Phoenix : A Self Learning Chess Engine

A Thesis Report

Submitted in Partial Fulfilment for the Award of

M.Tech in Information Technology

By

Rahul. A. R (MT2012109)

To

International Institute of Information Technology, Bangalore

Bangalore – 560100

June 2014

Certificate

This is to certify that the thesis report titled **Phoenix : A Self Learning Chess Engine** submitted by 'Rahul A R' (MT2012109) is a bonafide work carried out under my supervision from December 26th, 2013 to June 15th, 2014 in partial fulfillment of the M.Tech. Course of International Institute of Information Technology, Bangalore.

His performance and conduct during the thesis was satisfactory.

Prof. G. Srinivasaraghavan

Date: The 15th of June, 2014

Place: IIIT Bangalore

Abstract

Since the advent of Computers, many tasks which required humans to spend a lot of time and energy have been trivialized by the computers' ability to perform repetitive tasks extremely quickly. However there are still many areas in which humans excel in comparison with the machines. One such area is chess. Even with great advances in the speed and computational power of modern machines, Grandmasters often beat the best chess programs in the world with relative ease. This may be due to the fact that a game of chess cannot be won by pure calculation. There is more to the goodness of a chess position than some numerical value which apparently can be seen only by the brain. Here an effort has been made to improve current chess engines by letting themselves evolve over a period of time.

In this thesis, initially a brief history of chess programming is reviewed which includes well known concepts like alpha-beta pruning and quiescence search techniques. A basic introduction to Genetic Algorithms is given and an evaluation routine is constructed. A genetic algorithm is then used to optimize the routine. A working model is implemented, tested and the results are presented.

Acknowledgements

I take this opportunity to express my profound gratitude and deep regards to my guide Prof. G. Srinivasaraghavan for his exemplary guidance, monitoring and constant encouragement throughout the course of this thesis. I am also obliged to all my teachers without whom I would not have had the knowledge and confidence to complete this endeavor. Last but not the least, I thank the Almighty, my parents, grandparents and friends for their constant support.

In the days when Sussman was a novice, Minsky once came to him as he sat hacking at the PDP-6. 'What are you doing?', asked Minsky. 'I am training a randomly wired neural net to play Tic-Tac-Toe' Sussman replied. 'Why is the net wired randomly?', asked Minsky. 'I do not want it to have any preconceptions of how to play', Sussman said. Minsky then shut his eyes. 'Why do you close your eyes?', Sussman asked his teacher. 'So that the room will be empty.' At that moment, Sussman was enlightened.

— Rahul A R

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Invention

The precursors of chess originated in India during the Gupta Empire. There, its early form in the 6th century was known as Chaturanga, which translates as ‘four divisions (of the military)’: infantry, cavalry, elephantry, and chariotry. These forms are represented by the pieces that would evolve into the modern pawn, knight, bishop, and rook, respectively. According to chess historians Gerhard Josten and Isaak Linder, ‘the early beginnings’ of chess can be traced back to the Kushan Empire in Ancient Afghanistan, circa 50 BCE - 200 CE [11].

Though we do not know the exact moment in history when chess was invented, there are many stories which are intriguing. One story goes on to say that chess was invented by the demon king Ravana during the time of the Ramayana. It is said that one day his wife Mandodari complained that she was bored. To amuse his queen, he invented chess and taught her the rules of the game. It is also said that being a brilliant mind, she promptly beat him at it [18].

Another story talks about an ancient Indian Brahmin named Sissa as the inventor of chess. According to the Persian poet Firdowsi, when Sissa showed his invention to the ruler of the country, the ruler was so pleased that he asked Sissa to name his prize for the invention. The man cleverly asks for wheat, but with a condition. He stipulated that for the first square of the chess board, he would receive one grain of wheat, two for the second one, four on the third one, and so forth, doubling the amount each time. The king blindly agrees only to realize his mistake later and give his entire kingdom to Sissa as his gift [16].

1.2 Chess and AI

Many eminent researchers like John McCarthy, Allen Newell, Claude Shannon, Herb Simon, Ken Thompson and Alan Turing have put significant effort into computer chess research. This is because many factors make chess an excellent domain for AI research. Some of them are listed below: [39]

- Richness of the problem-solving domain
- Ability to monitor and record progress accurately through competition and rating, because of its well-defined structure
- Chess has been around for centuries - the basics are well-understood internationally, expertise is readily available and is (generally!) beyond proprietary or nationalistic interests. It has been considered a game of intelligence
- Detailed psychological studies of chess playing exist. These studies suggest

that human players use different reasoning modes from those in current chess programs. Further, these reasoning modes are also used in many other problem-solving domains

- It is an excellent test bed for uncertainty management schemes which is the basis of most expert problem-solving. The well-defined nature and discreteness of the game have led many to ignore this

1.3 Purpose of Research

The most important factor that influences the strength of a chess engine is the way in which it evaluates moves. Humans have mastered this evaluation process and use strategies which are abstract and complex. These techniques cannot be simulated using a computer because of its limitation to understand abstract concepts. Computers can only operate on numbers. Therefore we must represent a chess position as a group of numbers. Every positional parameter which might influence its goodness must have a numerical equivalent in this group. The construction of such a group/vector of number is non-trivial. There are 2 main reasons for this:

- A small change in the position can result in a large change in its goodness. For instance, 2 positions which are identical except for the position of a queen, which is offset by a single square, might differ largely in their goodness

- The value of a position might vary from player to player as it depends on the goal he sets for himself. That is, a position might be losing for a defensive player and might be drawable for an attacking player

Most chess engines today use brute force methods to simulate this thinking process by taking into consideration multiple parameters which influence the value of a position. But these parameters are decided by the programmers while coding and not by the engine itself. In this thesis, we make the engine learn these parameters by itself using genetic algorithms.

1.4 Chess Notation

Chess notation is the term used for several systems that have been developed to record either the moves made in a game of chess or the position of pieces on a chessboard. The earliest systems of notation used lengthy narratives to describe each move; these gradually evolved into terser notation systems. Currently algebraic chess notation is the accepted standard and is widely used. Algebraic notation has several variations. Descriptive chess notation was used in English and Spanish literature until the late 20th century, but is now obsolete. There are some special systems for international correspondence chess [6].

We are particularly interested in notations used by computers. The standard notation used in all chess engines today is the Portable Game Notation (PGN) which is a computer friendly version of the Standard Algebraic Notation (SAN). A brief description of SAN follows.

1.4.1 Standard Algebraic Notation (SAN)

SAN (Standard Algebraic Notation) is a representation standard for chess moves using the ASCII Latin alphabet. Examples of SAN recorded games are found throughout most modern chess publications. SAN as presented in this thesis uses English language single character abbreviations for chess pieces, although this can easily be changed to other languages.

1.4.1.1 Square Identification

SAN identifies each of the sixty four squares on the chessboard with a unique two character name. The first character of a square identifier is the file of the square; a file is a column of eight squares designated by a single lower case letter from ‘a’ (leftmost or queenside) up to and including ‘h’ (rightmost or kingside). The second character of a square identifier is the rank of the square; a rank is a row of eight squares designated by a single digit from ‘1’ (bottom side [White’s first rank]) up to and including ‘8’ (top side [Black’s first rank]). The initial squares of some pieces are: white queen rook at *a1*, white king at *e1*, black queen knight pawn at *b7*, and black king rook at *h8*.

1.4.1.2 Piece Identification

SAN identifies each piece by a single upper case letter. The standard English values are: pawn = ‘P’, knight = ‘N’, bishop = ‘B’, rook = ‘R’, queen = ‘Q’, and king = ‘K’. There is no special identification for the pawn.

1.4.1.3 Move Construction

A basic SAN move is given by listing the moving piece letter (omitted for pawns) followed by the destination square. Capture moves are denoted by the lower case letter ‘x’ immediately prior to the destination square; pawn captures include the file letter of the originating square of the capturing pawn immediately prior to the ‘x’ character. SAN kingside castling is indicated by the sequence ‘O-O’; queenside castling is indicated by the sequence ‘O-O-O’.

1.4.1.4 Check and Checkmate

If the move is a checking move, the plus sign ‘+’ is appended as a suffix to the basic SAN move notation; if the move is a checkmating move, the octothorpe sign ‘#’ is appended instead [5].

Chapter 2

Computer Chess

2.1 History of Chess Programming

The time period between 1949 and 1950 is considered to be the birth of computer chess. In 1949, Claude Shannon, an American mathematician, wrote an article titled Programming a Computer for Playing Chess [46]. The article contained basic principles of programming a computer for playing chess. It described two possible search strategies for a move, which circumvented the need to consider all the variations from a particular position. These strategies will be described later when we talk about implementing chess as a computer program. Since then, no other strategy has been developed which works better and all engines use one of these strategies at their cores.

About a year later, in 1950, an English mathematician Alan Turing [47] (published in 1953) came up with an algorithm aimed at teaching a machine to play chess. Unfortunately, at that time there was no machine powerful enough to implement such an algorithm. Therefore, Turing worked out the algorithm manually and played against one of his colleagues. The algorithm lost, but it was the beginning of computer chess.

In the same year, John von Neumann created a calculating machine which was very powerful for that time. The machine was built in order to perform calculations for the Manhattan Project. But before it was used there, it was tested by implementing an algorithm for playing a simplified variant chess (6x6 board without bishops, no castling, no two-square move of a pawn, and some other restrictions). The machine played three games: it beat itself with white, lost to a strong player, and beat a young girl who had been taught how to play chess a week before [30].

In 1958, a great leap in the area was made by scientists at Carnegie-Mellon University in Pittsburgh. Their algorithm, called alpha-beta algorithm or alpha-beta pruning, the modern version of which is considered in detail later in this section, allowed the pruning of a considerable number of moves without having any penalties in further evaluation. With this, the number of position evaluations per unit time increased by a factor of 5.

Another interesting idea to improve computer's expertise was proposed by Ken Thompson. He reorganized the structure of an ordinary computer and built a special machine named Belle [23], whose only purpose was to play chess. This machine appeared to be much stronger than any existing computer and held the leading position among all chess playing computers for a long period in the 1980s, until the advent of 'Hi-tech', a chess computer developed by Hans Berliner from Carnegie-Mellon University, and the 'Cray X-MPs' [30].

Since then the progress in computer chess is mainly the result of the ever increasing computing power. By the end of 1980s, an independent group of students made their own chess computer called Deep Thought that happened to be the prototype of the immortal Deep Blue which won against the then world chess champion Garry Kasparov in 1997 [36].

2.2 Chess Engine Internals

2.2.1 The Board

A chess program needs an internal board representation to maintain chess positions for its search, evaluation and game-play. Beside modeling the chessboard with its piece-placement, some additional information is required to fully specify a chess position, such as side to move, castling rights, possible *en passant* target square and the number of reversible moves to keep track on the fifty-move rule [4].

There are 2 main ways of representing a chess board: Mailbox and Bit boards. A brief description of each is given below.

2.2.1.1 Mailbox

The mailbox representation was one of the original ideas sketched out by Shannon in [46]. During his time the computing power and memory of the largest computers were nothing compared to what we have today. Therefore programmers always aimed at optimizing a program, sometimes at the cost of increasing its memory requirements.

According to this representation, the chessboard itself consists of 64 integers each from -6 to 6 (negative numbers represent the black pieces and vice versa. Empty squares are represented by 0). Another integer is used to indicate the side to move. This is not an optimal representation of a position but as mentioned previously, simplifies calculations. A move is described by specifying three parameters: index of the source square, index of the destination square and another to take care of pawn promotions as and when they happen.

The program then assigns the value of the source square to the destination square (or the value of the third parameter in the case of promotion) and then 0 to the source square. This is a convenient and efficient way of describing a move, and a similar (if not the same) idea is used in the implementations of most board games involving 2 players. The main drawback of such a representation is finding the edges of the board so as to prevent the pieces from moving outside the board.

2.2.1.2 Bit Boards

Another approach called Bit boards was invented independently by two groups of scientists: one from the Institute of Theoretical and Experimental Physics in Moscow, USSR, and one from Carnegie-Mellon University, Pittsburgh, USA, led by Hans Berliner. They represented each square of the chess board by a single bit, thus using just one 64-bit computer word to represent any state of the board. The entire chess board position could now be represented in 12 such words (one for each piece). Each Bit board is filled with zeros except for the bit which represents the square where the particular piece is present.

Two more Bit boards that contain all white pieces and all black pieces present on the board, respectively, are usually used. The bit boards containing the squares, to which a certain piece on a certain square is allowed to move, can be easily constructed using Boolean operations. Other information like King safety squares can also be stored using Bit boards if the developer wishes to. But, *en passant* and castling possibilities must be kept in separate variables.

As mentioned above, it is very easy to derive additional information from these boards. For instance, it greatly simplifies finding the legal moves for a piece: all the program has to do is to perform logical AND operation on the Bit board representing all possible moves of the piece with the negation of all other Bit boards which represent pieces of the same color. For more such examples and a comparison between Mailbox and Bit board representations see [29].

2.2.2 Move Search

Like most 2 player board games, the game of chess can be represented as a huge tree with the starting position as the root, all subsequently possible positions as its nodes and all terminal positions as its leaves. Therefore it is possible to scan the tree and find a path leading to victory from almost any given position.

