Kantonsschule Hohe Promenade, Gymnasium, Zürich
Schuljahr 2019/2020

Programming a Chess AI

Oliver Graf, Klasse 6e

Betreuende Lehrperson: Clemens Pohle
Korreferent: Stefan Müller

Contents

1 Introduction

When *Deep Blue* defeated the acting chess world champion Garry Kasparov in 1997, chess AIs became central to the chess world. Unbeatable for human players, they strongly influence the playing style of chess masters today and they are still being developed: every year, the best chess engine is determined at the TCEC (Top Chess Engine Championship). Last year, an interesting development occurred: for the first time, a chess engine based on neural network technology — *Leela Chess Zero* — won the TCEC while *Stockfish*, the winner of the last few years, only ranked second (Monokroussos, 2019).

A few months before, Google Deepmind's *AlphaZero* — also a neural network AI — had already defeated *Stockfish*, though in arguably unfair conditions (DeepMind and University College London, 2018). Since then, I've always asked myself how chess engines work and how hard it would be to code one myself.

There are two approaches to coding chess AIs: using neural networks like *AlphaZero* or with algorithms that already existed in the fifties — like *Stockfish* or *Komodo*. These algorithms make use of search trees, α–β pruning and heuristic evaluation algorithms. I decided to code a chess AI using the *Stockfish*-like approach because coding a neural network certainly would have blown the frame of a matura thesis.

As an hommage to *Stockfish*, I baptized my own chess engine *Mockfish*.

1.1 My question

Am I capable of programming a chess AI that can defeat me as a human player?

1.2 Prerequisites for understanding this report

To understand every part of this report, you only have to know the chess rules (including en passant captures, pawn promotion and castling) as well as a fundamental understanding of programming. This includes arrays, functions and object-oriented programming.

2 The work process

2.1 May and June 2019: *Mockfish* 0.1

I started using the name *Mockfish* starting with the second version of my chess engine, so this version is called 0.1 instead of 1.0. It included an advanced GUI and though it was possible to play a real game against it, it still was very inefficient: since I hadn't separated the frontend (graphics) from the backend (position evaluation), this version has severe performance issues that I couldn't get rid of without totally rewriting the application.

2.2 July and August 2019: *Mockfish* 1.0

The second version didn't use more advanced chess programming theory than version 0.1, but its code was more orderly and the back- and frontend, which was now greatly simplified, were separated. While the performance benefit wasn't great, at least it ran more stably and produced slightly better results than *Mockfish* 0.1.

2.3 September 2019: *Mockfish* 2.0

The first real performance improvement came with *Mockfish* 2.0: it used a modern approach to board representation and it made use of concepts like bitwise operators that enhanced performance. However, it still experienced the problem that the two key elements — tree expansion (section 3.3) and the α–β algorithm (section 3.6) — were separated, something that reduced performance somewhat.

2.4 After September 2019: *Mockfish* 2.1

What reduced computation time to 1% of *Mockfish* 2.0 were three new core ideas: sorting the search tree (section 3.7.1), dividing up the evaluation into several threads (section 3.7.2) and directly connecting the tree expansion and the α–β algorithm. While *Mockfish* 2.1 still cannot compete with professional chess engines, it is capable of defeating inexperienced chess players with relative ease.

3 How does a chess AI work?

There are different approaches to chess AIs. As mentioned on page 1, there are neural networks and algorithms. (In the future, I'll call the first "AI" and the latter "engine" to distinguish between them.) I'll only go into chess engines because neural networks function in a completely different way.

For a chess engine, several distinct parts have to work together perfectly so that you don't end up with annoying bugs. I'll discuss the most important ones.

3.1 Board representation

This is the heart of every chess application. Having an efficient board representation is vital for good performance, and there are dozens of different board representation types to choose from (piece lists, piece sets, 0x88 and bitboards just to name a few). I'll only analyze the types that I used in the different versions of *Mockfish*.

To compare the different board representation types, I'll use an empty board except for a white knight on h2 in the following sections.

3.1.1 *Mockfish* 0.1

The most intuitive idea is to use a two-dimensional array with 8 elements in both dimensions — like this:

```
// 8*8 array
char *board[8] = {
     {'-','-','-','-','-','-','-','-'},
     {'-','-','-','-','-','-','-','-'},
     {'-','-','-','-','-','-','-','-'},
     {'-','-','-','-','-','-','-','-'},
     {'-','-','-','-','-','-','-','-'},
     {'-','-','-','-','-','-','-','-'},
     {'-','-','-','-','-','-','-','N'},
     {'-','-','-','-','-','-','-','-'}
};
```

This does seem like a good idea. However, it is very inefficient for two reasons:

- Checking whether a coordinate exists on the board is not very efficient because both two coordinate elements have to be checked. Only if the x and the y coordinate are both ≥ 0 and <8 does a square exist on the board.
- Accessing a single element is also slower than with a one-dimensional array since the computer internally has to make a multiplication — something that wouldn't be necessary with a one-dimensional array. The computer has to do that because a two-dimensional array isn't actually stored in two dimensions but rather in one. This is done because the processor cannot handle two-dimensional objects. The only purpose of multidimensional arrays is making it easier for the human user to read, understand and manipulate matrices. For

accessing our knight on h2, the computer has to do the following operation: `board[6][7]` $\rightarrow$ *(internal)* `board[6*8 + 7]` = `board[55]`. Of course, one single multiplication doesn't measurably slow down the program. However, just to calculate all legal moves, elements of the board array have to be accessed hundreds, if not thousands, of times. And since the calculation of legal moves is done tens of thousands of times again when evaluating a position, the tiny performance issues pile up until they make a great difference.

3.1.2 *Mockfish* 1.0

For the second version, I rewrote the board representation as a one-dimensional array with 64 elements. For the following two elements in the same rank at the positions r_1 and r_2 and the elements with the positions f_1 and f_2 in the same file, the following rules apply:

$$\left\lfloor \frac{r_1}{8} \right\rfloor = \left\lfloor \frac{r_2}{8} \right\rfloor \tag{1}$$

$$f_1 \pmod 8 = f_2 \pmod 8 \tag{2}$$

Such an array looks like this:

```
// 64 array
char board[] = {
    '-','-','-','-','-','-','-','-',
    '-','-','-','-','-','-','-','-',
    '-','-','-','-','-','-','-','-',
    '-','-','-','-','-','-','-','-',
    '-','-','-','-','-','-','-','-',
    '-','-','-','-','-','-','-','-',
    '-','-','-','-','-','-','-','N',
    '-','-','-','-','-','-','-','-'
};
```

Finding out on which rank a certain number is can be done with the following function: `int GetRank(int num) { return num / 8; }`; while finding the file can be done with `int GetFile(int num) { return num % 8; }`. Thus, this type of board representation works the same way as the internal board of the two-dimensional array described in the section 3.1.1 on page 3.

