Assignment of bachelor's thesis

Title:	Generating valid chess positions using AI methods
Student:	Lukáš Tomáš Petrželka
Supervisor:	Ing. Mgr. Ladislava Smítková Janků, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2023/2024

Instructions

The aim of this work is to design a system for generating valid chess positions satisfying predefined constraints. The constraints are e.g. specification of the number and types of player pieces, the level of players in whose game the chess position was created, etc. Assume constraints applied to only one of the players.
1. Study the available literature on the problem.
2. Design an appropriate player model (use NN and game tree search algorithms), design an evaluation function to control the game.
3. Using a database of chess games played, perform data collection for training.
4. Implement and train the model, design experiments and evaluate them.

1. https://database.lichess.org/
2. Chess Programming Wiki https://www.chessprogramming.org/Main_Page
3. Michael D. Steinberg, Zachary R. Jarnagin: Monte Carlo Tree Search, Neural Networks, & Chess, Foundations of Artificial Intelligence, Spring 2021, Northeastern University, Khoury College of Computer Sciences
4. Li, X., He, S., Wu, L., Chen, D., Zhao, Y. (2020). A Game Model for Gomoku Based on Deep Learning and Monte Carlo Tree Search. In: Deng, Z. (eds) Proceedings of 2019 Chinese Intelligent Automation Conference. CIAC 2019. Lecture Notes in Electrical Engineering, vol 586. Springer, Singapore.

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 22 December 2022 in Prague.

Bachelor's thesis

GENERATING VALID CHESS POSITIONS USING AI METHODS

Lukáš Tomáš Petrželka

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Ing. Mgr. Ladislava Smítková Janků, Ph.D.
February 15, 2024

Citation of this thesis: Petrželka Lukáš. *Generating valid chess positions using AI methods.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

List of code listings

I would like to thank my supervisor Ing. Mgr. Ladislava Smítková Janků, Ph.D., for her supervision throughout this project.

Declaration

In Prague on February 15, 2024 . . . .

Abstract

The focus of this work is to investigate the generation of chess positions that satisfy given constraint using neural networks in conjunction with game tree traversal. An analysis of the chosen constraint is performed, including testing whether there are classes of constraints with similar properties. An algorithm is proposed to generate chess positions that satisfy the constraints, using the Negamax algorithm modified for the purpose of this work. In total, two experiments are performed, where the first experiment tests this algorithm for fewer combinations of the chosen constraint. The second experiment extends the algorithm to more combinations and tests whether the depth of recursion when traversing the game tree has an effect on constraint satisfaction.

Keywords chess, constrains, generating, position, ai, minimax

Abstrakt

Tato práce se zaměřuje na výzkum generování šachových pozic, které splňují zadané omezení, a to s využitím neuronových sítí ve spojení s procházením stromu hry. Je provedena analýza zvoleného omezení, která mimo jiné testuje, zda existují nějaké třídy omezení s podobnými vlastnostmi. Pro generování šachových pozic splňující omezení byl navržen algoritmus, který využívá Negamax algoritmus, modifikovaný pro účely této práce. Dohromady jsou provedeny dva experimenty, kde první experiment testuje tento algoritmus pro méně kombinací zvoleného omezení. Druhý experiment rozšiřuje algoritmus na více kombinací a testuje, zda hloubka rekurze při procházení stromu hry má vliv na splnění omezení.

Klíčová slova šachy, omezení, generování, pozice, ai, minimax

List the abbreviation

CNN	Convolutional Neural Network
PDR	Piece Difference Ratio
ReLU	Rectified Linear Unit
SAN	Short Algebraic Notation
UCT	Upper Confidence bounds applied to Trees

Introduction

Chess is a popular and complex game that has been studied for many years. Recently there has been a lot of research into how to create a system in chess that would beat even the chess grandmasters. Due to the complexity of the game tree it is not possible to find the perfect move by brute force alone. But chess systems such as AlphaZero or Leela Chess Zero have achieved very good results using neural networks. However, a problem that hasn't been explored as much is that of satisfying a constraint. Such a constraint might be, for example, the number of pieces that should be in the resulting position, or whether the position should contain a castling. How do you generate such a position? And how to make such a position legal, i.e. not against the rules of the game?