For instance, in a standard game of Tic-Tac-Toe (using the 3×3 board), the overall number of final positions is less than 9! and it takes less than a second for a modern computer to find the best path. In chess however, for each move played, there are generally about 30-35 different reply moves which can be played. Thus, assuming that an average game finishes in 60 moves i.e. 120 plies[1], we get $30^{120} \approx 1.8 \times 10^{177}$ leaves. This is more than the assumed number of atoms in the Universe squared!

Of course, it is to be noted that only a few moves among the 30-35 are really playable in any position if the player wishes to win. For example, the move '*Qxe4 d5xQ*' makes sense only when the sacrificing side is going to mate soon or will take the opponent's queen *en prise* later. Otherwise the game is most certainly lost.

[1]2 plies make 1 full move

The number of valuable moves varies for different positions, but on average there are not more than 4-5 such moves. Even this does not solve the problem, since the number $5^{120} \approx 7.5 \times 10^{83}$ is still humongous. But this question, i.e. selecting only few moves for consideration and ignoring the rest, is described already by Shannon in [46]. It is named 'type-B' strategy, whereas the other technique where no move is omitted is called 'type-A' strategy.

We know that humans always use type-B strategy, but the thinking process behind this is not yet quantifiable in terms of numbers and operations between them. Therefore we cannot program computers to use type-B strategy directly. History has only proved this fact: most computer chess programs using this strategy eventually overlooked the losing move, which seems unlikely to happen, according to the algorithms they used. Therefore the problem of creating a move generator that never fails is far from being solved [29].

2.2.3 Minimax Search

From the previous section we can conclude that it is reasonable for computers to evaluate positions up to a certain depth and then move, instead of traversing till the leaves. One technique used in this sort of evaluation is the *minimax*. After each ply, the algorithm rescans the game tree to the same depth, but starting at a level lower (in other words, going one level deeper), and thus obtaining acceptable results each time.

The word *minimax* expresses the idea of minimal loss and maximal profit. That is, the algorithm tries to minimize the maximum loss or maximize the minimum profit of a player. It means that 2 best values, the minimum and the maximum are considered. The best move is the one that leads to a position with the best evaluation score for the side to move.

In order to get a clear understanding, let's consider an example of such a process of choosing minima and maxima. Assume that we have an initial position with white to move and the depth to which the search is performed is 4 plies (2 full moves). Also, assume that the evaluation function returns white's score and it is symmetrical i.e. white's score is equal to black's score, but with the opposite sign. This situation is shown in Fig. 2.1. Here grey squares are the nodes from which the minimum value is chosen (at each level) and white squares are the nodes from which the maximum value is chosen.

The last move is made by black. It means that end positions must be considered from black's point of view and hence, the best move is the one that provides the lowest score or simply the minimum. At a higher level, all the numbers which were lifted up must again be compared with other nodes of that level and the one with the maximum value must be selected since it is white's turn. This procedure of alternation is repeated until the root of the tree is reached, where the maximum value is picked and the appropriate move is made.

2.2.4 Alpha-Beta Pruning

Let us again consider Fig. 2.1 and the depth-first method of observing the tree, which is usually used in modern computer chess programs [40]. Assume that we have already searched the tree till leaf 8 which has an evaluation score of -15. We can see

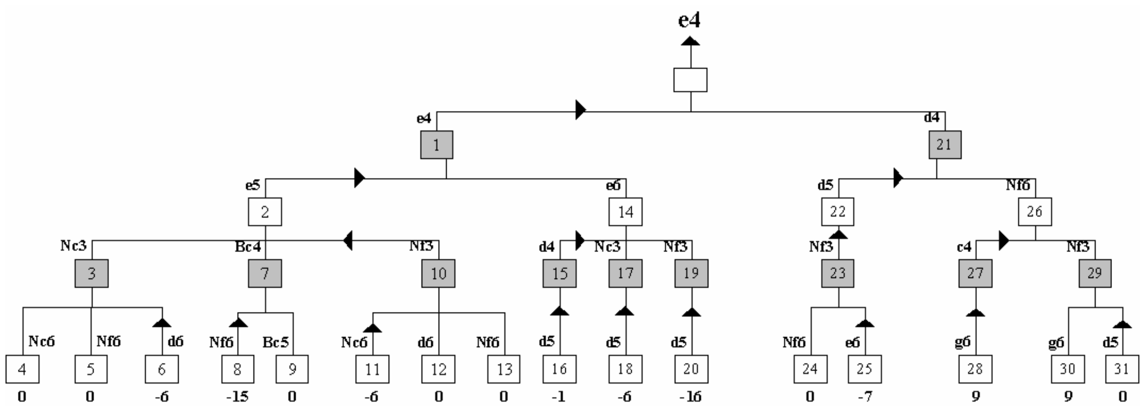


Figure 2.1: Full Game Tree

```
int Minimax(position p) {
        int m,i,t,d;
        Descendants(p,d); //Define all descendant positions p1,...,pd
        if d = 0
                return EvaluatePosition(p);
        else {
                m = - infinity;
                for i = 1 to d {
                        t = -Minimax(pi);
                        if t > m
                                m = t;
                }
        }
        return m;
}
```

Figure 2.2: The Minimax Algorithm

that the score at node 3 is -6 i.e. if white chooses the path to node 3, the worst score it may get is -6. But if it plays to node 7, it gets -15. This can be determined by just considering the first reply. It means that there is no reason to consider all other replies to the move at node 7 which have not been considered at the moment, since white will in any case play the move leading to node 3. In other words, we can simply skip considering leaf 9.

Using this line of reasoning, we can skip other leaves such as 12 and 13. The same procedure can be applied at the level above by reversing the logic i.e. taking the maximum instead of minimum. For instance, there is no reason to consider nodes 17 and 19 (and their children), since black will always choose the path leading to node 2. And finally, the entire sub-tree rooted at node 26 can be skipped, since white will always play to node 1, which gets a final score of -6, rather than to node 21, which already has a score of -7 with incomplete search.

The method described above is called α-β algorithm or α-β pruning and according to [40], was first presented in[43]. Later in [38], the topic was reviewed and supplemented with a proof of correctness and time complexity evaluation.

```
int AB_Search(position p, int a, int b) {
        int m,i,t,d;
        Descendants(p,d); //Define all descendant positions p1,...,pd
        if d = 0
                return EvaluatePosition(p);
        else {
                m = a;
                for i = 1 to d {
                        t = -AB_Search(pi, -b, -m);
                        if t > m
                                m = t;
                        if m >= b
                                break;
                }
        }
        return m;
}
```

Figure 2.3: Alpha-Beta Pruning Algorithm

At every stage, when there is no need to examine the rest of the sub-tree, it is cut-off. Values α and β are white's and black's best scores found so far. The main advantage of this algorithm is that we get the same result as we would if the entire tree were to be examined. The sketch of this algorithm [38] is given in Fig. 2.3.

2.2.5 Transposition Tables

There are many ways to reach the same position in chess. For instance:

- *1. e4 e5 2. Nf3 Nc6*
- *1. Nf3 Nc6 2. e4 e5*
- *1. e4 Nc6 2. Nf3 e5*
- *1. Nf3 e5 2. e4 Nc6*

Hence it seems natural to prevent a program from considering the same position multiple times. To make it clear, consider two positions A[2] and B[3] and a 4-ply search depth. Assume that position B is now being searched and the move *3. Ng1 Nb8* has been just examined and there are still two plies left. Now, this position is exactly the same A which has already been evaluated. Therefore spending time on evaluating this position again will be equivalent of searching to a depth of 2 plies and not 4. Hence positions which are already seen are stored in *Transposition Tables*.

To implement such a table we need to uniquely identify every position the evaluation routine has seen. One way to do this (the most famous and widely used) was suggested by Zobrist in [49]. Zobrist suggested assigning a 32 or 64-bit random number to each piece located on each square; i.e. $12 \times 64 = 768$ numbers altogether. An empty square is assigned 0. A set of numbers is generated for different castling possibilities and for *en passant* capture status.

Then starting with a null hash key, a XOR operation is performed between the current hash key and the random number assigned for the piece on the square in

[2]A: *1. e4 e5*

[3]B: *1. e4 e5 2. Nf3 Nc6*

question. The procedure is repeated for every square of the board. This value is then XORed with random numbers for castling and *en passant* possibilities. Finally, if it is black's turn, the result is again XORed with another random number. This number is a hash key for the current position.

Of course the total number of positions in chess is much larger than the largest number which can be held in 64 bits. The probability of two positions to have the same hash key is small but greater than zero. Therefore it is possible to repeat the whole procedure described above with different random numbers, thus obtaining a second hash key for the same position. The probability of two different positions to have identical hash keys is small enough to provide uniqueness of positions within a single game.

There is no definite rule as to how a transposition table should be filled. This is because the size of hash keys as well as the size of the table depends on the resources available. Also, the number of unique hash keys is far greater than what can be stored in such tables. Therefore the hash keys must somehow be mapped onto the table indices. One method is to simply obtain the index as a remainder of the current hash key when divided by the size of the transposition table.

Hence in every game there will be a number of positions that will point to the same entry. To handle this, there should be a measure of the age of a position so that the chess engine knows if a certain position is old enough to be replaced with the new one. Naturally, the more entries a transposition table has, higher the probability of a position to be found in it.

2.2.6 Move Ordering and Killer Move Heuristic

Coming back to α-β pruning, it is very important to have moves ordered in such a way that there are as many cut-offs during the search process as possible. Evidently, the most cut-offs happen when the best moves are searched first. This problem is considered to be among the most important ones in the area of α-β search in computer chess.

Generally it is impossible to know in advance which move proves to be the best in every scenario, as otherwise there would be no need to search at all. Therefore all we can do is, use prior results and combine them with information about the current situation on the board in order to create a sequence of moves which is likely to be in the best order.

To start with, all capture moves are worth considering first. For simplicity, we do not talk about special cases such as when the reply makes a check with a fork[4] to opponent's queen or starts a mating attack. These situations are much rarer and are handled in some special way. Pawn promotion can also be considered as a capture move as it changes the material balance on the board.

Next, all checks should be considered and then rest of the moves. This approach however, uses only information at hand and is obtained for every position independently of the game history. Another refinement consists of storing details of the search performed so far. For instance, it does not matter if a pawn moves one or two squares if the reply is a queen capture (*1. h5 2. Qxa5 or 1. h6 2. Qxa5* in Fig. 2.4). Therefore the sub-tree rooted at the queen capture can be cut-off and this move can

[4]a piece attacking two other pieces simultaneously

be added to the top of the move list to be evaluated.

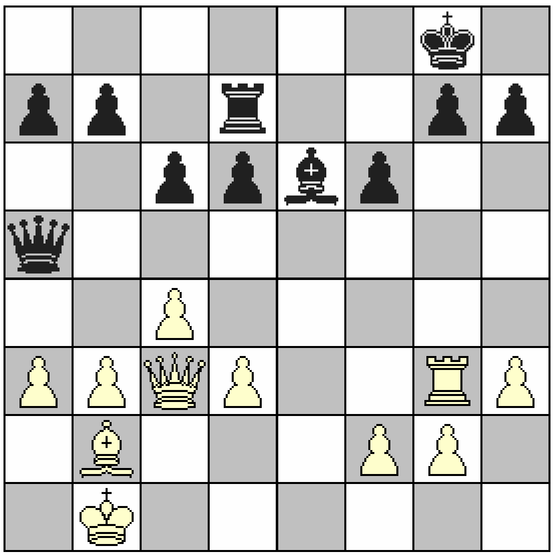


Figure 2.4: Valuable and Useless Moves

This idea is referred to as the *Killer Heuristic* and the moves that caused quick cut-offs are named *killer moves.* All this and some additional detailed information on the techniques used for move ordering can be found in [33] and [41].

2.2.7 History Table

As discussed previously, we can place some moves for the current position onto the top of the search list based on the scores they achieved a move ago. The History Table approach suggests a similar technique wherein we store information about all recently examined moves and not just the killer moves [33]. The advantage here is that, it is possible to accumulate information about the effectiveness of each move in the entire game tree unlike in Killer Heuristic where only a certain sub-tree is considered.

Each time a move proved to be good (caused a quick cut-off or achieved a high evaluation score), its characteristic which indicates how good this move is, is increased and the greater this characteristic, higher is the move's privilege in the list. For example, the move that was placed among the best ones 2 plies ago will still have a good characteristic and can be placed at the top even if a different piece can move now. Thus in Fig. 2.4, after the game continued *1. Qxc3 2. Bxc3*, white's move *Bxf6* (instead of *Qxf6* a move ago) is still dangerous. Of course all this makes sense only for a certain period of time and hence the history table must be cleaned periodically.

2.2.8 Iterative Deepening

Usually in chess, it is time that restricts the quality of moves in both humans and machines. As discussed already, a skilled human who uses type-B strategy and focuses on several mostly acceptable moves will generally fare well, whereas a machine cannot do so always. Therefore it is important for a machine to choose the best move within the given constraints (usually in a few seconds). Due to this sort of restrictions, machines usually cannot evaluate every move as desired by its algorithm.

Iterative Deepening tries to solve this problem. In this method, the program searches for moves in a breadth-first manner rather than the usual depth-first technique. In case the timer expires, it returns the best move belonging to the deepest level which has been searched completely. As stated in [33], the advantage of this method is that the number of nodes to be visited in successive iterations put together is generally much smaller than that of a single non-iterative search that goes depth-first. Another factor which influences the goodness of this technique is the fact that move ordering becomes easier when the search is made level by level which in turn improves the effectiveness of the α-β pruning algorithm.

