, I will introduce the independent variables of the system and how I deal with them.

First decision is about the database we use for the simulation. There are many databases online and free each including millions of games played at all different levels. I decided to choose games of top-level grandmasters. I have downloaded games played by 10+ super-grandmasters and mixed them.

Another decision I had to make was about the move, I want my individuals to solve of each game in the database. For this, I follow the same decision made in several other studies. Middlegame positions is the most appropriate positions for properly using the evaluation function parameters. Most of the parameters are useful when there are many pieces on the board and the position is dynamic. In the opening

stage of the game players generally use their opening knowlege to play without making deep analysis. The opening theory behind the grandmaster moves is hard to capture with static evaluation parameters.

Also the endgame positions most of the time have seperate endgame books because when there are less pieces on the board players can not create immediate threats but rather exploit subtleties such as a simple pawn advantage to win the game. Thus rather than statically evaluating the board by using all the parameters, players focus on certain targets. For example, in the endgame stage king safety is not a meaningful parameter because king becomes an attacking piece when there are no major pieces on the board. So I decided to give middlegame positions to the individuals. Rather than trying to come up with an exact move number for each game, I give an interval from which a number is picked randomly to decide which move to guess from each game in the database. My simple design is guessing the nth move where n= 20+ 20*rand() where rand() returns a value from 0 to 1 . I both succeed in asking positions from middlegame and also succeed in not giving an exact value which may result in overlearning. If the move to be predicted is always the 10th move then individuals will be compared according their ability to solve 10th move of chess rather than the whole middle game positions.

Also I made an experiment for opening stage of the game to be able to compare the results obtained from different stages of the game and draw conclusions about the results.

Another set of variables I had to decide about is the parameters of the genetic algorithm, namely Population size, Crossover rate, mutation rate, generation size. For this I combined several values from different studies that fit my engine and purpose the most. In my design it takes around 1 second for an individual to solve 1000 positions when doing a 1-ply search in the machine I am using. So having a population size around 1000 would be infeasible because simulation would take too long for big number of generations. Population size, generation size and number of positions to be solved determine how long it takes to make a simulation. These parameters are decided for having a feasible simulation time. I have made different simulations with different values. The table below shows the interval for these values I have used during my simulations.

Parameter Name	Low Boundary	Upper Boundary
Database	1000 positions	3000 positions
Population	10 individuals	50 individuals
Generation	100 generations	200 generations

Figure 4.4: Table of genetic algorithm parameters

Another decision was about the way to initialize weights of the parameters of the evaluation function of the first generation. There are different approaches used in several studies on the subject. Some studies use lower and upper boundaries for each parameter and initialize each parameter from that interval. This approach has the drawback in the sense that initial individuals are dependent to external human knowledge because the lower and upper boundaries should be set manually. Second approach is to randomly initialize the weights. This approach assumes no prior chess knowledge and starts the genetic algorithm from zero. However because of the nature of the problem at hand, tuning all 30+ parameters meaningfully from zero is infeasible. Several studies show that no meaningful progress has been made about tuning the weights with initializing the weights randomly because there are just too many parameters to be tuned simultaneously. Thus I have implemented the former approach and initialized the individuals by using lower and upper bounds. The table of the boundaries is shown in Fig4.2.

Chapter 5

Results

I first present the results of running the genetic algorithm without using the capture routine mentioned in the methods section. Then, I provide the results of running the experiment with using the capture routine. In the end, I show the results of running the same simulation for the opening stage of the game.

Capture Routine is a novelty of our work to overcome the severe limitiation caused by making 1-ply search. In 1-ply search it is impossible to see the result of the actions taken by the player because the search is limited to 1-ply. So no further analysis is made. Even though 1-ply search is selected because of complexity issues, this problem causes the individuals to have a quite low success rates.

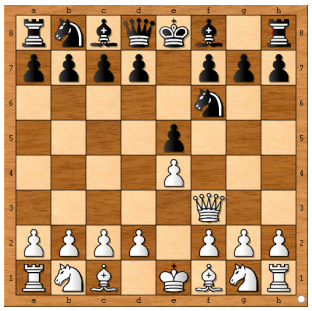

Figure 5.1: Typical 1-ply blunder

In Fig 5.1, a typical 1-ply search blunder is shown. Queen is attacking a defended knight on f6. Since the search is limited to 1-ply white player assumes capturing the knight ends up profiting the white side because white will be up by a knight.

