A Self Learning Chess Engine

Masters' Thesis

18-May-14

Rahul A R

Contents

List of Figures

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 face that a game of chess cannot be won by pure calculation. There is more to a goodness of a chess position than a numerical value which apparently can be seen only by our brains. Here an effort has been made to improve the current chess engines by letting itself 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.

Introduction

Invention

The precursors of chess originated in India during the Gupta Empire. There, its early form in the 6th century was known as chaturaṅga, 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 (1).

Though we do not know the exact moment in history when chess was invented, there are a couple of stories which are intriguing. One story goes on to say that chess was invented by the demon king Ravana during the events 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 (2).

Another story talks about an ancient Indian Brahmin names 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 stipulates 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 and later realizes his mistake and gives his entire kingdom to Sissa as his gift. (3)

Chess Notation

Chess notation is the term for several systems that have 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-language literature until the late 20th century, but is now obsolete. There are some special systems for international correspondence chess. (4)

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.

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 document uses English language single character abbreviations for chess pieces, although this is easily changed in the source. English is chosen over other languages because it appears to be the most widely recognized.

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.

Piece Identification

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

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".

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)

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: (6)

- 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, the reasoning modes are also used in many other problem-solving domains.
- 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.

Purpose of this 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 say offset by a single square, might differ largely in their goodness.
- The value of a position might vary from player to player which depend on the goals he sets for himself. That is, a position might be losing for a defensive player and might be drawable for an offensive 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.

Computer Chess

History of Computer Chess

The time period of 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” (7). 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 (8) (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 very powerful for the 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 of the game (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 (9).

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 pruning away 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 (10), whose only purpose was to play chess. That 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 (9).

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 appeared to be the prototype of the immortal Deep Blue, which won against the then world chess champion Garry Kasparov in 1997 (11).

Chess Engine Internals

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. (12)

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

Mailbox

The mailbox representation was one the original ideas sketched out by Shannon in (7). 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 reducing the memory requirements of a program, sometimes at the cost of increasing its calculation time.

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.

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 (13).

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 3x3 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, 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.

The number of valuable moves varies for different positions, but on average there are not more than 4-5 such moves. It 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, was described already by Shannon in (7). 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 (13).

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 the 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. 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.

Here, 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 be again compared among 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.