2.2.9 Quiescence Search

Let us now consider one more shortcoming of regular search through the following example. Assume that the search depth is 5 plies and at the 5th ply there is a move which takes the opponent's pawn with the rook. As it is the last ply and future moves cannot be considered, the evaluation function is called and the score is returned. Now the apparent result is that the moving side wins a pawn and therefore the move leading to this position is estimated as a favorable one.

But on the 6th ply, there may be a move with which the opponent simply takes back and goes a whole rook up. This kind of behavior, when a program is not able to see enough into the future, is called the Horizon Effect [21]. Of course, after the corresponding game path is chosen and a move is made, the game goes one level down and now the program can reach one level deeper and see the recapture. But what if the chosen move leads to the loss anyway? For example, instead of the rook capture there may be a bishop capture in the best case scenario. Situations such as these are unpredictable and dangerous.

Since every position during game play is not ready for the evaluation, only relatively *quiescent* positions [46] where the least possible action takes place should be evaluated (such positions are also called *dead* positions in [48]). This is why all capture moves and pawn promotions are usually considered separately and searched till a depth where the results of all material changes finally appear. In addition, moves are ordered in a special way depending on the taking piece and the piece to be taken in *most valuable victim/least valuable aggressor* manner.

More details can be found in [33]. Special considerations should also be given to check moves because they always allow only a few forced replies and the actual situation is not apparent. There may also be an explosion in the number of nodes to be considered when there is a series of checks given continuously. This situation can be resolved by limiting the number of extra plies given to inspect check moves. In [29] a value of 2 is said to be the mostly used one.

2.2.10 Null Move

Null move [19] [27] [31] as the name suggests, means skipping a turn and allowing the opponent to play 2 moves in a row. The idea here is to see if the opponent can change the situation of the game adversely by playing twice. If the result of applying a null move is acceptable for the skipping side, there is no need to continue with the full search because it most likely leads to a cut-off [33].

The significance of this technique is that it takes away a whole ply from the current search tree and hence the program needs to traverse the search tree to a depth of N-1 instead of the original N. In the middle of the game, when the number of legal moves is about 30-35, applying null moves take only about 3% of the time. In case of success i.e. a null move results in chipping off one ply, the program saves 97% of the time required to make a move by pure searching. If the application of null moves fails, only 3% of the total time is wasted.

However, there are special situations known as Zugzwang[5] in which a null move

[5]German for compulsion to move or forced to move

is the only way to avoid loss. But applying null moves to such positions will lead to mistakes. In position A of Fig. 2.5, black does not have a move which will maintain the current material balance. That is, any good move (such as *Ke8, Kf8, Kf7, Kf6* or *Rb7*) allows white to win the pawn on d6. Other possible moves (such as *Bc7, Rc7* or *Ra7*) lose even more material. The situation in position B is not so evident at first glance and requires deeper analysis. It is left as an exercise to the reader.

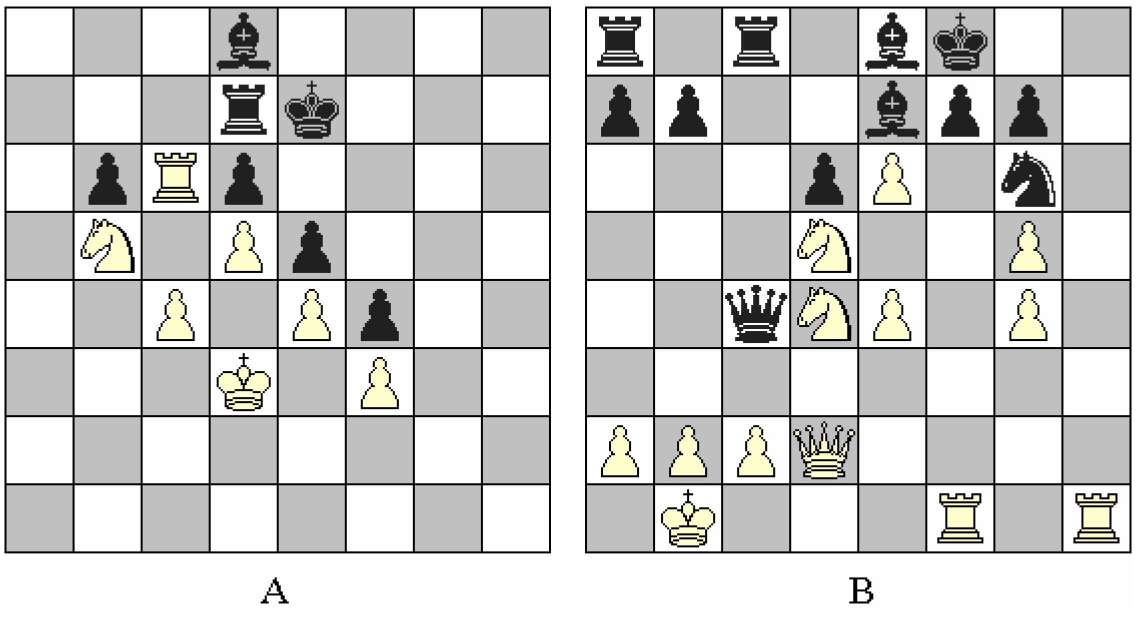


Figure 2.5: Zugzwang Positions

Zugzwang positions are almost always losing. Therefore the loss of performance generally does not affect the final result. On the other hand according to [33], Zugzwang happens extremely rarely in chess with the notable exception of late endgames. Therefore application of null moves is usually stopped when the number of pieces left on the board is less than some predefined value. So the null move refinement is worth being implemented, especially if the aim is to increase search speed.

2.2.11 Opening and End Games

Since chess play can lead to so many different positions, it is often useful to memorize a few moves which can be played directly if a certain pattern occurs on the board. Therefore people started documenting games played by higher ranking players and analyze them so that popular or frequently occurring positions can be studied and the best moves for them can be memorized for future use.

Nowadays due to the advent of computer analysis, chess theory has become a huge body of knowledge. There are hundreds of books, some of which are entirely devoted to a single opening. These books discuss in detail the main lines that appear when starting a game with the opening in question and bring out ideas and ways to develop pieces which are proved to be the best.

Since computers were built to store and retrieve data efficiently, storing opening lines in a database gives computers the ability to make the best moves without any evaluation whatsoever. The same idea can be implemented for endgames which are also analyzed deeply and in many cases solved. Every position in an endgame database is assigned a value of $+\infty$ (victory), $-\infty$ (loss), or 0 (draw); the final result of the game assuming perfect play from both sides.

During move search if there is a positional match with a database entry, that position becomes a leaf of the search tree and receives the evaluation score from the database directly. According to [33], there are three different kinds of endgame databases available (though many more are available today):

- Thompson's collection of 5-piece databases
- Edwards' tablebases (gives only depth to mate)

- Nalimov's tablebases (up to 6 pieces)

Nalimov and its derivatives have gained more popularity among recent chess programs due to their considerable advantages in indexing and size. Thompson's databases which were the first of its kind had a number of disadvantages such as slow search in the deeper levels of the game tree. Edwards' tablebase tried a different approach based on the depth to mate which became a success but with the disadvantage of being huge in size. Nalimov's tablebase is actually an improvement of Edwards' original with advanced indexing schemes. More information can be found in [33] where there is an entire chapter devoted to endgame databases used by the famous chess program Dark Thought.

2.3 Evaluation Routine

As said earlier, it is neither possible nor advisable to evaluate the whole tree of future moves at every stage of a chess game. Therefore we need some measure to quantify chess positions so that we can recognize good moves without having to traverse till the leaves.

In general, an evaluation function is a multivariate, linear function which measures the goodness of a chess position. There are various features which can be extracted from a chess position that will give us some insight into the goodness of the position. The inputs to the evaluation function are numbers which quantify these features. The output is a single number called the *Evaluation Score*.

$$F = \sum_{i=0}^{N} x_i \cdot v_i, \qquad \begin{cases} x_i \in \{0, 1\} \\ v_i \in \mathbf{R} \end{cases}, i = \overline{1, N}$$

Here x_i indicates the presence of the i^{th} parameter and v_i represents its importance as a real number. The position is good for white if this score is positive and vice versa. Also in some programs, a positive score is considered to be good for the current player and a negative score for the opponent.

A chess position is simply a legal permutation of pieces placed on different squares on the board. Each piece has a varying degree of importance and therefore a value is assigned to each of them. The sum of values of all pieces of a particular color is called the *material count* for that color. The difference between the material counts of white and black is known as *material difference* and is considered to be the most important parameter in deciding the goodness of a position.

The evaluation score is generally taken to be $+\infty$ if the opponent's king is checkmated and $-\infty$ if the player's king is checkmated. Trivial situations like 'king against king', 'king against king and knight', 'king against king and bishop' along with more complex positions which have already been analyzed and known to be drawn positions are put into a database with a score of 0. This avoids redundant calculations which ultimately lead to wastage of resources.

However, in general it is not possible to claim equality of two positions taking into account only the material balance. In several opening lines, one side is ready to sacrifice a pawn on purpose; for e.g. king's gambit accepted (*1. e4 e5 2. f4 exf4*) and queen's gambit accepted (*1. d4 d5 2. c4 dxc4*). Here the program that uses an opening database must still somehow consider itself to be in an advantageous position, even after having one pawn less.

Sometimes a player can sacrifice quality (like exchanging an inactive rook for an active bishop or knight) for achieving some non-material advantage that is considered worthwhile. Moreover, highly skilled chess players often agree to call the game a draw even when there is material imbalance. So in all these situations, other factors apart from the material balance have to be taken into consideration during position evaluation. These factors are known as *strategic* or *positional parameters*.

2.3.1 Positional Parameters

In non trivial chess engines, it is usually the positional parameters which play an important role in evaluation and give the engine an edge during gameplay. With the amount of computing resources available today, these parameters when combined with fast pruning and deepening techniques can easily achieve IM[6] if not the GM[7] level. Therefore it is extremely important to incorporate them into our program if we want to build a competitive engine. Some of the most important ones are listed below:

- **Castling**: It is very important for the king to be defended by friendly pieces in the opening and middle games. Pawns in the corner serve as excellent defenders.
- **Rook on an open file**: Rook is the second most powerful piece on the board after the queen and has a long reach. But this reach is useless if there are other pieces blocking its way. Therefore rooks should ideally be placed on open files which helps a player to initiate an attack.
- **Rook on a semi-open file**: A semi-open file is one in which there are only enemy pawns blocking the rook. Placed onto a semi-open file, a rook does not

[6]International Master
[7]Grandmaster

allow the opponent to leave his pawn unprotected, thus reducing the mobility of his pieces.

- **Knight's mobility**: A knight's mobility is directly dependant on its placement. A knight near the center of the board is far more valuable than one at the corners.
- **(Supported) knight/bishop outpost**: Outposts are those squares on the opponent's side of the board where a piece cannot be attacked immediately. These squares act as launch points for a mating attack.
- **Bishop pair**: A bishop pair is generally very useful in end games when there are few pieces left on the board. They are generally used in tandem to force the king out of outposts, break pawn chains and hinder the movement of passed pawns.
- **Center pawns (*d4, d5, e4, e5*)**: The main theme during openings is *center control* which results in healthy development of the minor pieces and hinder the opponent's development. This is largely achieved using the 'd' an 'e' pawns.
- **Doubled pawns**: A good pawn structure is very important during middle and end games. Doubled pawns are usually considered to be a weakness. When doubled, the upper pawn blocks the lower and in many cases they need to be defended by a piece.
- **Backward pawn**: A backward pawn generally holds an entire pawn chain in place. It is the pawn of a chain which is nearest to the back rank. Usually another piece must guard this against attacks which can be disadvantageous.

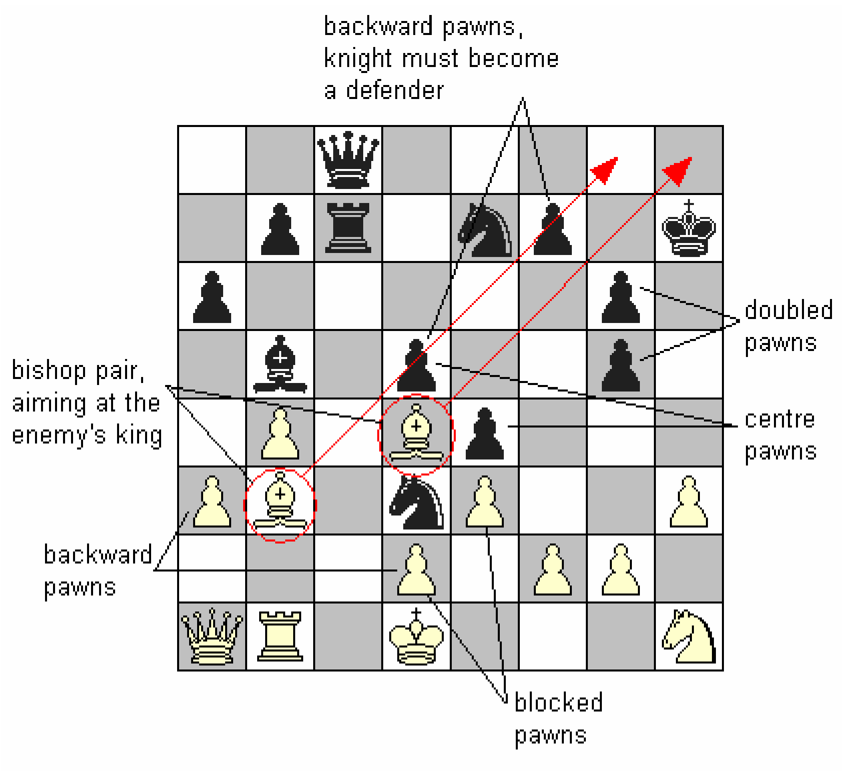


Figure 2.6: Positional Factors involving minor pieces

- **Rook(s) on the 7^{th} rank**: Rook(s) on the 7^{th} rank impede the movement of the opponent's pieces and also act as a very strong launch pad for the start of a mating attack as the opponent's king will generally be confined to the 8^{th} rank.