Capture Routine overcomes such blunders by checking all possible captures by opponent after reaching the end of the search. So the program checks available captures by black after Qxf6 and finds out that queen will be captured on the next move. So it will extract the queen value from the already calculated value of the position and return it as a result. Thus the individuals will not be affected by very simple blunders.

One other blunder Capture Routine overcomes is being unaware of the fact that one side is being attacked.

Figure 5.2 shows an example of this type of blunder. White queen is attacked by a black pawn but white is attacking blacks bishop. Without Capture Routine, the individual will capture the bishop losing the queen. This type of blunder is prevented because Capture Routine gives a lower value to the position when white captures the bishop. Capture Routine sees that in the next move queen is attacked and white will lose the queen.

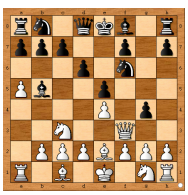

Figure 5.2: Attacked

5.1 Genetic Algorithm Without Using Capture Routine

Genetic Algorithms are simulated and improvement over 5% is observed.Our results show that genetic algorithms significantly improved the initial version of the individuals.

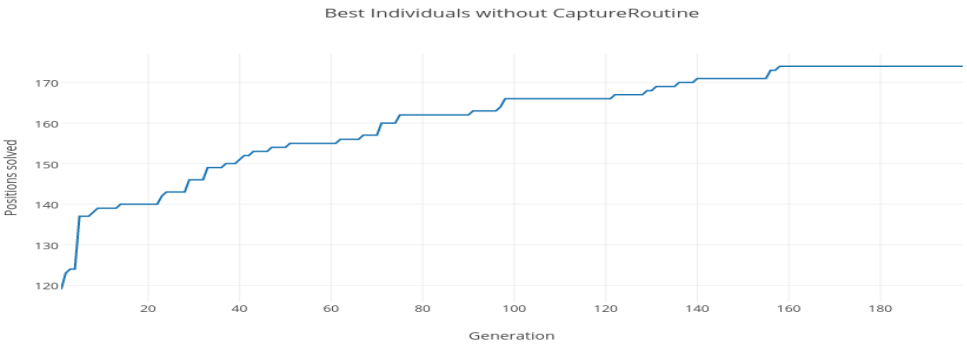

Figure 5.3: Genetic Alg. without Capture Routine

5.2 Genetic Algorithm with Capture Routine

Genetic Algorithms coupled with the capture routine shows significant improvement over other methods. Capture Routine enabled the individuals to start with a higher initail scores for the first generations, because they did not blundered simple positions similar to the ones explained above. Also, genetic algorithms helped the best individual improve over time. The simulation is stopped around 150th generation because no improvements have been recorded in the last 40 generations.

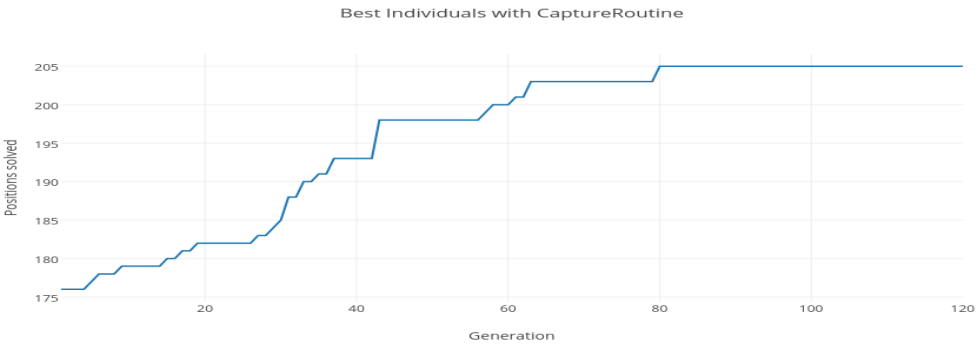

Figure 5.4: Genetic Alg. with Capture Routine

5.3 Genetic Algorithm with Multiple Move Comparison

In this experiment, we compared not only the best move returned by an individual but top 5 moves with the grandmasters move. This helps us assess the fitness of an individual more accurately. Previous studies compare only the best move returned by the chess program. This idea has some flaws. First one is because

of the nature of the chess game. Assuming that a grandmasters move is the best move possible is a good approximation. Yet ignoring the fact that in a position there can be multiple equally plausible moves is a mistake. By only comparing the best move returned, previous studies make the false assumption. Another flaw is that by only making one comparison and adding 1 point if the move matches with grandmaster's the algorithm gives no credit to an individual whose second best move always matches with the grandmasters move. So this way one can not accurately assess the fitness of an individual. Say we have two individual, where the first one has 10 first move matches and 0 other matches and second individual has 9 first move matches and 20 second move matches. Now without much discussion one can claim that the second individual is stronger. Yet previous studies are unable to catch this. The results can be seen below for using MMC.