The aim of this thesis is to create a system that is capable of generating one or more chess positions that satisfy a given constraint, while at the same time ensuring that these positions are always valid. This is done using neural networks and game tree search.

Chapter 1

Definition of bachelor thesis aims

1.1 Objectives

The main objective of this work is to create a system capable of generating valid chess positions while satisfying pre-defined constraints. The constraints are applied only to the white player. It does not matter how much the generated position makes sense from the perspective of intelligent play, but it must be possible to get into that position. It doesn't necessarily have to be a final position where one of the players got checkmate and lost. It can be at the beginning of the game, in the middle of the game, the position can contain a check, it can contain a checkmate, white can be at a big disadvantage positionally and materially, ... Examples of some possible restrictions:

1. Pieces requirements - specifying the exact number and types of player pieces
2. Positional requirements - specifying exact position where some pieces shall be
3. Player skill level - how skilled players could get into such position
4. Castling possibility - specifying whether player is allowed to castle (possibly if queen side, king side or both)
5. Check requirements - requiring the generating position to include check, to constrained player or the opponent

Constraints could, of course, be combined if that would make sense. Each constraint could theoretically require a different approach to generating a position that satisfies it, not to mention a combination of several different constraints. To simplify the overall problem, I will focus on solving just one constraint, **Piece requirements**.

I define the selected constraint as an ordered five numbers (Q, R, B, N, P), where:

$$\begin{aligned}
Q &: \text{number of Queens} \in \{0, 1\}, \\
R &: \text{number of Rooks} \in \{0, 1, 2\}, \\
B &: \text{number of Bishops} \in \{0, 1, 2\}, \\
N &: \text{number of Knights} \in \{0, 1, 2\}, \\
P &: \text{number of Pawns} \in \{0, 1, 2, \ldots, 8\}.
\end{aligned}$$

Each number represents the number of pieces of a specific type that must be on the board in order for the constraint to be satisfied. Of course, the king is not included in the constraint, since every valid chess game must contain a king according to the rules. The king cannot be

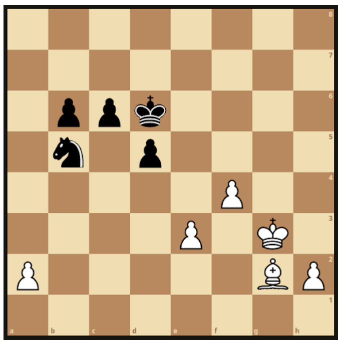


Figure 1.1 An example of satisfied $(0, 0, 1, 0, 4)$ constraint.

captured. An example of a chess position that satisfies the constraint $(0, 0, 1, 0, 4)$ is shown in figure number 1.1. Such a constraint contains no queen, rook or knight, but only 1 bishop and 4 pawns.

Since there are a large number of combinations of this constraint, exactly 486, a limited number of combinations is chosen to fit the scope of the bachelor thesis, namely 17.

In total there can be 486 combinations of the chosen constraint. A limited number of combinations of this constraint type were chosen to fit the scope of the bachelor thesis, namely 17 constraints.

The first step is to create a chess engine. I will not use an existing chess library. Instead I will create our own engine, using and describing some existing techniques. The downloaded dataset, which has the extension `.pgn`, contains moves written in SAN (Short Algebraic Notation) notation. In order to analyse and use the data it is necessary to convert this format into another one. Therefore the next task is to write a SAN parser. It will be used to parse the data, which will be used for analysis to better understand the constraints.

The next part is to design a neural network, train it and use it to create a system capable of generating valid positions that satisfy the constraints. The final step is to test the system through experiments and evaluate its success.

To summarize, the objectives are:

1. Create chess engine
2. Perform data collection
3. Write San parser
4. Analyse data
5. Implement Model
6. Create system capable of generating position which satisfy given constraint
7. Design experiments and evaluate them

1.2 Possible applications

There could be various benefits and applications of this work. Some of them:

1. Dataset generation - generated data could be used as training data for neural networks learning to play chess in specific situations
2. Puzzle generation - the system could be used for generation of chess puzzles, such as proof games
3. Exploration of game variants

Chapter 2

Analysis of the current state of the problem

This chapter begins with an introduction to the complexity of chess. Furthermore, the problem of proving the legality of a given chess position is discussed in section 2.2. Since I have programmed a chess engine, some basic techniques used in modern chess engines are explained. Modern chess AI programs typically combine a game tree search algorithm with a neural network as an evaluation function. For this reason 2 commonly used search algorithms are presented in section 2.4. The next section discusses which approaches have been successfully used to master chess. And the last section is an introduction to the problem of satisfying constraints in chess.