- **Connected rooks**: When two rooks are on the same file/rank without pawns or pieces between them, they are said to be connected. Connected rooks are untouchable as they support each other and can cause huge damage.

- **Passed pawn**: A pawn is said to be passed when there are no enemy pawns on its file or on either of the adjacent files. This pawn is usually considered very important as it has a high probability of getting promoted during end games.

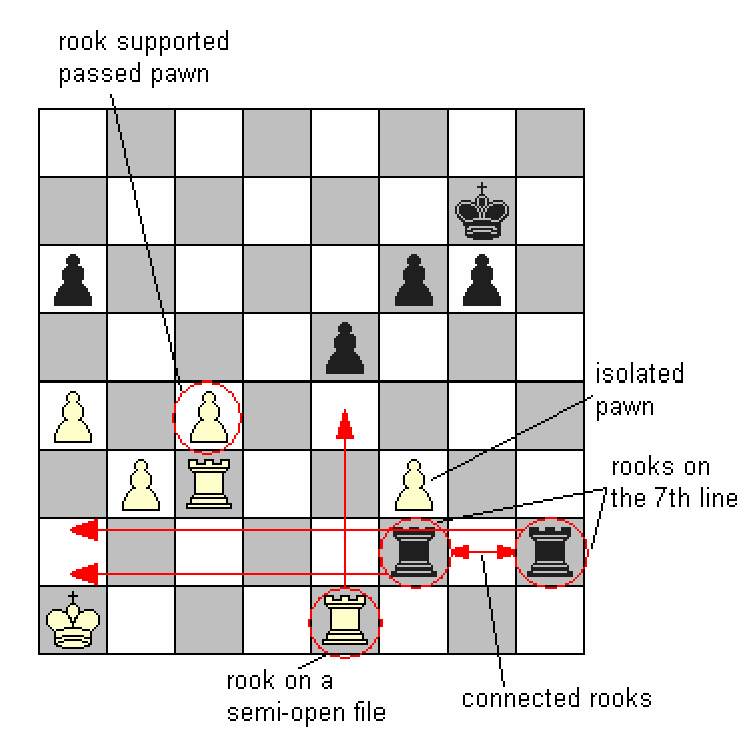


Figure 2.7: Positional Factors involving major pieces

- **Rook-supported passed pawn**: When there is a passed pawn on the board, a general rule of thumb is to put a rook behind it. This prevents the opponent from capturing the pawn easily and forces him to block the pawn with a piece.

- **Isolated pawns**: An isolated pawn has no friendly pawns on either of the adjacent files. This becomes a weak point in the game as it requires constant support from one of the other pieces.

- **Bishops on the large diagonals**: *a1-h8* and *h1-a8* are known as the large diagonals. When bishops are placed along these diagonals, they can be tucked into corners while maintaining their reach. This is greatly used to mount an attack on the enemy king during closed[8] middle games.

There are many other parameters which can be considered like pawn structures and closeness of positions. But the ones mentioned above can be easily represented using PVTs[9] which will greatly simplify the evaluation and learning process which is the main aim of this thesis.

[8]A position is said to be closed when there are 6 or more pawns occupying the 16 central squares

[9]Positional Value Tables

Chapter 3

Initial Ideas

Before genetic algorithms, we tried many other techniques which are considered to be good for machine learning. Since our main intention was to make a computer learn chess without any external help, our choices were limited to techniques which have the ability to interpret data and recognize underlying abstract patterns or features without human intervention.

That is, given a raw position the computer must be in a position to analyze its goodness without us having to tell the computer how to break this position down into a quantifiable features which can then be compared or learnt. In the next 2 sections, we briefly introduce *Neural Networks* and *Deep Learning* mechanisms which were used initially as learning tools and also explain why these techniques failed.

3.1 Neural Networks

3.1.1 Neuron

Artificial Neural Networks are inspired from Biology and extensively used in Pattern Recognition and Machine Learning. The fundamental building block for neural networks is a *neuron*. A neuron basically acts as an independent processing unit. Each neuron has a set of input links from other neurons, a set of output links to other neurons and an *activation threshold*. If the input is greater than this threshold, the neuron is activated. Within neural systems it is useful to distinguish 3 types of neurons:

- Input neurons which receive data from external sources
- Hidden neurons whose input and output remain within the network
- Output neurons which send data out of the neural network

These neurons are connected together in some fashion to from neural networks. Each connection between neurons are associated with weights which are the primary means of long-term storage in neural networks, and learning usually takes place by updating these weights.

3.1.2 McCulloch-Pitts Model

There are 2 equations which represent the McCulloch-Pitts model for a neuron. They are:

$$\zeta = \sum_i w_i \cdot x_i \tag{3.1}$$

$$y = \sigma(\zeta) \tag{3.2}$$

where ζ is the weighted sum of the inputs (the inner product of the input vector and the weight vector) and $\sigma(\zeta)$ is a function of the weighted sum. If we recognize that the weight and input elements form vectors w and x, the ζ weighted sum becomes a simple dot product:

$$\zeta = W \cdot x \tag{3.3}$$

This may be called as either the *activation function* (in the case of a threshold comparison) or a *transfer function*. In neurons, the division between the process of calculating the input sum using the weight vecto, and the calculation of the output value using the activation function may not be made explicitly.

The inputs to the network, x, come from an input space and the system outputs are part of the output space. For some networks, the output space Y may be as simple as $\{0, 1\}$ or it may be a complex multi-dimensional space. Neural networks tend to have one input per degree of freedom in the input space and one output per degree of freedom in the output space. The weight vector is updated during training by various algorithms of which the *Backpropagation Algorithm* is the most popular [3].

3.1.3 Backpropagation

The backpropagation algorithm is one of the most popular and robust tools in the training of artificial neural networks. Backpropagation passes error signals backwards through the network during training to update the weights of the network. To make things clear, it is useful to define a term called *interlayer* to be a layer of neurons and the corresponding input weights to that layer. We use a superscript to denote a specific interlayer, and a subscript to denote the specific neuron from within that layer. For instance:

$$\zeta_j^l = \sum_{i=1}^{N^{l-1}} w_{ij}^l x_i^{l-1} \tag{3.4}$$

$$x_j^l = \sigma(\zeta_j^l) \tag{3.5}$$

where x_i^{l-1} are the outputs from the previous interlayer (the inputs to the current interlayer), w_{ij}^l is the weight from the i^{th} input from the previous interlayer to the j^{th} element of the current interlayer. N^{l-1} is the total number of neurons in the previous interlayer.

The backpropagation algorithm specifies that the weights of the network are updated iteratively during training to approach the minimum of the error function. This is done through the following equations:

$$w_{ij}^l[n] = w_{ij}^l[n-1] + \delta w_{ij}^l[n] \tag{3.6}$$

$$w_{ij}^{l-1}[n] = \eta \delta_j^l x_i^{l-1}[n] + \mu \Delta w_{ij}^l[n-1] \tag{3.7}$$

The relationship between this algorithm and the gradient descent algorithm should be immediately apparent. Here, η is known as the learning rate and affects the rate of convergence of the algorithm. If the learning rate is too small, the algorithm will take a long time to converge. If the learning rate is too large, the algorithm might oscillate or diverge.

μ is known as the momentum parameter. The momentum forces the search to take into account the direction of movement from the previous iteration. By doing so, the system will tend to avoid local minima and approach the global minimum. The parameter δ is what makes this algorithm a 'back propagation' algorithm. We calculate it as follows:

$$\delta_j^l = \frac{dx_j^l}{dt} \sum_{k=1}^{r} \delta_k^{l+1} w_{kj}^{l+1} \tag{3.8}$$

The δ function for each layer depends on the from the previous layer. For the special case of the output layer (the highest layer), we use this equation instead:

$$\delta_j^l = \frac{dx_j^l}{dt}(x_j^l - y_j) \tag{3.9}$$

In this way, the signals propagate backwards through the system from the output layer to the input layer [2].

3.1.4 Implementation

Initially we generated training data by logging each position considered by Stockfish [13] and its evaluation scores while playing 5 minute games against itself. Generally a good engine like Stockfish searches the game tree till it reaches a depth of 16. This means that on average, before making any move, the engine considers anywhere between 10^3 and 10^5 positions. Therefore it was possible for us to obtain a huge database of positions by making Stockfish play a tournament of about 100 games.

The positions were recorded in FEN[1] format which could not be used as inputs to the network directly. Therefore, each position was converted into a binary string as follows:

- Since there are 12 unique pieces in chess, it is sufficient to use 4 bits to identify

[1]Forsyth-Edwards Notation

them. For e.g. white king: 0001, white queen: 0010 and so on. Empty squares were denoted by 0000.

- 6 bits were reserved for representing various nuances such as *en passant* availability, castling status, turn to move, etc

This resulted in a binary string of length 262 ($4 \times 64 + 6$) which could now be supplied as input to the network. A 2-layer backpropagation neural network with 262 inputs and 1 output was created and trained. After learning was complete, the evaluation module of Stockfish was replaced by our network and tournaments were conducted between the original and modified engines.

It was noted that, even though the network learned some abstract concepts like the importance of the queen over a rook, the gameplay was vague since position evaluation did not even come close to what was expected. The rate of learning was extremely slow and it took 48 hours of training to learn the aforementioned queen-rook importance. A 3-layer network showed no improvement.

3.2 Deep Learning

The computations involved in producing an output from an input can be represented by a flow graph: a flow graph is a graph representing a computation, in which each node represents an elementary computation and a value which is the result of the computation and is usually applied to the children of that node. The set of computations allowed in each node and possible graph structures defines a family of functions. Input nodes have no children. Output nodes have no parents.

A particular property of such flow graphs is depth: the length of the longest path from an input to an output. Traditional feedforward neural networks can be considered to have depth equal to the number of layers (i.e. the number of hidden layers plus 1, for the output layer). Support Vector Machines (SVMs) have depth 2 (one for the kernel outputs or for the feature space, and one for the linear combination producing the output).

Depth 2 is enough in many cases (e.g. logical gates, formal [threshold] neurons, sigmoid-neurons, Radial Basis Function [RBF] units like in SVMs) to represent any function with a given target accuracy. But this may come with a price: that the required number of nodes in the graph (i.e. computations, and also number of parameters, when we try to learn the function) may grow very large. Theoretical results showed that there exist function-families for which in fact the required number of nodes may grow exponentially with the input size.

We can see deep architectures as a kind of factorization. Most randomly chosen functions can't be represented efficiently, whether with a deep or a shallow architecture. But many that can be represented efficiently with a deep architecture cannot be represented efficiently with a shallow one (see the polynomials example in [20]). The existence of a compact and deep representation indicates that some kind of structure exists in the underlying function to be represented. If there was no structure whatsoever, it would not be possible to generalize well [12].

3.2.1 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are variants of MLPs. From Hubel and Wiesel's early work on the cat's visual cortex [37], we know there exists a complex arrangement of cells within the visual cortex. These cells are sensitive to small sub-regions of the input space, called a receptive field and are tiled in such a way as to cover the entire visual field. These filters are local in input space and are thus better suited to exploit the strong spatially local correlation.

CNNs exploit spatially local correlation by enforcing a local connectivity pattern between neurons of adjacent layers. The input hidden units in the m^{th} layer are connected to a local subset of units in the $m-1^{th}$ layer, which have spatially contiguous receptive fields. We can illustrate this graphically as follows:

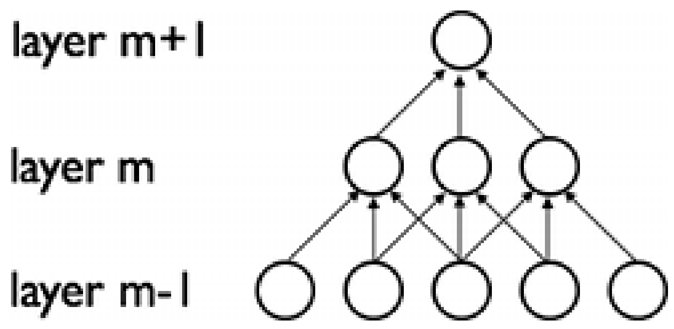


Figure 3.1: A Convolutional Neural Network

Imagine that layer $m-1$ is the input retina. In Fig. 3.1, units in layer m have receptive fields of width 3 with respect to the input retina and are thus only connected to 3 adjacent neurons in the layer below. Units in layer m have a similar connectivity with the layer below. We say that their receptive field with respect to the layer below is also 3, but their receptive field with respect to the input is larger (it is 5).