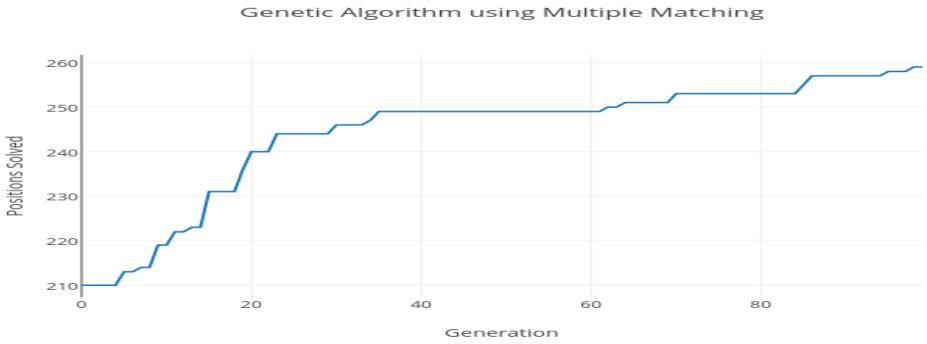

Figure 5.5: Genetic Alg. with Multiple Move Comparison

5.4 Genetic Algorithm with Multiple Move Comparison and 2-ply Search

In the next experiment, we tested the improvement obtained by using a 2-ply depth in search. This is made possible by adding several search algorithms that help prune the search tree better.

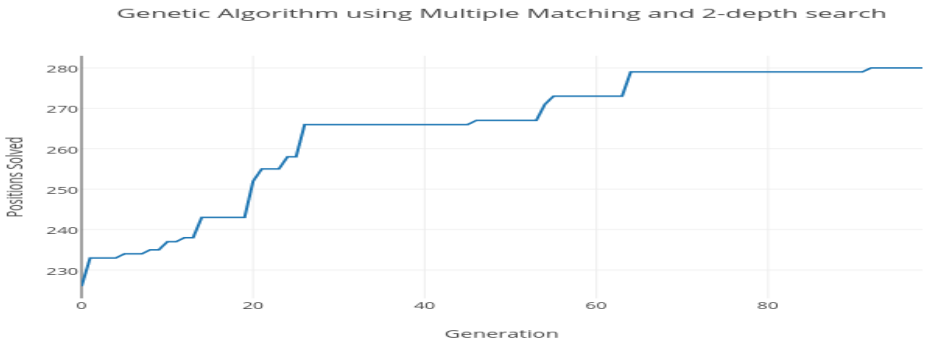

Figure 5.6: Genetic Alg. with Multiple Move Comparison at 2-ply

5.5 Comparing the Results

I have conducted 100+ experiments but only mentioning here the most important ones. In order to verify that the results I have obtained are meaningful I have made two further experiments. First, I implemented a random guesser which only generates all legal moves at any given chess position and picks one randomly. I used this routine and conducted a simulation on the same 1000 positions I used as training. Ten individuals are again used for this experiment and the result was 28 out of 1000 positions on average. So random guessing succeeds almost 3% of the data.

Second, I conducted an experiment with 1 generation and 1000 individuals meaning that all my individuals are initialized randomly by using the boundaries for each parameter. This experiment is to test whether spending 200 generations on genetic algorithms makes a significant difference or not. If the results are similar, we can just randomly generate individuals without using any algorithm and save the weights of the best individual.

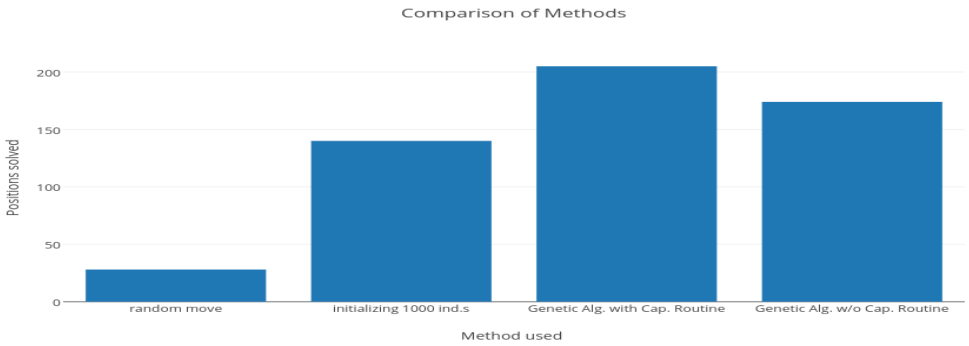

Figure 5.7: Comparing the Methods

Figure 5.5 shows that randomly initializing individuals can not generate an individual above about 14% success rate. So genetic algortihms really help us to finely tune the weights rather than random guessing.