2.1 Important definitions

It is important to begin with a definition of the chess position in order to make it clear what exactly the subject of this work is.

▶ **Definition 2.1.** *A chess position is the arrangement of all the pieces on the chessboard.*

It would be fairly easy to create a system which could generate chess positions according to this definition. However, the task of this work is to generate **valid** chess positions. What exactly does it mean that a position is valid?

▶ **Definition 2.2.** *A chess position is **valid/legal** if it can be reached from the initial position by legal moves.*

In other words, a chess position (also called a *position* in this thesis) is valid if there is a valid chess game (no rules of chess are broken) which leads to that position. An example of a position that is **not** valid is a position with 10 white pawns or a position where both kings are in check.

2.1.1 Chess Game Tree

A game tree is a term from game theory for a tree graph representing all the states of a game. It's complexity can be measured with **branching factor**. It corresponds to the number of children at each node. If nodes have different numbers of children, the **average branching factor** can be calculated. [1]

In the context of chess, the game tree contains all possible sequences of moves leading to the final position (checkmate, stalemate, draw, ...). The average branching factor is **35-38** moves per position. In other words, the number of possible moves increases exponentially with each move. The table 2.1 shows how much it grows with each half-move (a move consists of white's half-move and black's half-move).

Table 2.1 The number of possible positions in the first 10 half-moves (5 moves).

Number of half-moves	**Number of possible positions [2]**
1	20
2	400
3	8,902
4	197,281
5	4,865,609
6	119,060,324
7	3,195,901,860
8	84,998,978,956
9	2,439,530,234,167
10	69,352,859,712,417

The American mathematician Clause Shannon estimated the number of possible chess positions to be about 10^{43}. This includes some illegal cases and excludes legal positions after captures and promotions. He also gave a lower bound on the complexity of the game tree. It's approximated to be 10^{120} and the number is called **Shannon number**. It represents the minimum complexity of the chess tree. Of course, various pruning heuristics and optimisation techniques can be applied to reduce the number of branches explored and thus reduce the complexity. Nevertheless, the complexity of the tree remains extremely high, indicating that brute forcing through the game tree is computationally impossible. [3]

2.2 Problem of verifying valid chess position

It is not enough to verify basic properties such as:

1. Contains black and white kings that are not directly adjacent to each other
2. Does not contain an unreachable number of pieces (for example, 10 queens - only 8 queens (can be reached with pawn promotion) + 1 from the beginning of the game)
3. Pawn is not in the last rank
4. If a player has all pawns and two bishops, they are both on different coloured squares
5. A player who is not on the turn may not be in check
6. No more than 6 pawns in a row
7. ...

And one could come up with many other similar controls. By performing these checks it is possible to verify most chess positions, but there are other positions whose legality cannot be verified without knowing the history of previous moves. For example, elements such as impostors, blocking, retro-shielding, casting right disruption, en passant and Ceriani-Frolkin promotions, dead positions and many other such "obscure" matters. Simple rules won't be enough to check them and it is necessary to look back at previous possible moves. This technique is called retrograde analysis in chess and many different chess problems are based on it.

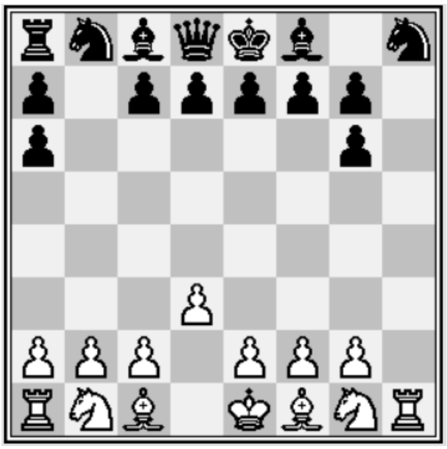


Figure 2.1 An example of Ceriani-Frolkin theme. [4]