The architecture thus confines the learnt 'filters' (corresponding to the input producing the strongest response) to be a spatially local pattern (since each unit is unresponsive to variations outside of its receptive field with respect to the retina). As shown above, stacking many such layers leads to 'filters' (not anymore linear) which become increasingly 'global' (i.e. spanning a larger region of pixel space). For example, the unit in hidden layer $m + 1$ can encode a non-linear feature of width 5.

In CNNs, each sparse filter is additionally replicated across the entire visual field. These 'replicated' units form a *feature map*, which share the same parametrization, i.e. the same weight vector and the same bias. Replicating units in this way allows for features to be detected regardless of their position in the input field. Additionally, weight sharing offers a very efficient way to do this, since it greatly reduces the number of free parameters to learn. By controlling model capacity, CNNs tend to achieve better generalization especially in vision problems [7].

3.2.2 Autoencoders

An autoencoder takes an input $x \in [0,1]^d$ and first maps it (with an encoder) to a hidden representation $y \in [0,1]^{d'}$ through a deterministic mapping, e.g.:

$$y = s(Wx + b) \tag{3.10}$$

where s is a non-linearity such as the sigmoid. The latent representation y, or code is then mapped back (with a decoder) into a reconstruction z of same shape as x through a similar transformation, e.g.:

$$z = s(W'y + b') \tag{3.11}$$

where $'$ does not indicate transpose and z should be seen as a prediction of x, given the code y. The weight matrix W' of the reverse mapping may be optionally constrained by $W' = W^T$, which is an instance of tied weights. The parameters of this model (namely W, b, b' and W' if we do not use *tied weights*) are optimized such that the average reconstruction error is minimized. The reconstruction error can be measured in many ways, depending on the appropriate distributional assumptions on the input given the code, e.g., using the traditional squared error $L(x, z) = \|x - z\|^2$, or if the input is interpreted as either bit vectors or vectors of bit probabilities by the reconstruction cross-entropy defined as:

$$L_H(x, z) = -\sum_{k=1}^{d} [x_k \log z_k + (1 - x_k) \log (1 - z_k)] \tag{3.12}$$

The hope is that the code y is a distributed representation that captures the coordinates along the main factors of variation in the data (similarly to how the projection on principal components captures the main factors of variation in the data). Due to the fact that y is viewed as a lossy compression of x, it cannot be a good compression (with small loss) for all x, so learning drives it to be one that is a good compression in particular for training examples, and hopefully for others as well, but not for arbitrary inputs. That is the sense in which an auto-encoder generalizes: it gives low reconstruction error to test examples from the same distribution as the training examples, but generally high reconstruction error to uniformly chosen configurations of the input vector.

If there is one linear hidden layer (the code) and the mean squared error criterion is used to train the network, then the k hidden units learn to project the input in the span of the first k principal components of the data. If the hidden layer is non-linear, the auto-encoder behaves differently from PCA[2], with the ability to capture multi-modal aspects of the input distribution. The departure from PCA becomes even more important when we consider stacking multiple encoders (and their corresponding decoders) when building a deep auto-encoder [34] [9].

3.2.3 Implementation

When shallow neural networks did not show promising results, the reason we attributed for the failure was the absence of an explicit feature extraction process which was omitted deliberately as we did not want the network to have any preconceptions about what it was learning. Therefore deep learning techniques were the obvious choice as they are essentially neural networks with a built-in feature extractor. Another factor influencing this decision was the fact that the features become more abstract as we go deeper into the network. This is how our brain constructs and understands complex concepts.

Initially 3 CNNs were constructed, each for one stage in the game; opening, middle game and end game. This is because the aim of a player at different stages of the game varies and it is not feasible for one network to learn everything. The idea here was that no matter what the position is, attacks and defense always follow a particular pattern. During attacks the king ring is hit at least by 2 or more pieces. A good defense consists of king safety, a successful castle, strong pawn structures near the king and so on. It is easy to see that all of these happen locally i.e. within some quadrant on the board.

As an example, let us take the king ring parameter. No matter where the king

[2]Principle Component Analysis

is on the board, he is said to be safe with good confidence if the ring is free from threats. Since CNNs learn spatially local patterns very well, it was expected to learn the underlying features of such positions if they exist. Again, due to the lack of computing power and training time, the networks' learning rate was low.

A similar process was carried out with Autoencoders and the same problem recurred. This failure made us think deeper into the root cause of the problem; we were asking the network to recapitulate centuries of chess theory with limited data in very limited time just to learn what we already know. Therefore we decided to formulate the problem with an entirely different perspective:

- avoid spoon feeding the machine with code to identify features or parameters
- provide a platform which guides the machine to learn the right parameters and ignore the rest

This led us to Genetic Algorithms.

Chapter 4

Proposed System

4.1 Genetic Algorithms

Genetic Algorithms aim mainly at solving optimization problems by means of applying the principle of natural selection seen in living organisms. Genetic Algorithms borrows heavily from phenomena such as adaptability of life to environmental changes, inheritance of the vital properties by descendants and of course natural selection, following the 'survival of the fittest' paradigm from Darwin's theory of evolution [25]. In scientific literature, the idea of mimicking evolution to form genetic algorithms was first proposed by John Holland in 1975 [35]. In his work, Holland suggested a schematic of how a genetic algorithm would look like. In 1989, David Goldberg created a simple Genetic Algorithm [32] and its first famous computer implementation (using Pascal).

Natural Genetics	**Genetic Algorithms**
Phenotype	A set of parameters to be optimized
Genotype	A population of individuals
Chromosome	Individual
Gene	A parameter from the set
Locus	Position of the parameter (index)
Allele	Value of the parameter

Table 4.1: Comparison of Genetic Terminology

4.1.1 Terminology

In biological systems which act as an inspiration for genetic algorithms, *chromosomes* are seen as the fundamental building blocks. The set of chromosomes of a living organism is called the *genotype* and the organisms which posses a particular genotype are called the *phenotype.* The parts constituting a chromosome are referred to as *genes*, which are located on different *loci.* Each gene controls the inheritance of one or several *alleles.* In artificial genetic systems a different terminology is accepted [32]. A comparison between the corresponding terms is given in Table 4.1.1.

4.1.2 Background

The basic component of a genetic algorithm is the *individual*[1]. An individual is a potential solution to the problem at hand. A collection of many such individuals form a *population.* Each individual is usually a string of values, each of which represents a

[1]Here the words chromosome and individual are used interchangeably

parameter which is being optimized. Each value is called a *feature* and is identified by an index. Features can be whatever makes sense for the problem being solved.

For example, in [42] a function maximization problem is considered with two variables and binary chromosomes[2] are used to represent these variables. In [24] integer chromosomes are used to solve the N-Queens problem and in [28] the chromosome is a specially designed data structure used to solve a job scheduling problem. Each of these articles solve different problems using an identical algorithm with the difference being in how they represent potential solutions.

When applying genetic algorithms, the chromosome acts as a storage of features i.e. it entirely describes a potential solution. Genetic algorithms operate on populations in a temporal manner. A population at any given point in time is called a *generation*. Each individual in a generation is evaluated using some fitting criterion. This criterion is unique to the problem at hand. The fittest individuals are allowed to reproduce by means of genetic operators and will result in the formation of a new generation.

As new generations are born, the individuals get fitter and hence tend towards optimality. This is why choosing the right parameters for the chromosome is considered to be the most important activity during the formulation of a genetic system. If the wrong parameters are chosen for optimization, the resultant values though optimized, will fail to solve the problem.

[2]Individuals whose parameters take binary values

4.1.3 Genetic Operators

In genetic algorithms, it is important for individuals to reproduce in a manner that will somehow imbibe their traits to the offspring while allowing them to have their own individuality. This helps the solutions to avoid local optima and eventually reach the global optimum. There are two classical genetic operators which allow individuals to do this; *Crossover* and *Mutation*.

At its simplest, crossover takes 2 chromosomes from the current population, selects an index randomly which is called the *crossover point* and interchanges the parameters of the parents from that point onwards. To understand the idea better, let us consider its visual representation. Assume x_1 and x_2 are 2 chromosomes of length '8'. Let us also assume that the crossover point is '3'. Now, the chromosome is divided into 2 parts as shown in Fig. 4.1. The first part of x_1 is attached to the second part of x_2 and vice versa to form 2 new individuals or offspring.

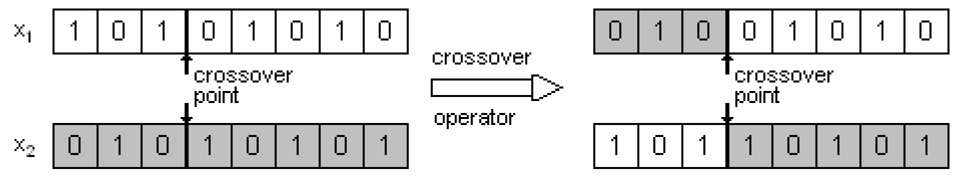


Figure 4.1: A Simple Crossover

Mutation is the process of changing some parameters in the chromosome to avoid recycling of parameters (after some generations of evolution, there might come a point where the parameters remain constant even after crossovers) which will ultimately result in the system being stuck in a local optimum. After obtaining offspring from crossover, some indices are chosen and the values of the parameters at those indices are changed according to some rule. In Fig. 4.2, a single mutation point is chosen (index '1') and hence the value of the first parameter is flipped.

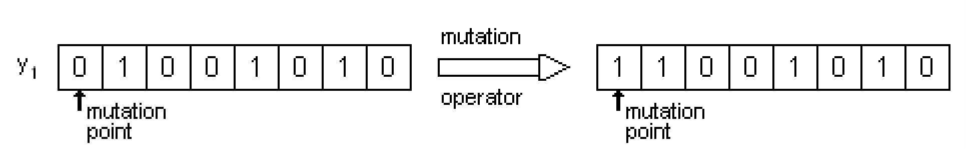


Figure 4.2: A Simple Mutation

There exist more complex variations of these genetic operators such as two-point crossover and two-point mutation. In two-point crossover, a chromosome is divided into 3 parts and the middle ones are exchanged. In two-point mutation, values at 2 positions are changed. This process can be extended for multi-point operators [42].

There is another genetic operator which is seldom used called *Inversion*. Simple inversion selects 2 points along the length of a chromosome and the parameters between these 2 points are reversed [42].

4.1.4 Evaluation and Selection

Before crossover and mutation are applied, each chromosome needs to be evaluated and the fittest among them should be separated. This process differs from problem to problem. Some of the most popular techniques are:

- Choose the fittest individuals and reject the rest
- Throw away the poorest performers and keep the rest
- Use some randomized heuristics such as Roulette wheel selection[3].

[3]Here, better solutions have higher probability of selection

- Retain the best solution(s) as it is and modify others (called the *Elitist model*).

There are also other static and dynamic selection methods in which the selection probabilities remain constant and vary over the generations, respectively [42]. We can also develop selection procedures tailor made for our problems. One element which remains constant in all these techniques is a function which evaluates the goodness of each potential solution. This function is called the *fitness function*. Every solution is analyzed and given a *fitness value* which determines whether it is eligible for reproduction or not.

4.2 Implementation

4.2.1 CuckooChess

CuckooChess [44] is an advanced free open source chess program under the GNU General Public License written in Java by Peter Osterlund. It contains many of the standard algorithms for computer chess discussed previously such as iterative deepening, quiescence search with SEE pruning, MVV/LVA move ordering, hash table, history heuristic, recursive null moves, opening book and magic bit boards. It also uses some advanced techniques like Negascout, aspiration windows, futility pruning and late move reductions [8]. We have cloned this engine and modified it to suit our needs. These modifications include:

- Replacing the entire evaluation module with our new module
- Adding a genetic training system with a tournament selector

- Adding PVTs[4] which is the core of the new evaluation function
- Modifying search behavior to take advantage of the new PVTs
- A tournament simulator which is used for both fitness evaluation and testing

Each point is explained in detail in the future sections. Fig. 4.3 is a flowchart which gives an overview of what is going on under the hood.