This seems more efficient since it at least uses only one dimension, but it isn't really that much faster because of one problem: testing whether something is on the board or not becomes now even more complicated: since moving one step to the right from our knight (to the imaginary square i2) results in stepping from `board[55]` to `board[56]`, and this doesn't seem to cause a problem at all: `board[56]` stands for square a1. However, this is not what we intended to achieve and thus shouldn't be allowed. That's why before testing, the coordinates have to be split up in X and Y coordinates which makes the entire calculation much more time-consuming than before. For example, to test if the move from our knight's coordinates to the right would be legal can be done in the following way:

```
int position = 55;
int X = GetFile(position), Y = GetRank(position);
if (OnBoard(X + 1, Y))
     ; // it is a legal move
```

3.1.3 *Mockfish* 2.0

This version finally used a more efficient, modern approach to board representation. I decided to use the so-called 12×10 array because it is still quite close to an actual chess board while other approaches to board representation like bitboards are rather abstract. This type of array uses a chess board and a frame of hypothetical squares around it — two ranks on the top and the bottom, respectively, while the left and the right side have one hyptothetical file. The entire board is stored in a one-dimensional array:

```
// 12*10 array
char board[] = {
     '*','*','*','*','*','*','*','*','*','*',
     '*','*','*','*','*','*','*','*','*','*',
     '*','-','-','-','-','-','-','-','-','*',
     '*','-','-','-','-','-','-','-','-','*',
     '*','-','-','-','-','-','-','-','-','*',
     '*','-','-','-','-','-','-','-','-','*',
     '*','-','-','-','-','-','-','-','-','*',
     '*','-','-','-','-','-','-','-','-','*',
     '*','-','-','-','-','-','-','-','N','*',
     '*','-','-','-','-','-','-','-','-','*',
     '*','*','*','*','*','*','*','*','*','*',
     '*','*','*','*','*','*','*','*','*','*'
};
```

It is much easier to test if a square doesn't exist when those hypothetical squares (like i2) actually exist in the board array. Moving one to the right from h2 would result in moving from `board[88]` to `board[89] = '*'`[1] which is out of the board (from now on to be called **OOB**). Hence, to check whether an accessed square exists, not its coordinates but its contents have to be checked: if it is an OOB square, it simply doesn't exist. The almost doubled size of the board array doesn't really matter since storage space isn't a problem nowadays.

The knight is the reason why the left and the right side only have one imaginary file while there are two each on the top and on the bottom. Since from one of the border squares, it can reach squares two ranks or two files away from the real chess board, two extra ranks are necessary. Because the array is also one-dimensional, it is actually an illusion that there is only one file per side: when one move passes the

[1]To move to the right, you have to add 1 to your position (88 in the example) for every square. To move to the left, you have to subtract one. However, to move upwards or downwards, you have to subtract 10 or add 10, respectively.

right border, its target square is just moved to the left hypothetical file, but it does still land on an OOB square. Figure 1 on page 6 explains this in more detail.

Mockfish 2.1 uses the same board representation.

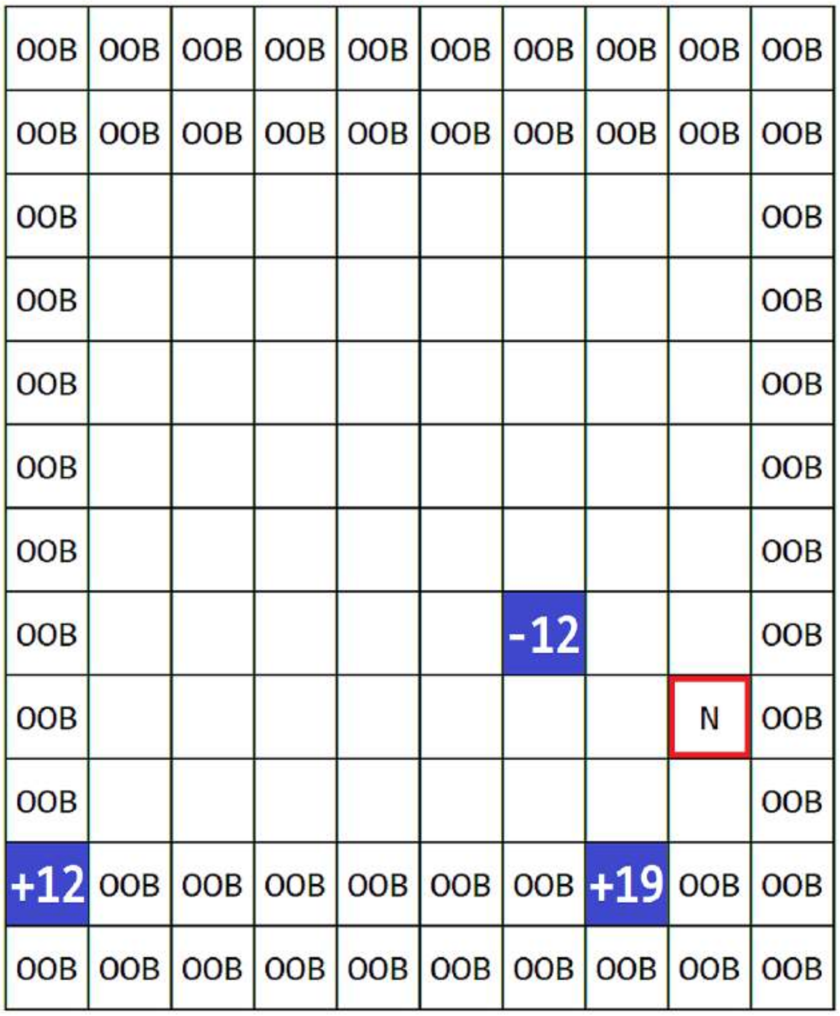


Figure 1: This picture illustrates how the 10×12 array allows to store the entire board in a one-dimensional array: since a knight move is composed of a horizontal component (+1 or -1 per square) and a vertical component (+10 or -10 per square), we can compose the following moves: One square up and two to the left results in -12. This is the square f3. One down and two to the right corresponds to +12. This is a hypothetical square with the OOB flag, so it is an illegal move. Note that this OOB square is in the hypothetical file to the left of the board because it passed the right border of the OOB frame. The third move shown in this figure (two down, one to the left) results in +19. This, too, is an OOB square and hence cannot be accessed.