Another comparison is made between methods that use Multiple Move Comparison and methods with different search depths.

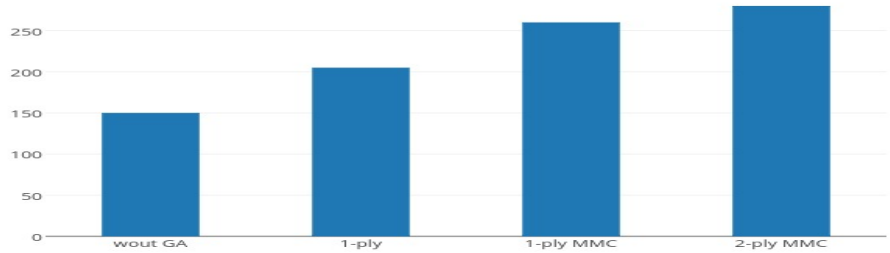

Figure 5.8: Comparing the Methods

Figure 5.8 shows that as we increase the search depth and use MMC the strength of the best individual obtained in the end of the simulation increases.

5.6 Testing against itself

In order to objectively claim that the strength of the program increased the program played 30 chess games against the initial version of itself, i.e before applying the genetic algorithms. Out of 30 games 13 resulted in a win, 5 resulted in a loss and 12 resulted in a draw. Below is a pie chart for visualizing these results.

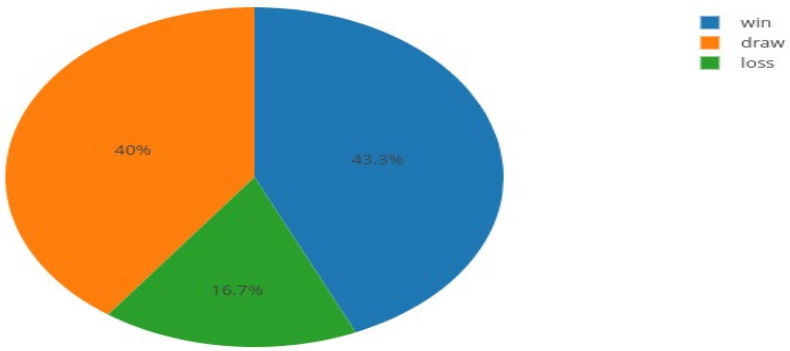

Figure 5.9: Results of the matches against the initial version

5.7 Opening Stage Simulation

Last simulation I will mention here is about the opening stage of the game. I have already explained above why we decided on working with middlegame stage. This simulation is to confirm our assumptions about the opening stage of the game. The genetic algorithm produced individual that reached 30% success rate in mere 40 generations. This corresponds to roughly 10% difference with our best results from the middlegame positions.

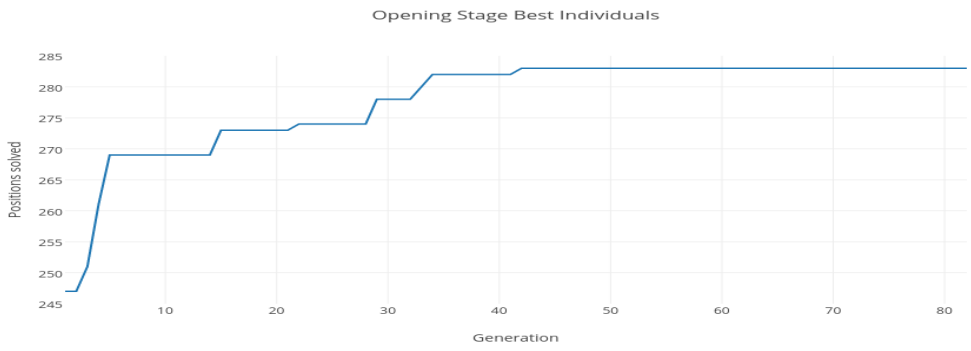

Figure 5.10: Opening Stage Success

Figure 5.6 shows the resullts for the simulation of the opening stage. It should be noted that best randomly initialized individual have a success rate of around 25% which is a lot higher than other randomly initialized individuals.