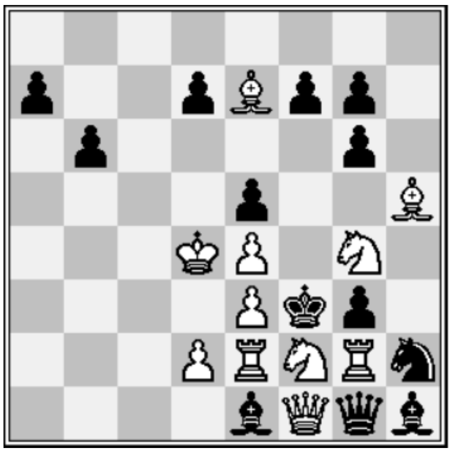


Figure 2.2 Quadruple Ceriani-Frolkin theme. [4]

Take the Ceriani-Frolkin problem as an example. This is one of the well-known classes of problems in retrograde analysis and can be defined as **promotion followed by capture of the promotee**. One such example is in the figure 2.1. You can see that White is missing a queen and the h2 pawn, while Black is missing a rook. Black had to take the b7xa6 and h7xg6 pawns in the previous moves. He had to take the queen with one pawn and the promoted **h** pawn with the other. If Black had taken White's queen on a6, there would have been no way for him to get his second pawn on the g6 square, since White's only remaining piece is the **h** pawn, and he certainly could not have got the **g column** pawn. In order to get his knight to the h8 square, he had to do so before moving his h7xg6 pawn. So this is what must have happened:

1. Black first gets his rook on the **g8** square and his knight on the **h8**
2. White then moves his queen to the **g6**, where the black pawn takes it
3. White takes his **h2** pawn to the h7 square, from where he promotes his pawn by taking the black rook to **g8**

White and his promoted piece then had to get from **g8** to **a6**, where Black then took it with a pawn. It certainly couldn't have been a rook or a bishop, because he wouldn't have taken them off. It isn't sure whether he did it with a bishop or a queen. Nor it isn't sure that he did not arrive on the square with his knight from the first rank, promote the pawn to the next knight, which he then used to return to the starting position. Another even more complex issue can be seen in the picture 2.2, which contains four Ceriani-Frolkin promotions. It is clear that checking the legality of a position is not a trivial problem and cannot always be done without looking at possible previous moves.

Therefore, to check the legality of a chess position, it is usually necessary to find a **sequence of consecutive valid moves from the initial position to the generated position**. Such a problem is called a **proof game problem**. It is a type of retrograde analysis chess problem where the parties work together to find such a sequence of moves. There are different types of proof games, such as the shortest proof game. In this variant the solver has to find the shortest sequence of moves from the starting position to the given position. Although this is a common variant, it's not the case here. It's enough to find any sequence of moves, i.e. of any length, not necessarily the shortest.

But in the worst case you have to go through all possible move sequences to prove that it's not possible to get into such a position, so it's not valid. For positions in the later stages of the game the size of the backtracked tree grows dramatically (as the branching factor is 35 - 40) and

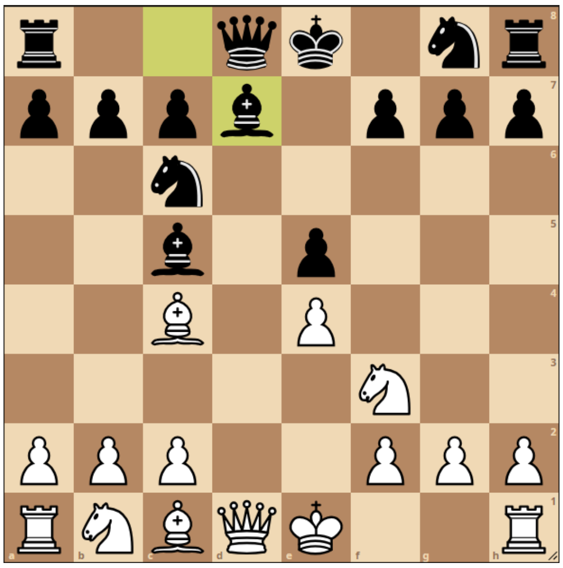


Figure 2.3 White to move: Is castling possible in this position?

it is computationally impossible to traverse the entire tree. And not only in the later stages of the game. After only the 10th half-move (5 moves) there are 69,352,859,712,417 possible games and after the 11th there are already 2,097,651,003,696,806 (about 30 times more). This increase in the number of possible games suggests that this is potentially an NP-complete problem. [2]

Not all of these specific cases require the construction of the entire sequence of moves leading to a given position, some heuristic may be sufficient. The problem becomes more complex by increasing the number of pieces on the board and by changing the types of pieces. However, it is still a more complex problem that cannot always be solved by simply checking the current position.