3.1.4 Square representation

To represent a square, I used a very naive approach in version 0.1: a string with two elements — one to indicate the color of the piece and one to indicate its type. For example, a white knight would be `"wN"`, a black queen `"bQ"` and an empty square `"**"`.

For version 1.0, I reduced this to a single character: capital letters represent the white pieces, a single asterisk empty squares and a small letter the black pieces.

However, there is a more efficient way to tackle square representation and that is splitting a single byte into different bits that have different roles.

Bits 7-6	Bit 5	Bit 4	Bit 3	Bits 2-0
unused	color	en passant flag	castle flag	piece type

This way, more information can be stored without losing performance — castling and en passant rights had to be stored in separate variables in the position object[2] before.

3.2 Move generation

Another vital aspect of chess engines is the function that generates legal moves. It is quite a bulky function that takes up most of the calculation time used in evaluation and if it is buggy, the entire chess program won't correctly behave. Luckily, there are many tools to debug the move generation function quickly — like Perft results: Perft (**PERF**ormance **T**est) is a test that creates all possible positions from a certain start position to a predetermined depth. Then, the number of all positions at the lowest depth is taken and compared with a result table. If the results match, the move generation function created the correct moves. The Perft results also show how large the search tree can get with increasing depth. The following table shows the results for the first 6 moves in the starting position:

Depth	1	2	3	4	5	6
Number of leaf nodes	20	400	8,902	197,281	4,865,609	119,060,324

There are two main ways to generate moves — a function that walks through the position and progressively adds moves as it analyzes all the pieces as well as a more advanced concept using so-called attack and push maps. The latter are quite complicated and thus I'll only go into the first approach which I used for all *Mockfish* versions.

Lookup tables greatly simplify the task of generating moves because the same steps can be re-used for all pieces (except for pawns, because they don't follow the normal rules). The lookup table I used in the latest version looks like this:

```
// move generation lookup table
int *lookup[10] = {
    {},{}, // [0],[1] -> pawn has special rules, so no lookup table
    { false,   -21,-19,-12,-8,8,12,19,21,0 },   // 2 - knight
    { true,    -11,-9,9,11,0 },                 // 3 - bishop
    { true,    -10,-1,1,10,0 },                 // 4 - rook
    { true,    -11,-10,-9,-1,1,9,10,11,0 },     // 5 - queen
```

[2] I always used a class type to represent a position. This class type had to store every necessary element to distinguish it from all other possible positions (such as castling rights, en passant rights, the pieces and the board, whose turn it is and what the status of the game is (checkmate, running, stalemate, agreed draw, etc.).

```
    { false,   -11,-10,9,-1,1,9,10,11,0 }       // 6 - king
};
```

The first element of every subarray is either `true` or `false`. If it's `true` then the piece type this subarray references to moves in a ray (bishop, rook and queen). If it is `false`, then the piece can only take one step in every direction that it can move to (knight and king). The other elements indicate the possible directions in a 12×10 board type. Those numbers are derived from the four basic rules: +1 for a square to the right, -1 to the left; +10 for a square down, -10 upwards. With those, any move vector can be constructed. For example, the fourth knight direction (-8) is one square up and two to the right, resulting in `-10+1+1 = -8`.

The reason for the zero values at the end of each sub-array is the following: when stepping through all the possible directions for the different pieces, there needs to be some way to stop the loop. I did this with a zero value which proved to be quite simple. Walking through a single sub-array of `lookup` looks like the following:

```
for (int i = 1; lookup[/*PIECE*/][i]; i++) {...}
```

There is another element that I first implemented in version 2.1 that speeds up move generation a bit: by separating the move generation for positions that are in check and such that aren't, some parts of the complete move generation don't have to be executed:

- **Ray pieces and testing for check**: If the moving color's king isn't in check, only the first move of a ray[3] piece in every direction has to be tested for a discovered attack on its own king. This is thanks to the fact that a ray piece can only discover a check on the king if the first move in a certain direction does so; and subsequently, all further moves in the same direction cannot result in an attack on its own king if the first one didn't. However, if the moving player is in check, every move in every direction with the ray pieces have to be checked because they might block the checking piece on a square past the first one.

- **Castling and check**: If the moving color's king is checked, then castling isn't an option because castling is illegal for a checked king. Thus, all castling possibilities can be disregarded if the moving color is checked. However, if the moving color isn't in check, of course castling has to be considered because then it is legal.

3.3 Search tree

The search tree is a tree that contains all positions that can be reached from the mother node with all possible legal moves up to a certain depth.

[3]A ray piece is a piece that can move in a straight line until it reaches the borders of the board. The ray pieces of the original chess game are the bishop, the rook and the queen.

The first node of a tree is called **root** node. The nodes that are all linked to one node are called **daughter** nodes while the node from which a specific node stems is called its **mother** node. The set of all nodes without daughter nodes are called **leaves**. The number of different levels in the tree is called **depth**.

The depth of the search tree is vital for the quality of the evaluation and since the speed at which it can be built is largely based on the efficiency of the move generation algorithm, the latter is fundamental for the performance of the chess engine. When the search tree is larger (an increased depth), then the engine also reaches deeper into the position and can better estimate the consequences of every possible move.

The search tree can be represented in different ways in the backend. Version 0.1 used a quite inefficient approach: a two-dimensional queue (`std::deque`) in which every element had to manage the addresses of the different elements in the search tree on its own — so the information basically existed twice for every node. This reduced performance a lot and it was very error prone because the safe access to information in the tree could never be totally guaranteed since it could have been changed by a position type instance or in the tree itself.

In version 1.0, I used a simpler way: a single tree node only ever knew its mother node and its daughter nodes and thus only had to manage those. To store the mother and daughter nodes' addresses in the position object, I used smart pointers (`std::shared_ptr`). This reduced performance but it guaranteed that I wouldn't have to deal with memory handling issues like memory leaks.

After version 2.0, I went back to normal C-style pointers, so that the position type had the following structure:

```
// basic tree node class
class Node {

protected:
      Node *mother;
      vector<Node*> daughters;
};

// position node class
class TreePosition : public Node { ... };
```

3.4 Heuristic Evaluation

Heuristic evaluation is a concept fundamental to evaluating a search tree. A heuristic evaluator guesses the positional value of a chess position. I'll explain the point behind that in the chapter about minimax on page 12.