There are two main reasons for this:

- Less available moves in the opening stage.
- Similar patterns are repeated by grandmasters in opening stage.

Chapter 6

Conclusion

In this project, it is verified that genetic algorithms work successfully to tune the evaluation function parameters. 1-ply search without a capture routine makes simple blunders because of the search limitations. However, 1-ply approach is used because the aim of this work is to assume as little chess knowledge as possible at this stage of the game and 1-ply search is really fast and suitable for simulations. 1-ply search can not capture the subtleties of grandmasters so the success rates are not high. Rather, 1-ply search captures the simple patterns of chess. To further exploit the advantages of the evaluation function parameters deeper search should be implemented.

6.1 Future Work

Up to now we have implemented several search algorithms to a simple chess program and tuned its parameters by using Genetic Algorithms. In the future I am planning to implement several other search algorithms and further strengthen the chess program. Brief explanation for each algorithm is given below.

Null-move Pruning:

This algorithm is a further improvement on alpha-beta. It assumes making a null move that is just changing the side to move and evaluates the position based on that. If the value returned is still higher than a certain value we can be sure that this position is good because we assume that there is always a move better than doing nothing. This algorithms only flaw is the zugzwang positions where the side who moves loses the game. So additional control of such positions should be included to have a bug-free null move pruning.

Killer Move Search:

Killer moves are moves that result in a cutoff during the alpha-beta pruning. This means that they are good moves. Since during a game of chess some moves are remain to be dangerous even if the position changes slightly this search approach considers trying the killer moves of previosly searched positions that look similar first. Since move ordering is very important for alpha beta pruning, searching the killer moves first may result in faster cutoffs.

Mate Search:

As the name suggests, this search algorithm works in a different way compared to normal search. Mate search only tries to find a sequence that results in a checkmate. There generally is an upper bound for the search depth in order to return results in a short amount of time. In this search several heuristics are used which prunes many available moves that does not look like help checkmate. For example, a pawn

move that is far from the opponent's king will probably not going to help. Distance between the square a piece moves and the opponent king is a good information for this kind of search. Also, direct attacks on the king are the first candidates to be considered.

Appendix A

Appendix

As a standard I am using ARENA chess user interface. I connect to ARENA through UCI protocol.

Appendix B

Web References not cited

- https://chessprogramming.wikispaces.com/
- https://en.wikipedia.org/wiki/The_Game_of_the_Century_(chess)
- https://www.chess.com/article/view/better-than-ratings-chess-com-s-new-caps-system
- http://wbec-ridderkerk.nl/html/UCIProtocol.html
- https://www.youtube.com/user/jonathanwarkentin
- http://s.hswstatic.com/gif/chess1.gif

Bibliography

[1] Moshe Koppel. Expert-driven genetic algorithms for simulating evaluation functions. *Genetic Programming and Evolvable Machines*, pages 5–22, 2010.

[2] Vazquez Fernandez. An evolutionary algorithm coupled with the hooke-jeeves algorithm for tuning a chess evaluation function. *IEEE Congress on Evolutionary Computation*, 2012.

[3] Mandziuk Jacek. Some thoughts on using computational intelligence methods in classical mind board games. *IEEE International Joint Conference on Neural Networks*, 2008.

[4] Rahul A R. Phoenix: A self-optimizing chess engine. *International Conference on Computational Intelligence and Communication Networks*, 2015.

[5] Sharma Diwas. An improved chess machine based on artificial neural networks. 2014.

[6] Moshe Koppel. Genetic algorithms for evolving computer chess program. *IEEE Transactions on Evolutionary Computation*, pages 779–789, 2014.

[7] David Fogel. A self-learning evolutionary chess program. *Proceedings of the IEEE*, 2004.

[8] Kubot Miroslav. Learning middle-game patterns in chess: A case study. *Intelligent Problem Solving. Methodologies and Approaches Lecture Notes in Computer Science*, pages 426–433, 2000.

[9] Vazquez Fernandez. An adaptive evolutionary algorithm based on tactical and positional chess problems to adjust the weights of a chess engine. *IEEE Congress on Evolutionary Computation*, 2013.

[10] Boskovic b. A differential evolution for the tuning of a chess evaluation function. *2006 IEEE International Conference on Evolutionary Computation*, 2006.

[11] Cook Andrew. Using chunking to optimise an alpha-beta search. *International Conference on Technologies and Applications of Artificial Intelligence*, 2010.

[12] Penceci Ihsan. Comparison of search algorithms and heuristic methods in the solution of chess endgame problems. *24th Signal Processing and Communication Application Conference*, 2016.