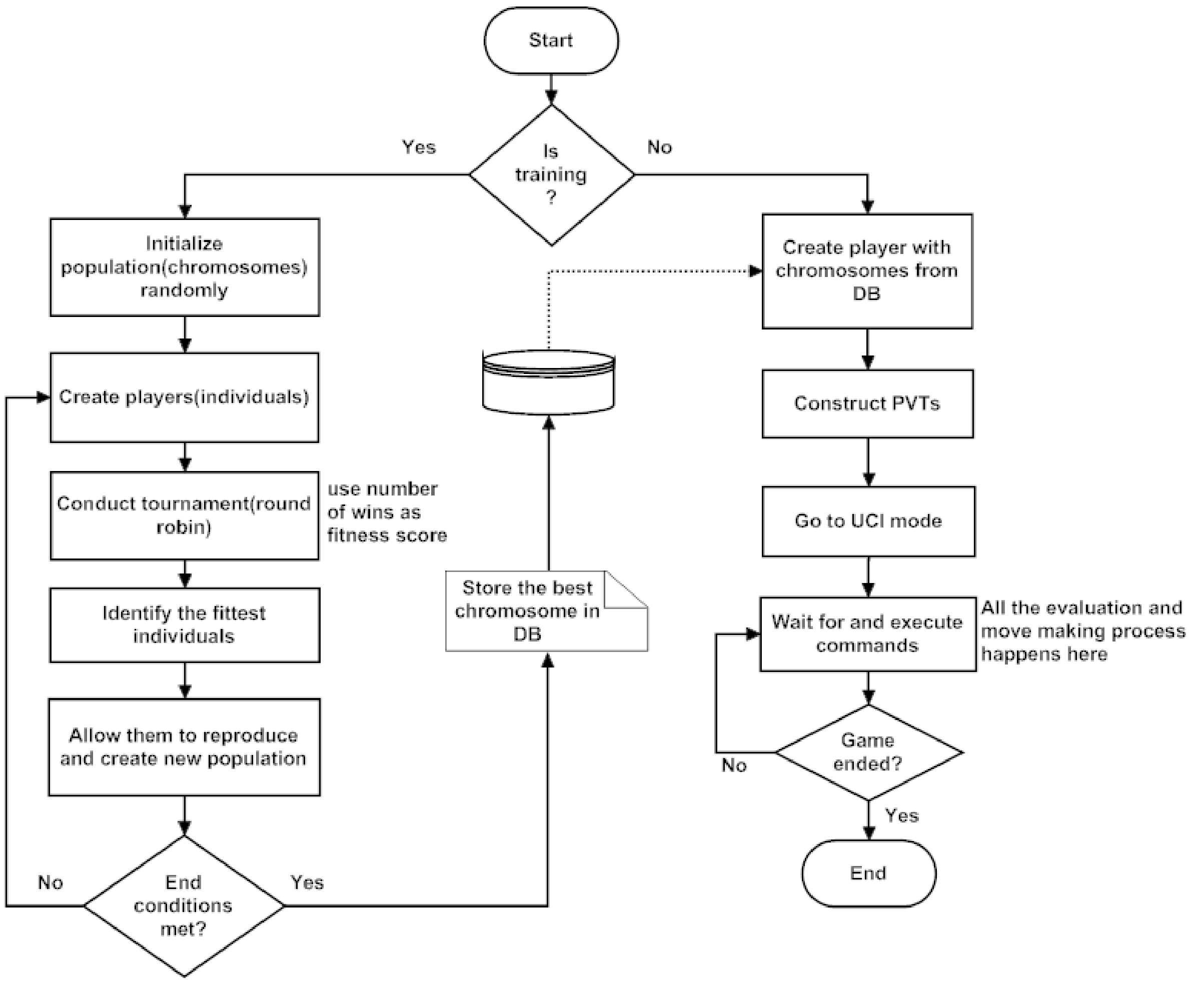

Start
Is training ?
Yes
No
Initialize population(chromosomes) randomly
Create players(individuals)
Conduct tournament(round robin)
use number of wins as fitness score
Identify the fittest individuals
Allow them to reproduce and create new population
End conditions met?
No
Yes
Store the best chromosome in DB
Create player with chromosomes from DB
Construct PVTs
Go to UCI mode
Wait for and execute commands
All the evaluation and move making process happens here
Game ended?
No
Yes
End

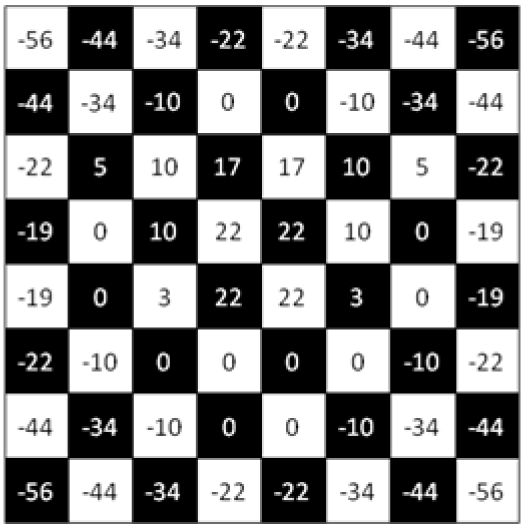
Figure 4.3: Workflow

[4]Positional Value Tables

4.2.1.1 Formulation Phase

Usually evaluation functions consider almost if not all the parameters discussed in the Positional Parameters section. But we wanted the evaluation to be totally dependent on whatever the computer learns on its own and nothing else. Therefore we needed to formulate the problem in such a way that the engine itself recognizes the relative importance of each parameter. This requirement forced us to use PVTs.

A PVT is a grid of numbers which indicate the best squares for a piece to occupy. Greater the number, better is its position. For example, Fig. 4.4 shows the PVT for a black knight. It is easy to spot that the values at the center of the board are higher than those at the corners. This means it is desirable for the knight to be on the central squares rather than the corners.



-56	-44	-34	-22	-22	-34	-44	-56
-44	-34	-10	0	0	-10	-34	-44
-22	5	10	17	17	10	5	-22
-19	0	10	22	22	10	0	-19
-19	0	3	22	22	3	0	-19
-22	-10	0	0	0	0	-10	-22
-44	-34	-10	0	0	-10	-34	-44
-56	-44	-34	-22	-22	-34	-44	-56

Figure 4.4: PVT for a black knight during endgame

By carefully constructing these PVTs, we can quantify most of the positional parameters. Even though some of the abstract ones such as rook behind a past pawn

Piece	**Middle Game**	**End Game**
Pawn	✓	✓
Rook	✓	×
Knight	✓	✓
Bishop	✓	✓
Queen	✓	×
King	✓	✓

Table 4.2: PVTs

cannot be described using this technique alone, combining this with other search algorithms it is possible to get more accurate evaluation scores. This is achieved by using PVTs for move ordering by arranging moves in descending order of the PVT values for the destination squares.

It is easy to see that each piece must have its own PVT. Also, most of the pieces have different responsibilities at different junctures of the game. Therefore it is necessary to maintain different PVTs for middle and end games (PVTs for openings are not required as moves are made using opening books). Table 4.2 shows the various PVTs that are maintained in our engine.

Every potential solution (player/individual in the population) should have a chromosome which will influence its gameplay. A good chromosome will often lead to victory where as a bad chromosome will result in a loss. In our program, each value in every PVT is considered as a parameter to be optimized. Therefore, all the PVTs are joined together to form a chromosome as shown in Fig. 4.5.

This is essentially a single dimension array containing 640 floating point numbers

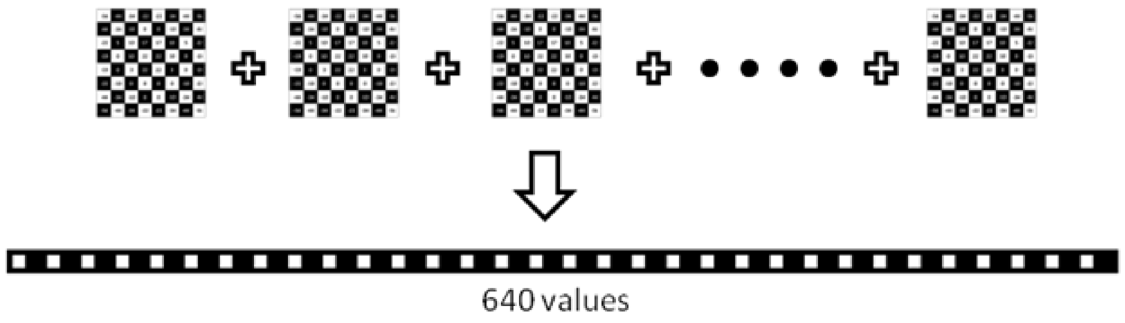


Figure 4.5: PVTs to Chromosome

(we club together the 10 PVTs mentioned in Table 2, each having 64 values). From here on, the problem of learning reduces to an optimization of these 640 values.

4.2.1.2 Optimization Phase

First, a population of 20 players is created. The players' chromosomes are randomly initialized. At this stage, the players make random and losing moves. They are pit against each other in a round-robin tournament where each player plays at least 3 games and the results are recorded. The players get 1 point for a win, 0.5 for a draw and 0 for a loss. After the tournament is finished, the results are tabulated and these points are used as an indicator of their chromosomes' fitness level.

The winner and the runner-up are called *elites* as they are the best solutions found in the current generation and they are retained as it is for the next generation. The other players in the top 10 are allowed to reproduce among themselves using single-point crossovers with a probability of 0.8 to create 18 offspring and the previous generation is discarded. Next all the new players except the elites undergo randomized mutation with a probability of 0.02. The elites and the offspring form the new generation. This process is repeated till any of the following end conditions are met:

- 1000 generations have passed
- The best solution remains the same for 10 or more generations
- The rate of change in the best solution chromosome structure is below 1% for 20 or more generations

Fig. 4.6 clearly illustrates and summarizes the entire process. One problem in this technique is that, it always finds a single solution. But as we know, there is no one right way to play chess. The style of a player who plays attacking chess cannot be compared to that of a player who prefers defensive positional play. Both may be equally good. It is therefore safe to assume that, a mathematical function representing position evaluation (if it exists) is multimodal. Hence it is desirable for us to get solutions representing most if not all the modes of the function.

```
Player evolve() {
      Player[] players = createPlayers();
      While(!endConditions()) {
            playTournament(players);
            Player[] elites = selectElites();
            Player[] winners = selectWinners();
            Player[] offspring = reproduce(winners);
            mutate(winners);
            mutate(offspring);
            players = newGeneration(elites, offspring);
      }
      return bestPlayer(players);
}
```

Figure 4.6: Evolution Process

The problem with simple genetic algorithms is that it will eventually converge to one of the many global optima (if they do not get trapped in local optima) which depend on the initial population and the random genetic drift [45] occurring throughout the run. Eventually we will get copies of the same individual in one of the valleys/plateau.

For instance, consider a simple function $f(x) = sin(x^2)$. A plot of the function along with the position of the individuals (red marks) trying to find the minimum is shown in Fig. 4.7 [15].

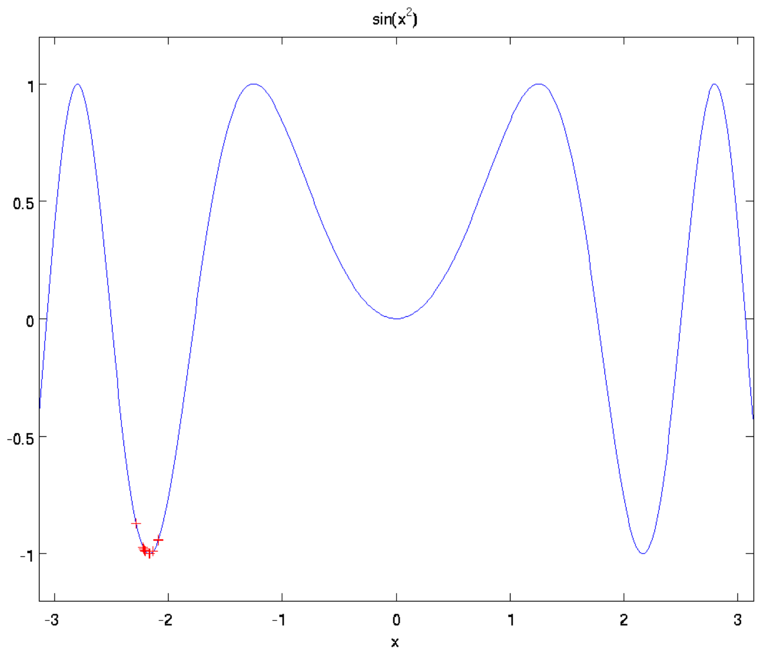


Figure 4.7: Potential solutions using simple GA

As we can see this function has more than one minimum and we want our algorithm to find the other minimum as well. This is where the concept of *niching* comes in handy. Niching is a general class of techniques that promote the formation and maintenance of stable sub-populations in a genetic algorithm. 2 main objectives of such techniques are:

- To converge to multiple, highly fit, and signicantly different solutions (for multimodal optimization)
- To slow down convergence in cases where only one solution is required (to avoid premature convergence)

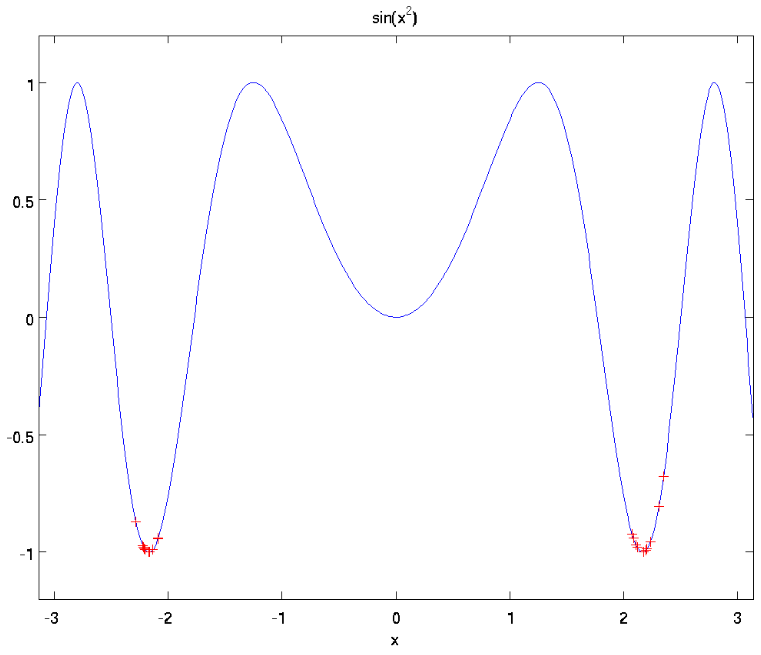


Figure 4.8: Potential solutions using niching

Fig. 4.8 shows how individuals have converged onto 2 separate optima for $f(x) = sin(x^2)$ when using niching. There are many approaches to niching and the one used here is called *multi-niche crowding (MNC).*

4.2.1.3 Multi-Niche Crowding

Crowding [26] is a generalization of pre-selection. In crowding, the selection and reproduction processes are the same as those carried out in simple genetic algorithms, but the replacement process is different. Let us assume that 2 parents produce 2 offspring. In order to make room for the newborns, it is necessary to identify 2 members from the population for replacement. The policy of replacing a member of the present generation by an offspring is carried out as follows:

- A group of C individuals is selected at random from the population. C is called the crowding factor and a value 2 or 3 appears to work well in [26].
- The chromosomes of the offspring are compared with those of the C individuals in the group using Hamming distance as a measure of similarity. The group member which is most similar to the offspring is replaced by the offspring.
- This procedure is repeated for the other offspring as well.