The heuristic position evaluator (**HPE**) returns an estimated value from the position and thereby considers the overall material situation, the placement of the pieces and how they interact. Typical measurements a HPE takes are looking for doubled pawns and checking if the king is safe.

Generally, the HPE returns a positive value if white is in a better situation, 0 if it's exactly equal and a negative value if black's position is better. If the black side is mated, the evaluation is `INT_MAX`; if white is mated, it's `INT_MIN`.

In version 0.1, I used by far the most complicated evaluator. While of course material and the piece positioning were rated, it also checked whether the rooks were on open files, if there were doubled pawns and if the knight was in the thick of the battle or in a region on the board that was rather deserted. It also compared the square colors of the remaining bishops and adjusted the evaluation accordingly[4].

For the later *Mockfish* versions, I used massively simplified versions in order to cut down on the time actually spent with the HPE and instead extending the search tree which grants much better results even if that reduced the accuracy of the HPE a bit. The new versions just evaluate the material and the piece placement, not all the other elements of a position mentioned above. For that, I used two lookup tables: one for the value of the pieces and one for the positional value which averages `0` and can be higher or lower, depending on how good this specific square is deemed to be for a piece.

To show this, I'll use two subarrays in this lookup table: one for the knight in the opening and one for the pawn in the endgame. Note that in the descriptions of the tables, I always used the white player's perspective because the tables were written from white's perspective. In order to get black's perspective, I used a lookup table to reflect the values.

3.4.1 The pawn in the endgame

```
const int PAWN_ENDGAME[] =
{
     0,   0,   0,   0,   0,   0,   0,   0,
     50,  60,  65,  65,  65,  65,  60,  50,
     25,  45,  50,  50,  50,  50,  45,  35,
     0,   10,  20,  25,  25,  20,  10,  0,
     -10, -5,  5,   10,  10,  5,   -5,  -10,
     -15, -10, -5,  -10, -10, -5,  -10, -15,
     -20, -15, -15, -20, -20, -15, -15, -20,
     0,   0,   0,   0,   0,   0,   0,   0
};
```

The following are the three most important reasons that explain why the numbers in the lookup table are exactly the way they are:

- **Encouraging pawns to capture towards the center:** Whenever reasonable, a square to the front and towards the middle of any other square has a higher value. This encourages rim pawns to capture towards the center which gives one more center control. Also, pawns on the a and h files are a lot

[4]If both teams only have one bishop left in the endgame, two cases can happen: if they are bishops with different square colors, even an advantageous position for one player will often result in a draw, while it can be the difference between victory and defeat if both bishops have the same square colors. (Markushin, 2013).

weaker than the other pawns since the opponent can force a draw more easily against a passed pawn on the rim than against a passed central pawn. This is why a pawn on the b file is more valuable than one on the a file on the left.

- **Encouraging pawns to move forward:** Usually, moving one square forward for a pawn either reduces the evaluation penalty or increases its evaluation bonus. This encourages the AI to select forward moves with a pawn.

- **Evaluating left-behind pawns worse in the middle:** Though the difference is small, it does influence *Mockfish*'s playing style in the endgame. Because the central pawns exert much more control over the board than the flank and rim pawns, advancing them is much more important. Giving a larger penalty to left-behind pawns in the center encourages *Mockfish* to move them forward and thus taking control of the center and bringing its most important pawns closer to promotion. Not only the values in the array show this idea, but also the number of points that can be gained from advancing a pawn on the second rank to the fourth: On the a, b, g and h files, the benefit is 10 points. On the c and f files it's 20 points and on the two central files even 30 points.

3.4.2 The knight in the opening

```
const int KNIGHT_OPENING[] =
{
     -60, -55, -50, -45, -45, -50, -55, -60,
     -55, -50, -45, -30, -30, -45, -50, -55,
     -45, -30, -15, -5,  -5,  -15, -30, -45,
     -25, -10, 10,  10,  10,  10,  -10, -25,
     -15, -5,  15,  5,   5,   15,  -5,  -15,
     -15, 0,   20,  5,   5,   25,  0,   -15,
     -15, -5,  5,   15,  15,  5,   -5,  -15,
     -25, -5,  0,   5,   5,   0,   -5,  -25
};
```

The lookup table for the knight in the opening is equally complicated as the one for the pawn in the endgame, but makes use of completely different ideas:

- **Encouraging knights to only take one single step in the opening:** Note that the squares c3 and f3 have the highest evaluation and from there, it isn't possible to move the knight to a square with a better evaluation. Thus, once the knight stands on either c3 or f3, the engine usually decides against moving it again. There are exceptions to that rule, of course — one example being the Two Knights Defense in the Italian Game where the engine likes the move Nf3-g5 which leads to the Fried-Liver Attack[5].

- **Encouraging knights to stay away from the central four squares:** The squares d4, e4, d5 and e5 are rated worse than the squares around them.

This is because those squares are put to better use by placing pawns there to establish a stable center and instead having the knights on f3 and c3 control said squares which makes them much more effective. Also, while knights on the central squares do control their potential maximum of 8 squares, those squares aren't on the center of the board and thus that theoretical benefit is reduced somewhat.

- **Encouraging knights to develop in the opening:** The penalty for standing on the squares is almost as great as for the secondary development squares a3 and h3. This encourages to move the knight early in the opening — optimally to the squares f3 and c3 which get the best evaluation. This also encourages *Mockfish* to move its knights before the bishops in the opening because they get less of a penalty for standing on the starting square. Note that the squares d2 and e2 gain a bonus of 15 points since these are also quite popular destinations for the knights in the opening, especially in games with a closed pawn structure.

Now, even though the reduced accuracy of the HPE might be a problem in some special positions, this just improves performance a great deal. Also, since I took so many factors into account when creating the lookup tables, there isn't much accuracy lost — except for factors that depend on the interaction of multiple pieces — like leaving a queen unattended attacked by a pawn generally isn't a good idea.

Naturally, the two lookup tables (of 18 in total) don't show all the factors that I considered when creating those lookup tables. However, I decided to show those two in this document because they display lots of recurring elements also found in the other tables and because they show best the beauty of position evaluation in *Mockfish*.

3.5 Minimax

Minimax is the most fundamental algorithm for evaluating a position. It uses the concepts of the search tree and the HPE and combines them to calculate the value of a position.