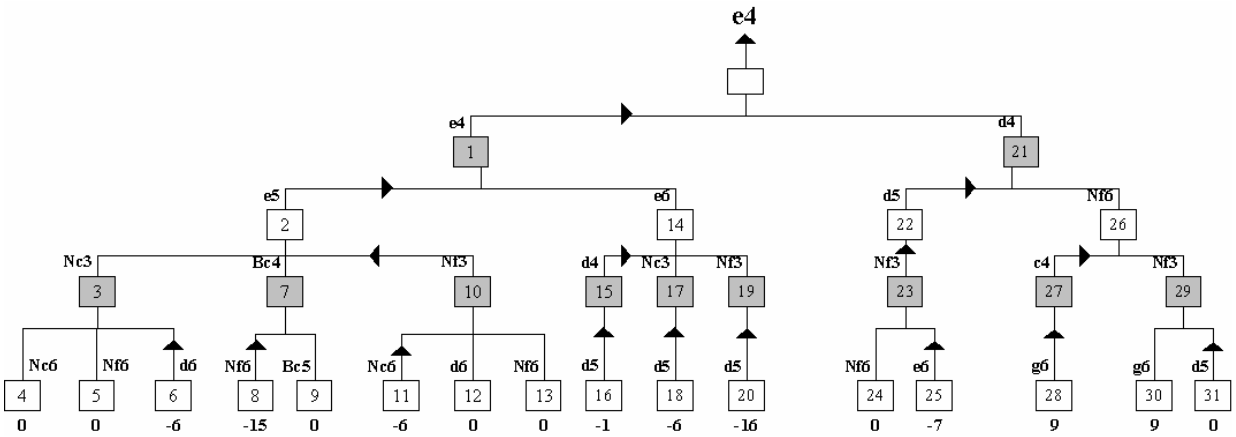

Figure 1 : Full Game Tree

```
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 : Minimax Algorithm

Alpha-Beta Pruning

Let's again consider Fig. 1 and the depth-first method of observing the tree, which is usually used in modern computer chess programs (14). Assume that we have already searched the tree till leaf 8 which has an evaluation score of -15. If We also know 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. And 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 (14), was first presented in (15). Later, in (16), 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 3 : α-β 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 (16) is given in Fig. 3

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[1] and B[2] 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 further calculations from this position will be equivalent of searching to a depth of 2 plies and not 4.

Moreover, every chess position is unique. Two positions with the same pieces on the same squares but with different *en passant* status or castling possibilities are different. So, positions that have been already searched should be stored in some database containing information like the side to move, evaluation score, ply depth, etc. This database is looked through each time new position is to be examined. This database is called 'Transposition Table' and it is usually implemented as a hash dictionary (17).

[1] A: 1. e4 e5
[2] B: 1. e4 e5 2. Nf3 Nc6

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 (18). Zobrist suggested assigning a 32 or 64-bit random number to each piece located on each square; i.e. 12 × 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, XOR operations are performed between the current hash key and the random number assigned for the piece on the square in 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 the 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 be somehow mapped onto the table indices. One method proposed in [30] simply obtains the index as a remainder of the current hash key 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. And 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. Obviously, the more entries a transposition table has, higher the probability of a position to be found in it.

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 speaking, it is impossible to know in advance, which move proves to be the best in every scenario, for otherwise there would be no need to search at all. Therefore, all we can do is try to use prior results and combine them with information about the current situation on the board in order to make up the 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 here, such as when the reply makes a check with a *fork* 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, because it changes the material balance on the board.

Later, 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. 4). Therefore, the sub-tree rooted at the queen capture can be cut-off and this move can be added to the top of the move list to be evaluated.

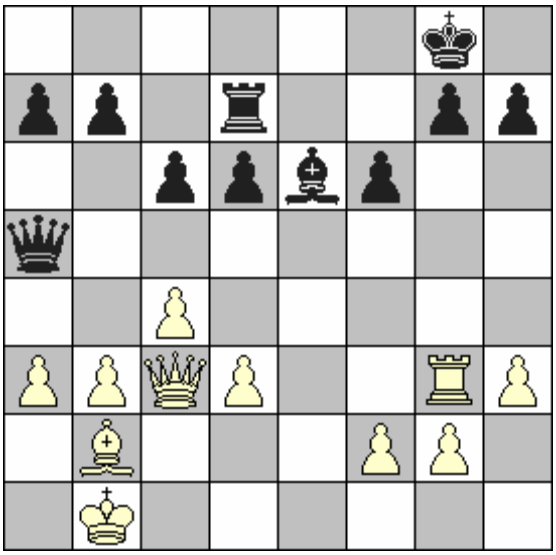

Figure 4 : Valuable and Useless Moves

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 (17) (19)

History Table

As we just discussed, 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, not just the killer moves (17). The advantage here is that, it is possible to accumulate information about the effectiveness of each move throughout the whole game tree, unlike killer moves 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 is, 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 so now. Thus, in Fig. 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 so the history table must be cleaned periodically.

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 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 not more than a few seconds). Due to these sorts 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 bread-first manner rather than the usual depth-first technique. In case the timer expires, it returns the best move belonging to a level which has been searched completely. As stated in (17), 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 which 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.

Quiescence Search

Let us now consider one more shortcoming through the following example. Assume that the search depth is 5 plies and at the 5^{th} ply there is a move with 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 will be estimated as a favorable one.

But on the 6^{th} ply, there may be a move in 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 (20). 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 in anyway? For example, instead of the rook capture there may be a bishop capture in the best case scenario. This sort of a situation is for sure unfavorable.