Crowding is essentially a successive replacement strategy. This strategy maintains the diversity in the population and postpones premature convergence. However generic crowding cannot maintain stable subpopulations for long due to *selection pressure*[5].

In multi-niche crowding (MNC), both selection and replacement steps are modified with some type of crowding. The idea is to eliminate the selection pressure caused by *fitness proportionate reproduction (FPR)* while allowing the population to maintain some diversity. This objective is achieved in part, by encouraging mating

[5]Any cause that reduces reproductive success in a portion of the population

and replacement within members of the same niche while allowing for some competition for slots among the niches. The result is an algorithm that (a) maintains stable subpopulations within different niches, (b) maintains diversity throughout the search, and (c) converges to different optima.

In MNC, the FPR selection is replaced by what is called *crowding selection*. In crowding selection, each individual in the population has the same chance for mating in every generation. Application of this selection rule takes place in two steps. First, an individual A is selected for mating. This selection can be either sequential or random. Second, its mate M is selected, not from the entire population, but from a group of individuals of size C_s, picked at random (with replacement) from the population. The mate M thus chosen must be the one who is the most 'similar' to A. The similarity metric used here is not a *genotypic* metric such as the Hamming distance, but a suitably defined *phenotypic* distance metric. Crowding selection promotes mating between individuals from the same niche while allowing mating between individuals from different niches.

During the replacement step, MNC uses a replacement policy called *worst among the most similar*. The goal of this step is to pick an individual from the population for replacement by an offspring. Implementation of this policy follows these steps. First, C_f groups are created by randomly picking s individuals (with replacement) per group from the population. These groups are called *crowding factor groups*. Second, one individual from each group that is most phenotypically similar to the offspring is identified. This gives C_f individuals that are candidates for replacement by virtue of their similarity to the offspring that will replace them. From this group of most similar individuals, we pick the one with the lowest fitness to die and that slot is filled with the offspring. The offspring could possibly have a lower fitness than the individual being replaced [22].

Thus, we replace our vanilla genetic algorithm with MNC which helped us find multiple optimal solutions which are then stored in a database of chromosomes (the details of best chromosome found in all the runs can be found in Appendix A). When the chess engine is asked to play, it selects one from the database and re-forms PVTs from the chromosome and uses it to make moves.

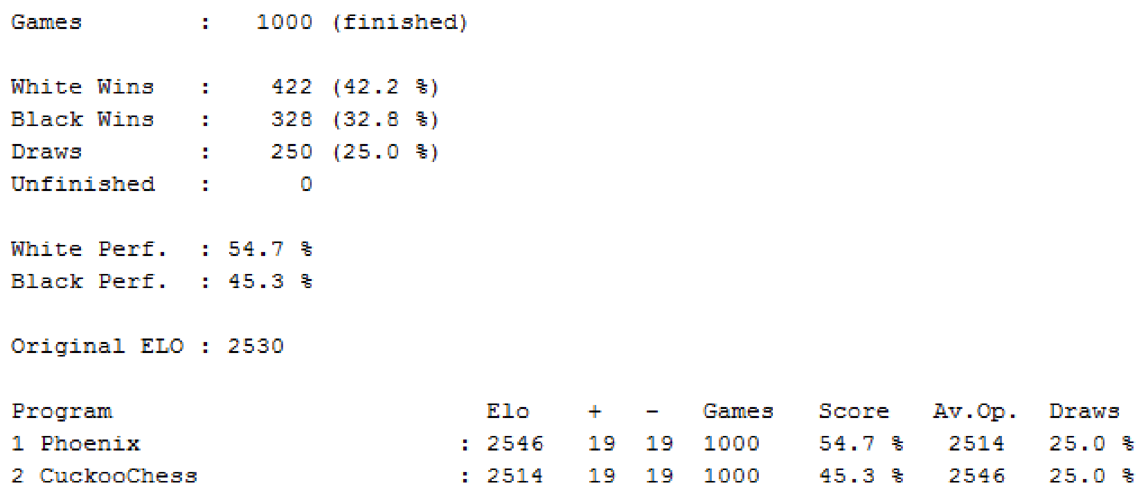
4.3 Results

The best way to test a chess engine is to make it play against other engines. This can be daunting and painfully slow if there is no common language through which the engines can communicate. In chess programming circles, there exist 2 protocols which are used as standards while building chess engines. They are:

- Universal Chess Interface (UCI)
- Chess Engine Communication Protocol (used in XBoard and WinBoard)

UCI is more robust and is supported by most of the prominent engines today. Therefore our engine also communicates through UCI. More information about this protocol can be found in [14]. Many chess GUIs also are UCI compatible and hence it is easy to plug our engine into a GUI such as Arena [1] and actually see the games being played, rather than read the PGN.

Our engine was tested against the original CuckooChess engine as it would provide us with a clear benchmark about any improvements achieved. A total of 1000 games were played and recorded with a time control of 3 seconds per move. The games were then analyzed using the EloStat algorithm developed by Frank Schubert. This algorithm calculates the Elo rating [10] of a player provided the rating of the opponent is known. The results are tabulated below.



```
Games         :   1000 (finished)

White Wins    :    422 (42.2 %)
Black Wins    :    328 (32.8 %)
Draws         :    250 (25.0 %)
Unfinished    :      0

White Perf.   : 54.7 %
Black Perf.   : 45.3 %

Original ELO  : 2530

Program                          Elo    +   -   Games   Score   Av.Op.  Draws
1 Phoenix                      : 2546   19  19  1000    54.7 %   2514   25.0 %
2 CuckooChess                  : 2514   19  19  1000    45.3 %   2546   25.0 %
```

Figure 4.9: EloStat Result

CuckooChess is rated at 2530 according to CCRL (Computer Chess Ratings List). After 1000 games, it was seen that our modified engine outperformed its parent with a rating of 2546 (an increase of 19 points). This rating puts our engine in the 'International Grandmaster' category. A link to the PGN of all the games played can be found in Appendix A.

Even though the increase in rating seems small, it should be noted that the rise in rating is tapered off gradually when large number of games are played with an opponent of similar strength. It can be seen that our engine has won 422 games compared to the 328 games won by CuckooChess. This indicates that there is a considerable increase in the strength of our modified engine.

Also it has to be noted that the solution we obtained is not optimal. It can be seen clearly in the chromosome (see Appendix A), where some parts of the PVTs seem random. This is because it is not possible for the algorithm to find out which mutations will help and which do not. This will result in the replacement of some good values with random ones. Therefore there is still room for improvement in the engine if we can find a way to mutate only bad genes.

Chapter 5

Conclusion

This thesis provides a new perspective for designing complex self-learning programs whose purpose is to mimic human behavior. As we can see in the results section, the engine is not a world class one by any stretch of imagination. But it was never meant to be. The intention was to find a way to reduce abstract and complex concepts of chess into something that a computer understands.

Shallow and deep neural networks which are considered to be the future of machine learning and computer vision failed to give good results here probably because:

- The features are not spatially correlated which means that it is not easy to recognize patterns
- A universal mathematical function for chess evaluation does not exist. If it does, its dimensionality will be so high that it would require extremely large networks, huge training data and very long training phases for any learning to happen

- Learning happens only when the input given to networks have a direct relation with what is to be learnt. In chess however, no such relation exists among the squares, pieces and positions

The 'no external help' and 'no explicit feature extraction' rules which we abided by forced us to convert a learning problem into a problem of optimization. We found a clever way to formulate the problem in such a way that all the positional and other parameters were converted into a group of numbers using PVTs. Multimodal optimization of a large number of parameters seems to be an area where research has come to a standstill in recent years with many open questions left to be answered. Multi-Niching and Crowding which give us the best shot at solving such problems allowed us to optimize parameters efficiently.

This work concentrates only on learning chess evaluation. The underlying search algorithms remain the same. In the future, one can extend the engine by using some heuristic similar to PVTs which stores information about position patterns and eliminate or at least minimize the role of complex search techniques like iterative deepening. Also, one could increase the efficiency of search algorithms by pruning out moves using past experience and avoid futile evaluations which will speed up the engine.

Bibliography

[1] http://www.playwitharena.com/.

[2] Artificial neural networks/error-correction learning. http://en.wikibooks.org/wiki/Artificial_Neural_Networks/Error-Correction_Learning.

[3] Artificial neural networks/neural network basics. http://en.wikibooks.org/wiki/Artificial_Neural_Networks/Neural_Network_Basics.

[4] Board representation. https://chessprogramming.wikispaces.com/Board+Representation.

[5] Chess club. http://www6.chessclub.com/help/PGN-spec.

[6] Chess notation. http://en.wikipedia.org/wiki/Chess_notation.

[7] Convolutional neural networks. http://www.deeplearning.net/tutorial/lenet.html#lenet.

[8] Cuckoochess wiki. http://chessprogramming.wikispaces.com/CuckooChess.

[9] Denoising autoencoders. http://www.deeplearning.net/tutorial/dA.html#autoencoders.

[10] Elo rating system. http://en.wikipedia.org/wiki/Elo_rating_system.

[11] History of chess. http://en.wikipedia.org/wiki/History_of_chess.

[12] Introduction to deep learning algorithms. http://www.iro.umontreal.ca/pift6266/H10/notes/deepintro.html.

[13] Stockfish: Powerful open source chess engine. http://stockfishchess.org/.

[14] The uci protocol. http://wbec-ridderkerk.nl/html/UCIProtocol.html.

[15] What is niching scheme? http://stackoverflow.com/questions/13775810/what-is-niching-scheme.

[16] Wheat and chessboard problem. http://en.wikipedia.org/wiki/Wheat_and_chessboard_problem#Origin_and_story.

[17] P. Aksenov. Genetic algorithms for optimising chess position scoring. Master's thesis, University of Joensuu, 2004.

[18] V. Anand. The indian defense. http://www.chess.com/article/view/where-was-chess-invented.

[19] D. Beal. Experiments with the null move. pages 65–79, 1989.

[20] Y. Bengio. Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[21] H. Berliner. Chess as problem solving: The development of a tactics analyzer. Technical report, Carnegie-Mellon University, Pittsburgh,, 1974.

[22] W. Cedeno, V. Rao, and T. Slezak. Multi-niche crowding in genetic algorithms and its application to the assembly of dna restriction-fragments. Technical report, University of California, Davis.

[23] J. Condon and K. Thompson. Belle chess hardware. *Advances in Computer Chess*, (3):45–54, 1982.

[24] K. Crawford. Solving the n-queens problem using genetic algorithms. *Proceedings of ACM/SIGAPP Symposium on Applied Computing*, pages 1039–1047, 1992.

[25] C. Darwin. *The Origin of Species.* Murray, London,, 1859.

[26] K. DeJong. An analysis of the behavior of a class of genetic adaptive systems. Technical report, University of Michigan.

[27] C. Donninger. Null move and deep search: Selective search heuristics for obtuse chess programs. 16(3):137–143, 1993.

[28] E. Falkenauer and S. Bouffouix. A genetic algorithm for job shop. *Proceedings of IEEE International Conference on Robotics and Automation*, pages 824–829, 1991.

[29] P. Frey. Chess skill in man and machine. 1977.

[30] F. Friedel. A short history of computer chess. http://www.chessbase.com/columns/column.asp?pid=102.

[31] G. Goestch and M. Campbell. Experiments with the null-move heuristic. *Computers, Chess, and Cognition*, pages 159–168, 1990.

[32] D. Goldberg. Genetic algorithms in search, optimization and machine learning. 1989.

[33] E. Heinz. Scalable search in computer chess. 2000.

[34] G. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, July 2006.

[35] J. Holland. Adaptation in natural and artificial systems. 1975.

[36] F. Hsu. Behind deep blue: Building the computer that defeated the world chess champion. 2002.

[37] D. Hubel and T. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *Journal of Physiology*, 195:215–243, 1968.

[38] D. Knuth and R. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[39] R. Levinson. The role of chess in artificial intelligence research. Technical report.

[40] A. Linhares. Data mining of chess chunks: A novel distance-based structure, 2003.

[41] T. Marsland. Computer chess and search. 1991.

[42] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag, 1996.

[43] A. Newell, J. Shaw, and H. Simon. Chess playing programs and the problem of complexity. *IBM Journal of Research and Development*, pages 320–335, 1958.

[44] P. Osterlund. cuckoochess. https://code.google.com/p/cuckoochess/.

[45] A. Rogers and A. Prugel-Bennett. Genetic drift in genetic algorithm selection schemes. *IEEE Transactions on Evolutionary Computation*.

[46] C. Shannon, 1950.

[47] A. Turing, C. Bates, and B. Bowden. Digital computers applied to games. 1953.