The code for the minimax function could look like the following:

```
// Minimax algorithm
int Minimax(int depth, Position &pos) {

    if (depth == 0)
        return pos.HPE();

    pos.CreateDaughters();

    if (pos.turn == WHITE) {
        pos.eval = INT_MIN;
```

[5]The Fried-Liver attack is the exact combination of the following moves: 1. e4 e5 2. Nf3 Nc6 3. Bc4 (the Italian Game) Nf6 (the Two Knights Defense) 4. Ng5 d5 5. exd5 Nxd5 6. Nxf7.

```
        for (auto u : pos.daughters)
            pos.eval = max(pos.eval, Minimax(depth - 1, *u));
    }
    else {
        pos.eval = INT_MAX;
        for (auto u : pos.daughters)
            pos.eval = min(pos.eval, Minimax(depth - 1, *u));
    }

    return pos.eval;
}
```

The minimax code is a recursive function that steps through the tree. When it reaches the bottom of the tree, it returns the heuristic evaluation of the leaf node. Then, the algorithm maximizes (for the white player) or minimizes (for the black player) the value returned by the function applied to all the position's daughter nodes in order to determine which position is reached when both players play the best moves. The heuristic evaluation of this node then determines the returned value of the node.

As a simple example: the white player has three nodes to choose from which will return the following values: -4, 2 or 1. He'll choose the node with evaluation 2 because it is the best result from the white player's perspective. Had it been black's turn, the black player would have played the move leading to the position with evaluation -4.

Of course, the greater the depth used, the more accurate the evaluation is. Were the depth 17,691, then the result would be absolute: as the maximal length of a chess game is 17,691 moves, there are no chess games after that and thus the entire chess game could have been solved. (Labelle, 2015). As there are, however, more different chess games than atoms in the universe, there is no definite solution to chess that can be stored in any way (Shannon, 1950). Thus, we'll have to be satisfied with approximations using a depth of about 10. However, by discarding nodes that aren't deemed important, chess engines like *Stockfish* can reach depths of up to 50. In fact, the reason why there are no known Perft results for a depth greater than 15 is because it would take days, weeks or even months to calculate them.

3.6 α–β pruning

Minimax isn't the most efficient algorithm to evaluate a tree: there are many nodes in the tree that don't have any impact at all on the position's evaluation — in fact, all of them except for the nodes that lead to the "minimaxed" leaf node. However, it's impossible to directly determine the correct leaf node without any other comparison.

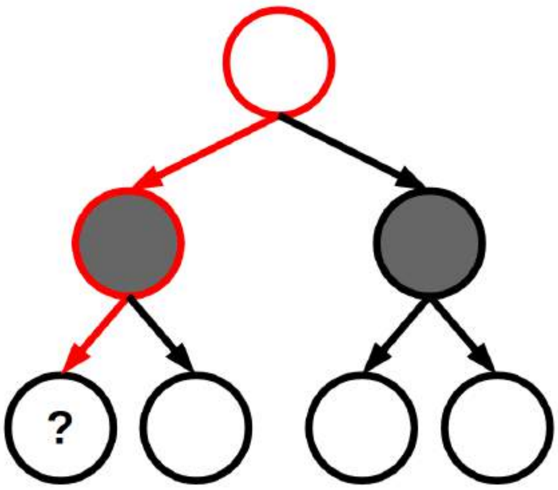
The α–β algorithm, however, finds a way to efficiently cut (prune) away nodes that don't have an impact on the evaluation without changing the results. This has the potential of massively cutting down on calculation time. The big change when comparing the α–β algorithm with minimax is the addition of two new variables that are used within the function: `alpha` is the worst possible result for white while `beta` is the worst possible case for black. This means that white wants to increase the value of `alpha` while it's black's goal to decrease `beta`. The main difference between naive Minimax and α–β is that if the cutoff inequality `alpha >= beta` is true, then all the following nodes in the same branch can be ignored. I'll explain this in more detail with figure 6 on page 17.

```cpp
// Alphabeta algorithm
int Alphabeta(int depth, Position &pos, int alpha, int beta) {

      if (depth == 0)
            return pos.HPE();

      pos.CreateDaughters();

      if (pos.turn == WHITE) {
            pos.eval = INT_MIN;
            for (auto u : pos.daughters) {
                  pos.eval = max(pos.eval, Alphabeta(depth - 1, *u, alpha,
                      beta));
                  alpha = max(pos.eval, alpha);
                  if (alpha >= beta)
                        break;
            }
      }
      else {
            pos.eval = INT_MAX;
            for (auto u : pos.daughters) {
                  pos.eval = min(pos.eval, Alphabeta(depth - 1, *u, alpha,
                      beta));
                  beta = min(pos.eval, beta);
                  if (beta <= alpha)
                        break;
            }
      }

      return pos.eval;
}
```

The initial call of the α–β function would then be:

```cpp
result = Alphabeta(depth, root node, INT_MIN, INT_MAX);
```

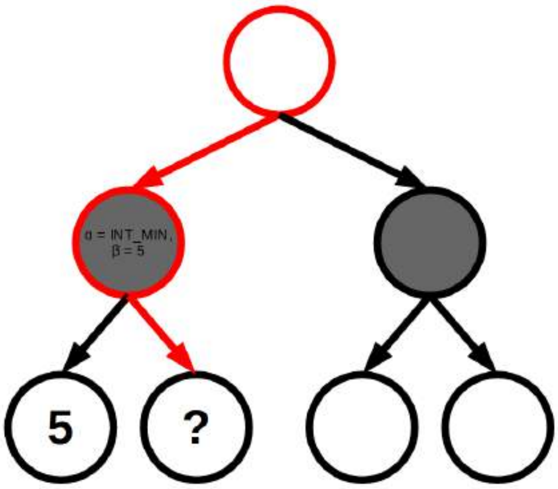
The functionality of the α–β function is best explained by some images. Note that in the following search tree, the leaf nodes is the group of the four white nodes at the bottom of the tree, the black nodes are generally referred to as the (root's) daughter nodes while the topmost white node is the root.



```
// In Alphabeta(...) of the first
    daughter node
for (auto u : daughters) {
      // Alphabeta(...) call of the
          first leaf node
      pos.eval = min(pos.eval(MAX),
          Alphabeta(0, *u,
          alpha(MIN), beta(MAX)));
      {...}
}
```