Since every position during game play is not ready for the evaluation, only relatively *quiescent* positions (7) where the least possible action takes place should be evaluated (such positions are also called dead in (21)). This is why all capture moves and pawn promotions are usually considered separately and searched to till a depth where the result of all material changes finally appears. 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 (17). 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 (13) a value of 2 is said to be the mostly used one.

Null Move

Null move (22) (23) (24), 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 (17).

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 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[3] (see Fig. 5), in which a null move is the only way to avoid loss. But, applying null moves to such positions will lead to mistakes. In position A of Fig. 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, and 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.

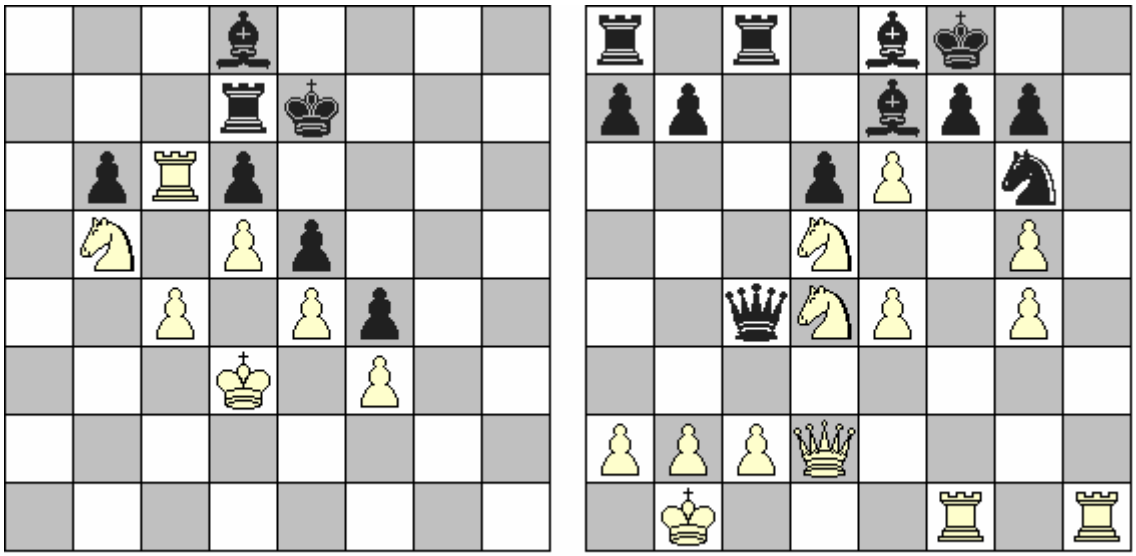

Figure 5 : Zugzwang

Zugzwang positions are almost always losing. Therefore the loss of performance generally does not affect the final result. On the other hand, according to (17), Zugzwang happens extremely rarely in chess with the notable exception of late endgames. Therefore, application of null moves is usually stopped

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

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.

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 the 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 (17), 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 only disadvantage being its huge size. Nalimov's tablebase is actually an improvement of Edwards' original with advanced indexing schemes. More information can be found in (17) where there is an entire chapter devoted to endgame databases used by the famous chess program Dark Thought.

Evaluation Routine

Parameter Selection

Material Balance

Position

Genetic Algorithms

Overview

Terminology

Population

Genetic Operators

Evaluation and Selection

Implementation & Result

Conclusion

Bibliography

1. History of Chess. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/History_of_chess.

2. **Anand, Vishwanathan.** The Indian Defense. *Chess.com.* [Online] http://www.chess.com/article/view/where-was-chess-invented.

3. Wheat and chessboard problem. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Wheat_and_chessboard_problem#Origin_and_story.

4. Chess Notation. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Chess_notation.

5. Chess Club. *Official PGN Spec.* [Online] http://www6.chessclub.com/help/PGN-spec.

6. **Levinson, Robert.** *The Role of Chess in Artificial Intelligence Research.*

7. **Shannon, C.E.** *Programming a Computer for playing Chess.* s.l. : Philosophical Magazine, 1950. 41(4).

8. *Digital Computers Applied to Games.* **Turing, A.M, Bates, C and Bowden, B.V.** 1953.

9. **Friedel, F.** A Short History of Computer Chess. [Online] http://www.chessbase.com/columns/column.asp?pid=102.

10. *Belle Chess Hardware.* **Condon, J.H and Thompson, K.** 3, s.l. : Pergamon Press, 1982, Advances in Computer Chess, pp. 45-54.

11. *Behind Deep Blue: Building the Computer That Defeated the World Chess Champion.* **Hsu, F.** Princeton : Princeton University Press, 2002.

12. Board Representation. *Chess Programming Wiki.* [Online] https://chessprogramming.wikispaces.com/Board+Representation.

13. *Chess Skill in Man and Machine.* **Frey, P.W.** New York : Springer-Verlag, 1977.

14. **Linhares, A.** Data Mining of Chess Chunks: A Novel Distance-based Structure. *Data Mining.* 2003. Vol. 4.

15. *Chess Playing Programs and the Problem of Complexity.* **Newell, A, Shaw, J.C and Simon, H.A.** 1958, IBM Journal of Research and Development, pp. 320-335.

16. *An Analysis of Alpha-beta Pruning.* **Knuth, D.E and Moore, R.W.** 4, 1975, Artificial Intelligence, Vol. 6, pp. 293-326.

17. *Scalable Search in Computer Chess.* **Heinz, E.A.** s.l. : Vieweg Verlag, 2000.

18. *A New Hashing Method with Application for Game Playing.* **Zobrist, A.L.** 2, 1990, ICCA Journal, Vol. 13, pp. 69-73.

19. *Computer Chess and Search.* **Marsland, T.A.** Edmonton : Computer Science Department, University of Alberta, 1991.

20. **Berliner, H.J.** *Chess as Problem Solving: The Development of a Tactics Analyzer.* Carnegie-Mellon University. Pittsburgh : s.n., 1974. Ph.D. thesis.

21. **Turing, A.M, et al.** *Digital Computers Applied to Games.* s.l. : Pitman, 1953. pp. 286-310.

22. **Beal, D.F.** Experiments with the Null Move. *Advances in Computer Chess.* 5, 1989, pp. 65-79.

23. *Null Move and Deep Search: Selective Search Heuristics for Obtuse Chess Programs.* **Donninger, C.** 3, s.l. : ICCA Journal, 1993, Vol. 16, pp. 137-143.

24. *Experiments with the Null-Move Heuristic.* **Goestch, G and Campbell, M.S.** New York : Springer-Verlag, 1990, Computers, Chess, and Cognition, pp. 159-168.