[48] A. Turing, C. Strachey, M. Bates, and B. Bowden. *Digital Computers Applied to Games*. Pitman, 1953.

[49] A. Zobrist. A new hashing method with application for game playing. *ICCA Journal*, 13(2):69–73, 1990.

Appendix A

Appendix

A.1 Links

- Phoenix vs. CuckooChess PGN

  https://github.com/rahular/phoenix/blob/master/games/PhoenixGames1000.pgn

- Source Code

  https://github.com/rahular/phoenix

A.2 Best chromosome

The engine played its best chess when this chromosome was used. As we can see, some sections of PVTs seem random. This is due to the fact that even after 1000 generations of evolution, this solution is sub-optimal.

```
King Middle Game
+------+------+------+------+------+------+------+------+
|-28.2 |28    |-42.97|15.99 |0.6   |32.24 |22.1  |20.75 |
+------+------+------+------+------+------+------+------+
|9.55  |-22.77|29.96 |-34.25|44.44 |27.02 |0.51  |-0.07 |
+------+------+------+------+------+------+------+------+
|12.89 |1.13  |-20.39|-2.95 |-36.04|-26.08|24.1  |-51.53|
+------+------+------+------+------+------+------+------+
|-53.77|36.46 |-24.08|41.7  |-2.55 |-37.02|-15.02|-49.85|
+------+------+------+------+------+------+------+------+
|-22.32|-46.16|41.8  |10.11 |9.42  |-44.49|-33.48|-8.16 |
+------+------+------+------+------+------+------+------+
|4.28  |46.76 |16.7  |-5.45 |18.55 |23.03 |-52.05|-27.78|
+------+------+------+------+------+------+------+------+
|-44.83|2.43  |45.54 |-3.53 |-33.55|-20.32|37.32 |-7.05 |
+------+------+------+------+------+------+------+------+
|-24.81|43.53 |11.69 |-35.47|-52.76|-38.76|23.37 |-47.71|
+------+------+------+------+------+------+------+------+
```

```
Queen Middle Game
+------+------+------+------+------+------+------+------+
|-16.24|-6.16 |-27.28|-54.72|43.64 |5.76  |-0.07 |27.6  |
+------+------+------+------+------+------+------+------+
|-3.09 |13.7  |-49.9 |46.54 |-14.06|-18.36|-47.26|1.67  |
+------+------+------+------+------+------+------+------+
|6.36  |-55   |-20.96|-46.2 |-48.16|-13.59|11.94 |-19.61|
+------+------+------+------+------+------+------+------+
|38.87 |-23.88|32.1  |6.59  |-40.92|0.01  |-19.43|6.18  |
+------+------+------+------+------+------+------+------+
|6.86  |-2.66 |2.92  |8.89  |15.32 |37.65 |35.71 |32.49 |
+------+------+------+------+------+------+------+------+
|14.16 |-2.53 |-32.66|2.67  |-17.92|-14.63|30.11 |4.53  |
+------+------+------+------+------+------+------+------+
|6.19  |10.16 |-14.18|-53.43|-16.49|-45.52|36.32 |-36.69|
+------+------+------+------+------+------+------+------+
|14.91 |-27.39|-34.17|36.72 |-37.19|31.65 |45.72 |20.18 |
+------+------+------+------+------+------+------+------+
Rook Middle Game
+------+------+------+------+------+------+------+------+
|-14.61|-27.71|10.64 |-49.54|-15.5 |-25.85|-37.52|33.41 |
+------+------+------+------+------+------+------+------+
|-3.41 |1.37  |24.15 |-10.31|-0.22 |-0.86 |46.51 |28.84 |
+------+------+------+------+------+------+------+------+
|-37.04|21.32 |21.67 |24.5  |-43.6 |-14.62|27.81 |-34.46|
+------+------+------+------+------+------+------+------+
|30.26 |-14.68|-50.93|-35.61|13.52 |-44.91|39.16 |-41.83|
+------+------+------+------+------+------+------+------+
|-46.01|-54.1 |19.37 |42.1  |-19.48|-11.28|46.09 |-18.78|
+------+------+------+------+------+------+------+------+
|-47.61|-21.15|1.55  |-55.2 |2.5   |6.06  |-7.82 |45.58 |
+------+------+------+------+------+------+------+------+
|-12.97|6     |-32.31|47.24 |10.54 |25.13 |12.62 |-20.47|
+------+------+------+------+------+------+------+------+
|-26.08|-5.07 |12    |-15.52|31.81 |-49.42|-38.06|0.07  |
+------+------+------+------+------+------+------+------+
```

```
Bishop Middle Game
+------+------+------+------+------+------+------+------+
|21.02 |7.4   |4.95  |-21.18|-0.03 |8.84  |23.67 |-41.17|
+------+------+------+------+------+------+------+------+
|7.19  |-33.86|-1.2  |1.34  |-4.94 |20.53 |3.78  |-23.52|
+------+------+------+------+------+------+------+------+
|16.19 |-18.34|-11.5 |-47.06|-43.57|16.14 |21.53 |-42.51|
+------+------+------+------+------+------+------+------+
|1.39  |-20.63|22.33 |29.31 |3.12  |44.9  |-0.11 |-50.07|
+------+------+------+------+------+------+------+------+
|12.06 |-27.59|-55.22|-52.81|17.98 |-0.18 |5.87  |-28.82|
+------+------+------+------+------+------+------+------+
|25.77 |46.57 |-45.34|-5.66 |-16.69|-8.37 |16.03 |12.33 |
+------+------+------+------+------+------+------+------+
|37.86 |-32.87|2.8   |-4.87 |-12.57|-26.45|-26.08|-30.67|
+------+------+------+------+------+------+------+------+
|-14.27|2.46  |-40   |10.97 |-30.77|-37.62|38.02 |-9.86 |
+------+------+------+------+------+------+------+------+
Knight Middle Game
+------+------+------+------+------+------+------+------+
|-12.18|15.54 |-13.02|-14.67|-26.16|-26.48|-43.1 |-5.15 |
+------+------+------+------+------+------+------+------+
|13.54 |34.72 |17.29 |35.87 |19.97 |-21.76|1.45  |32.81 |
+------+------+------+------+------+------+------+------+
|-8.43 |43.17 |-3.45 |40.5  |35.21 |-2.5  |-38.9 |-4.38 |
+------+------+------+------+------+------+------+------+
|-47.18|-37.87|41.7  |-18.75|37.08 |-45.62|1.61  |30.96 |
+------+------+------+------+------+------+------+------+
|14.54 |-15.71|27.86 |17.25 |-49.59|21.46 |18.75 |-49.56|
+------+------+------+------+------+------+------+------+
|37.15 |11.41 |-6.16 |39.3  |-52.29|40.83 |-40.21|-22.57|
+------+------+------+------+------+------+------+------+
|-41.94|7.35  |24.69 |11.78 |-20.57|20.16 |-31.19|33.68 |
+------+------+------+------+------+------+------+------+
|31.32 |-6.16 |-45.69|-20.07|-55.76|-31.78|-11.14|-36.91|
+------+------+------+------+------+------+------+------+
```

```
Pawn Middle Game
+------+------+------+------+------+------+------+------+
|-2.01 |47.75 |-52.88|-23.93|-46.65|-45.4 |-38.67|47.75 |
+------+------+------+------+------+------+------+------+
|-23.95|42.84 |-33.08|19.31 |13.52 |-40.98|-12.47|-38.25|
+------+------+------+------+------+------+------+------+
|-39.31|-28.74|17.2  |-31.16|34.93 |-3.24 |-48.22|39.48 |
+------+------+------+------+------+------+------+------+
|16.82 |-32.97|45.19 |35.3  |-10.77|-55.5 |18.96 |-23.1 |
+------+------+------+------+------+------+------+------+
|-39.24|30.09 |-51.35|11.46 |-32.4 |-53.46|-16.04|-51.72|
+------+------+------+------+------+------+------+------+
|-4.44 |-3.59 |30.58 |-3.92 |5.88  |22.9  |-37.25|19.74 |
+------+------+------+------+------+------+------+------+
|11.4  |24.71 |-7.19 |22.9  |-29.82|5.26  |-40.65|39.15 |
+------+------+------+------+------+------+------+------+
|5.45  |6.02  |-7.04 |-54.35|-40.03|-23.62|-36.65|9.94  |
+------+------+------+------+------+------+------+------+
King End Game
+------+------+------+------+------+------+------+------+
|-22.27|-42.92|0.21  |-19.89|10.24 |-32.84|14.95 |-47.42|
+------+------+------+------+------+------+------+------+
|-38.52|-11.77|-22.32|-55   |12.64 |46.12 |-33.39|-33.9 |
+------+------+------+------+------+------+------+------+
|27.08 |27.57 |-25.44|10.48 |-45.48|-55.59|9.4   |0.12  |
+------+------+------+------+------+------+------+------+
|-0.73 |3.14  |-28.52|-15.73|-0.16 |-38.25|2.53  |46.08 |
+------+------+------+------+------+------+------+------+
|1.98  |25.17 |-52.26|-7.74 |23.43 |-34.01|24.87 |39.04 |
+------+------+------+------+------+------+------+------+
|-47.96|40.63 |42.71 |26.91 |-2.87 |-30.3 |16.86 |-55.36|
+------+------+------+------+------+------+------+------+
|12.71 |-32.08|-55.31|-21.75|9.93  |46.75 |-2.24 |-12.35|
+------+------+------+------+------+------+------+------+
|-51.55|7.78  |-45.17|0.66  |-23.37|-31.22|36.22 |12.4  |
+------+------+------+------+------+------+------+------+
```

```
Bishop End Game
+------+------+------+------+------+------+------+------+
|28.57 |-18.21|19.15 |29.34 |-5.32 |28.67 |10.8  |46.93 |
+------+------+------+------+------+------+------+------+
|-41.49|-48.02|-35.88|-6.97 |-24.04|-26.01|39.91 |19.08 |
+------+------+------+------+------+------+------+------+
|31.5  |-53.86|37.68 |-8.25 |-13.95|-39.24|44.87 |-15.26|
+------+------+------+------+------+------+------+------+
|-14.17|-48.65|-4.39 |-47.81|-38.29|2.61  |36.52 |-4.24 |
+------+------+------+------+------+------+------+------+
|-17.54|-4.76 |-42.18|-44.14|-35.08|15.44 |17.31 |28.91 |
+------+------+------+------+------+------+------+------+
|-27.36|46.06 |44.69 |14.17 |9.3   |-22.7 |-49.15|40.76 |
+------+------+------+------+------+------+------+------+
|-2.31 |-25.53|22.61 |24.79 |43.95 |6.18  |-15.27|18.21 |
+------+------+------+------+------+------+------+------+
|26.75 |37.53 |-39.27|-1.8  |43.07 |13.07 |-52.47|-20.22|
+------+------+------+------+------+------+------+------+
Knight End Game
+------+------+------+-----+------+------+------+------+
|-5.4  |-51.55|-44.06|27.77|14.28 |13.49 |-49.56|9.97  |
+------+------+------+-----+------+------+------+------+
|-30.96|-13.99|-22.69|12.9 |-35.5 |41.97 |-11.34|35.57 |
+------+------+------+-----+------+------+------+------+
|-23.78|-39.82|-34.82|28.96|42.63 |27.97 |41.65 |-54.44|
+------+------+------+-----+------+------+------+------+
|-46.57|-16.38|-40.85|22.33|-14.97|-9.02 |-27.41|-8.29 |
+------+------+------+-----+------+------+------+------+
|-23.65|-36.06|-21.92|45.19|-23.52|-40.11|44.59 |-45.34|
+------+------+------+-----+------+------+------+------+
|-53.25|-54.62|19.06 |-15.4|-17.62|16.09 |16.72 |-16.8 |
+------+------+------+-----+------+------+------+------+
|-30.09|-48.93|28.05 |22.16|-24.07|26.17 |25.73 |4.91  |
+------+------+------+-----+------+------+------+------+
|30.11 |-39.72|-8.49 |-2.19|1.93  |6.11  |10.32 |-55.41|
+------+------+------+-----+------+------+------+------+
```

```
Pawn End Game
+------+------+------+------+------+------+------+------+
|4.44  |-52.19|-55.56|28.78 |-24.75|12.14 |-10.38|-51.68|
+------+------+------+------+------+------+------+------+
|-32.39|-18.2 |-16.96|10.09 |-26.2 |45.92 |26.13 |-11.47|
+------+------+------+------+------+------+------+------+
|-11.84|34.1  |12.66 |-40.65|-24.52|-50.55|4.67  |-39.24|
+------+------+------+------+------+------+------+------+
|12.99 |21.59 |38.06 |-38.51|21.62 |40.18 |3.65  |40.06 |
+------+------+------+------+------+------+------+------+
|35.42 |12.04 |10.79 |-4.92 |22.9  |-38.06|22.81 |-39.55|
+------+------+------+------+------+------+------+------+
|31.87 |21.26 |23.78 |36.86 |-43.08|-42.17|44.21 |13.3  |
+------+------+------+------+------+------+------+------+
|-5.41 |35.46 |16.66 |42.66 |-3.18 |23.54 |-43.08|21.03 |
+------+------+------+------+------+------+------+------+
|-26.79|-51.12|20.81 |35.1  |34.97 |27.03 |7.2   |39.27 |
+------+------+------+------+------+------+------+------+
```