Figure 2

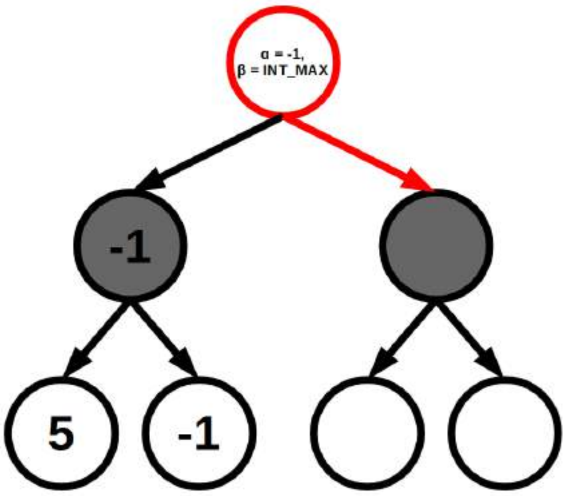
Figure 2 shows a search tree with depth 2 and two daughter nodes per node. Note that even in this example with a small number of daughter nodes per node, the majority (4/7) of the tree's nodes are leaf nodes. The code to the right of figure 2 shows what is happening in the source code at the point in time that the figure is showing.



```
// In Alphabeta(...) of the first
    daughter node
for (auto u : daughters) {
      // After receiving the
          evaluation of the first
          leaf node
      beta = min(pos.eval(5),
          beta(MAX));
--------------------------------------
      // Alphabeta(...) call of the
          second leaf node
      pos.eval = min(pos.eval(5),
          Alphabeta(0, *u,
          alpha(MIN), beta(5)));
      {...}
}
```

Figure 3

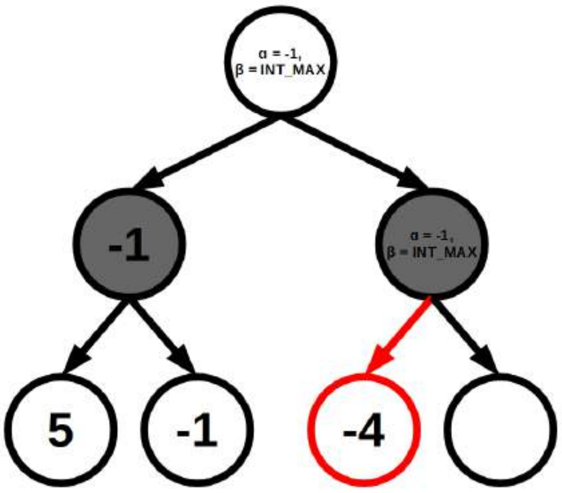
Figure 3 shows how `beta` is set to 5 in the first daughter node of the root. Now, with the newly gained information, the α–β algorithm is used on the search tree's second tree node. Up to now, there hasn't been any difference between α–β and minimax except for how the different evaluations are stored.



```
// In Alphabeta(...) of the root
    node
for (auto u : daughters) {
      // After receiving the
          evaluation of the first
          daughter node
      alpha = max(pos.eval(-1),
          alpha(MIN));
-----------------------------------------
      // Alphabeta(...) call of the
          second daughter node
      pos.eval = max(pos.eval(-1),
          Alphabeta(1, *u, alpha(-1),
          beta(MAX)));
      {...}
}
```

Figure 4

Now, after the second leaf node has returned -1, the first daughter node of the root node is set to -1 as the minimizing player (black) prefers the evaluation -1 over 5. Then, -1 is returned to the mother node and `alpha` is set to -1 as described in the source code next to Figure 4. Now, as the cutoff comparison `alpha >= beta` still isn't true, the algorithm continues down to the second daughter node.



```
// In Alphabeta(...) of the second
    daughter node
for (auto u : daughters)
      // Alphabeta(...) call of the
          third leaf node
      pos.eval = min(pos.eval(MAX),
          Alphabeta(0, *u, alpha(-1),
          beta(MAX)));
      /* now, pos.eval = -4 */
}
```

Figure 5

In Figure 5, the α–β algorithm is applied on the third daughter node and it returns -4.

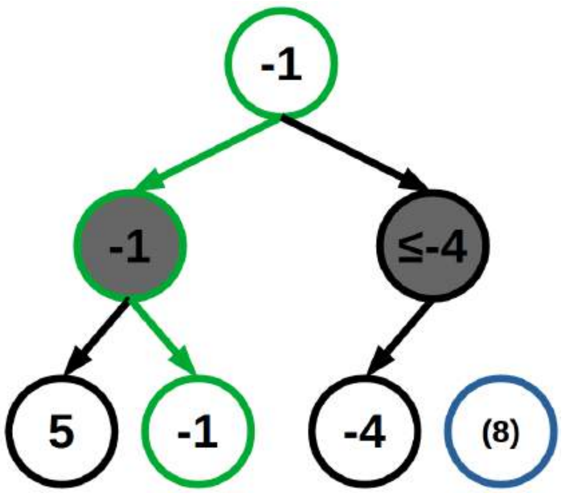


Figure 6

```
// In Alphabeta(...) of the second
    daughter node
for (auto u : daughters) {
      // After receiving the
          evaluation of the third
          leaf node
      beta = min(pos.eval(-4),
          beta(MAX));
      if (beta(-4) <= alpha(-1))
            break; // alpha-beta
                cutoff
}

return pos.eval(-4);
```

Since now, finally, `alpha >= beta` is true, the `for` loop through the remaining leaf nodes is cut off and the current evaluation value (-4) is returned. When this value is sent up to the mother node, the mother node determines which one (of `-1` and `-4`) is the better value for white (the maximizing player). Since `-1 > -4` is true, the position's evaluation is `-1`.

The reason why this cutoff can be made is because the fourth leaf node doesn't matter. There are two cases to distinguish from for this fourth leaf node but they both give the same result: The moving player of the root node can safely discard the fourth leaf node:

- **`eval > -4`:** Because black is the minimizing player, it's her/his goal to get an evaluation as small as possible. Since `eval` is definitly greater than `-4`, the evaluation keeps its value of `-4`. And thus, because the white player prefers `-1` over `-4`, the fourth node with `eval > -4` doesn't influence the final evaluation.

- **`eval <= -4`:** Now, the evaluation of this daughter node gets an even lower value, so the white player can still ignore this evaluation and instead take the path that leads to `-1`.

Therefore, whatever value `eval` has, it doesn't affect the end result which leads to the green path illustrated in figure 6.

In this example, the α–β algorithm is only faster than regular minimax — which evaluates all leaf nodes — by a small margin. However, in a larger search tree with many more daughter nodes per node (in chess up to 40), the α–β algorithm can be very efficient: if n is the number of leaf nodes in a search tree and $e(n)$ the number of evaluated nodes by the α–β algorithm, then the following equation holds:

$$\lim_{n\to\infty} e(n) = \lim_{x\to\infty} \sqrt{x} \tag{3}$$

with $e(n)$ approaching $\sqrt{x}$ from above (Chess Programming Wiki, Alpha-Beta, 2018). This means that the relative number of evaluated nodes decreases with a greater node number and hence with greater depth, the algorithm's efficiency compared to minimax improves.

3.7 Improvements upon standard α–β

There are several ways to make α–β pruning faster. In *Mockfish*, I implemented the following two ideas:

3.7.1 Sorting

What makes α–β pruning more efficient than minimax is the fact that some parts of the tree can be pruned away because they aren't impactful for the value of the position. However, even α–β still evaluates lots of nodes that are actually obsolete. The reason why is that often, the node that could cause a cutoff only comes after many unnecessary nodes that neither change the evaluated score nor cause a cutoff. To reduce the time spent evaluating useless nodes, it is possible to sort the tree's nodes before stepping through it with the α–β algorithm. The goal of this sorting is to make the α–β algorithm evaluate nodes that cause cutoffs first.

This is done in the following way:

```
// Revised alphabeta algorithm
int Alphabeta(int depth, Position &pos, int alpha, int beta) {

      {...}
      if (pos.turn == WHITE) {
            if (depth >= 3)
                  sort(daughters.begin(), daughters.end(),
                        [](Position *a, Position *b) { return a->HPE() >
                           b->HPE(); }
            {...} // the code already shown in the standard alphabeta code
      }
      else {
            if (depth >= 3)
                  sort(daughters.begin(), daughters.end(),
                        [](Position *a, Position *b) { return a->HPE() <
                           b->HPE(); }
            {...}
      }
      {...}
}
```

Of course, this doesn't place all cutoff-causing nodes at the start because the heuristic evaluation is in no way exact enough — if that were the case, evaluating a search tree wouldn't be necessary. However, it does reduce the number of nodes being unnecessarily evaluated by a lot and thus almost halved the evaluation time in my case.

The reason why the listing above doesn't sort the three lowest level of the tree is because the benefit of sorting is eclipsed by the time it actually takes to sort the tree since the number of nodes in the $(n+1)^{\text{th}}$ level of the tree generally contains about 30 times as many nodes as all n levels before.

3.7.2 Multithreading

By using a C++11 feature called *threads*, it is possible to evaluate nodes of the same tree at the same time. A thread is a function that is executed along with the `main` thread in the same processor and that has access to the same resources and memory addresses.

With minimax, multithreading would be a clear improvement — for n daughter nodes of the root, the evaluation time would be cut down n-fold if every one of these nodes had its own thread. However, the nature of α–β pruning makes this more complicated: the entire concept of α–β pruning depends upon the fact that daughter nodes are evaluated after one another so that the pruning can actually be effective. If every thread had to treat its daughter node as a mother node itself, the number of cutoffs would be reduced which, in turn, would reduce performance. To circumvent this problem, I used a variable called `cutoff_val` shared amongst all threads that partly replaces `alpha` and `beta`.

The following listing contains the code used for multithreaded position evaluation:

```
// Evaluation function within a thread
void Evaluate(vector<Position*> positions, int depth, int *cutoff_val) {

     if (u->turn == WHITE) {
          for (auto u : positions) {
               *cutoff_val = max(*cutoff_val,
                    Alphabeta(depth, u, INT_MIN, *cutoff_val));
          }
     }
     else {...}
}

// Evaluator for the mother node
int Position::Evaluate(int depth, int threads) {

     vector<thread> thr;
     thr.resize(threads);

     CreateDaughters();

     vector<vector<Position*>> positions;
     positions.resize(threads);

     for (int i = 0; i < daughters.size(); i++)
          positions.at(i % threads).push_back(daughters.at(i));

     int cutoff_val = ((!turn) ? INT_MAX : INT_MIN);
```

```
    for (int i = 0; i < threads; i++) {
        thr.at(i) = thread(
            Evaluate, positions.at(i), depth - 1, &cutoff_val);
    }

    {...} // Check if the threads are still running

    if (turn == WHITE) {
        sort(daughters.begin(), daughters.end(),
            [](Position *a, Position *b) -> bool { return a->eval >
               b->eval; });
    }
    else {...}

    return daughters.front().eval;
}
```

The root evaluator First, the main evaluation function (`int Position::Evaluate(...)`) creates a two-dimensional dynamic array whose elements contain position pointers. Such an array is used to increase functionality: if the number of threads is lower than the number of daughter nodes of the position object, a single thread has to take care of more than one node. Thus, every element of the `positions` array contains an array of nodes. Because the arrays use the `vector` template, dealing with empty arrays isn't a problem, either if the number of threads is larger than the number of nodes. `positions` is then filled with daughter nodes so that every thread has as few nodes as possible.

Afterwards, the cutoff value is set either to the minimal value (for the maximizing player) or the maximal value (for the minimizing player). Note that those values are equal to the values of the `alpha` and `beta` variables in the default α–β function without threads. Then, the threads are run while in the main thread, a loop checks for threads that are finished already and terminates them. Afterwards, by sorting the daughter nodes according to their evaluation, the best value (for the moving player in the root node) is determined and returned.

The thread function evaluator The first function evaluates an array of position pointers. After every time using the α–β algorithm on one daughter node, the cutoff value is modified like the `alpha` and `beta` variables in normal α–β. Since the cutoff variable is shared amongst all threads, changing this variable also affects other α–β calls across the entire program. In the end, multithreading resulted in increased performance by about 200%[6]— comparable to my sorting algorithm.

[6]With a smaller number of threads, the benefit gained by this shared cutoff value is increased because a single thread has to take care of more than one position. Meanwhile, with a large number of threads, the benefit is reduced because it only has an impact on a small number of nodes. However, a larger number of threads results in more positions being evaluated simultaneously — it's the goal to find the most efficient balance.

4 Conclusion

I am proud to say that I have reached the goal I'd set myself on page 1. While *Mockfish* doesn't beat me on a regular basis, it won on several occasions. In order to compare *Mockfish*'s strength with players all over the world, I created a `https://www.chess.com/` account for my engine where its ELO score stabilized around 1300 when playing with the 10 minute time format and a depth of 6.

4.1 Statistical remarks

Interestingly, on this account, *Mockfish* had much better results with the black pieces than with the white pieces: of the first 22 games that it took to reach ELO 1300, *Mockfish* had the scores 5-2-3[7]with the white pieces and 10-0-2 with the black pieces. Most probably, the reason for that is the opening: because *Mockfish* doesn't have access to an opening book like many modern chess engines like *Stockfish* do, it tends to play quite unorthodox moves in the opening that may not be the best moves but are still among the top 5 moves in most positions. This left the white players quite confused because they couldn't play out their favorite openings, so they made concessions and mistakes that *Mockfish* could take advantage of[8]. However, with the white pieces, the advantage of playing unorthodox openings turned into more of a curse: players at the 1300 level know better how to take advantage of white's bad moves because often they cannot follow their favorite openings with the black pieces[9].

4.2 Personal remarks

Another interesting discovery I made was that *Mockfish* played in a quite similar style to my own: as opposed to the daring tactics and spectacular sacrifices *AlphaZero* is known for, *Mockfish* — like me — prefers a slow, positional game where both sides attempt to gain the positional advantage required to launch a well-guarded attack. The reason for this playing style can be found in the HPE tables: while a more aggressive player might have valued piece positioning more than the material value of the pieces, my lookup tables rewarded a cautious pawn structure and slow, positional play. Even though my own creation has defeated me, it can never really leave its creator behind.

[7]The first number stands for the number of wins, the second for the number of draws and the third for the number of losses. In chess, a win counts as 1 point, a draw as 0.5 and a loss as 0. Thus, *Mockfish* reached 6/10 points with the white pieces and 10/12 points with the black pieces.

[8]One opening that illustrates this is the Scandinavian Defense. According to the Lichess Opening Explorer, it only makes up about 2.2% of all openings after 1.e4, but it is in no way bad per se and gives black good chances of equalizing if the opponent doesn't know the theory — which is exactly what happened when *Mockfish* played inexperienced players using this rare opening, which is, coincidentally, also one of my favorites.

[9]This problem is made evident by the Center Game which *Mockfish* likes a lot. Reached by the moves 1. e4 e5 2. d4, it is quite similar to the Scandinavian Defense, but three times rarer. It is generally seen as inferior because it throws white's advantage of the first move out of the window.

5 Sources

Information

[1] Wikipedia. (2002). *Alpha-beta pruning.* [online] Available at: https://en.wikipedia.org/wiki/Alpha-beta_pruning [Accessed 05.2019]

[2] Lague, S. (2018). *Algorithms Explained — minimax and alpha-beta pruning.* [online] Available at: https://www.youtube.com/watch?v=l-hh51ncgDI&t=577s [Accessed 05.2019]

[3] Jones P. (2017). *Generating legal chess moves efficiently.* [online] Available at: https://peterellisjones.com/posts/generating-legal-chess-moves-efficiently [Accessed 07.2019]

[4] Romstad T., Costalba M., Kiisiki J. (2008). *Stockfish Chess.* [online] Available at: https://stockfishchess.org/ [Accessed 20.09.2019]

[5] Kerrigan T. (1997). *TSCP.* [online] Available at: http://www.tckerrigan.com/Chess/TSCP/ [Accessed 20.09.2019]

[6] Berent, A. (2010). *Guide to Programming a Chess Engine.* [online] Available at: http://www.adamberent.com/wp-content/uploads/2019/02/GuideToProgrammingChessEngine.pdf [Accessed 04.10.2019]

[7] Angry Bits. (2013). *Chess board move generations.* [online] Available at: http://blogs.skicelab.com/maurizio/movegen.html [Accessed 04.10.2019]

[8] Chess Programming Wiki. (2018). *Perft Results.* [online] Available at: https://www.chessprogramming.org/Perft_Results [Accessed 05.10.2019]

[9] Chess Programming Wiki. (2018). *Evaluation.* [online] Available at: https://www.chessprogramming.org/Evaluation [Accessed 07.11.2019]

[10] Chess Programming Wiki. (2018). *Alpha-Beta.* [online] Available at: https://www.chessprogramming.org/Alpha-Beta [Accessed 11.10.2019]

[11] Chess Programming Wiki. (2018). *Thread.* [online] Available at: https://www.chessprogramming.org/Thread [Accessed 11.10.2019]

[12] Chess Programming Wiki. (2018). *Parallel Search.* [online] Available at: https://www.chessprogramming.org/Parallel_Search [Accessed 11.10.2019]

[13] Chess Programming Wiki. (2018). *Board Representation.* [online] Available at: https://www.chessprogramming.org/Board_Representation [Accessed at 12.10.2019]

[14] Chess Programming Wiki. (2018). *10x12 Board.* [online] Available at: https://www.chessprogramming.org/10x12_Board [Accessed 12.10.2019]

[15] Labelle, F. (2015). *The longest possible chess game, and bounds on the number of possible chess games.* [online] Available at: http://wismuth.com/chess/longest-game.html [Accessed 10.11.2019]

[16] Klein, M. (2017). *Google's AlphaZero Destroys Stockfish In 100-Game Match.* [online] Available at: https://www.chess.com/news/view/google-s-alphazero-destroys-stockfish-in-100-game-match [Accessed 21.11.2019]

[17] Monokroussos, D. (2019). *LEELA ZERO WINS TCEC 15.* [online] Available at: http://www.thechessmind.net/blog/2019/6/1/leela-zero-wins-tcec-15.html [Accessed 21.11.2019]

[18] Seirawan, Y. (2010). *Winning Chess Endings.* 6th ed. London, England: Everyman Chess, 237 p.

[19] Claude, S. (1950). *XXII. Programming a Computer for Playing Chess.* New York, USA: Bell Telephone Laboratiories, Inc., 18 p.

[20] Silver D., Hubert T., Schrittwieser J., Antonoglou I., Lai M., Guez A., Lanctot M., Sifre L., Kumaran D., Graepel T., Lillicrap T., Simonyan K., Hassabis D. (2018). *A general reinforcement learning algorithm that masters chess, shogi and Go through self-play.* London, England: DeepMind and University College London, p. 32.

[21] Markushin Y. (2013). *The Opposite Color Bishops Endgame.* [online] Available at: https://thechessworld.com/articles/endgame/the-opposite-color-bishops-endgame/ [Accessed 25.11.2019]

[22] Lichess Opening Explorer [online] Available at: https://lichess.org/analysis/ [Accessed 27.11.2019]

[23] Yaoqing G, Marsland T. (1996). *Multithreaded Pruned Tree Search in Distributed Systems* [Accessed 05.12.2019]