Nevertheless, there is another reason to know at least something about the history of the game in order to make better use of the generated position. Let's say the user generated the position shown on image 2.3 because he wants to practise to get better at chess. Looking at the position, it looks as if the king hasn't moved. Castling is generally an important objective. It keeps the king safe and develops the rook into a more active position. So it might be a good choice in this current position. But what if the king left the square in the previous moves and then returned to the starting position? The reason for this would be the check by the black bishop. In this case, castling wouldn't be possible. The player can't be sure that it's possible to make kingside castle. The same can happen with an en passant move and so on. The number of identical consecutive moves is also an important additional piece of information because of the threefold repetition rule. This states that a player can claim a draw if the same position is repeated 3 times. In the context of this work I only need to generate a valid chess position. However, for a more efficient use of the generated positions it would be necessary to have this additional information.

2.3 Chess engine - board representation

The board representation is needed for the current board state, search and evaluation. A correctly chosen representation can speed up the whole process of finding possible moves. The most common representations are **Two-dimensional array**, **One-dimensional array** and **Bitboards**.

Two-dimensional array is the most intuitive, where the board is represented as an 8x8 array of squares. Each square contains a value representing a piece. Addressing squares is fairly simple: one index represents rank number (row) and the other one represents files (column) number. Navigating the board is also straightforward. Increase or decrease the rank to move forward or backward, and increase or decrease the file to move right or left.

The main advantages of this representation are its intuitiveness and simplicity of implementation. However, it can be inefficient and potentially a little slow. Two variables are needed to access a square and twice as many to generate moves, which adds a layer of complexity. Two variables for the starting square and two more for the destination square. Two nested loops are needed to iterate over the whole board.

These disadvantages may be insignificant and probably won't affect performance, but the main problem arises when performing the move generation. The move generation process is critical and needs to be highly efficient due to its frequent use. Since 2D static (continuous) arrays are stored internally as linear lists of elements, when accessing square board[rank][file] the compiler calculates *rank*8+file* and uses this as an index. Frequent conversions (such as when generating moves) can lead to a loss of performance. [5]

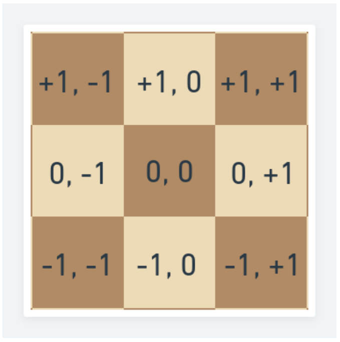


Figure 2.4 Navigating in 2D array. [6]

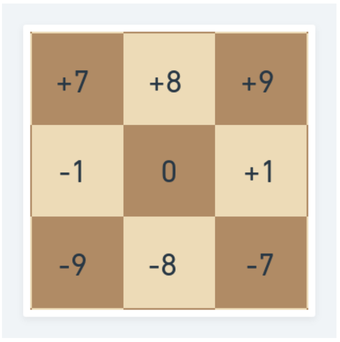


Figure 2.5 Navigating in 1D array. [6]

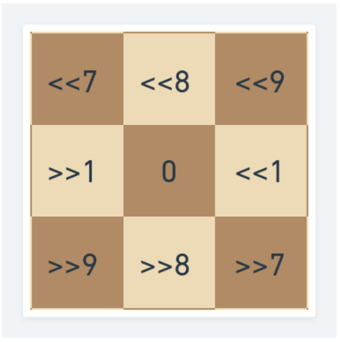


Figure 2.6 Navigating in bitboard. [6]

A little improvement is **One dimensional array** board representation. This method represents board as 1d array of 64 squares, again each one containing value that represents a specific piece. Here it is possible to directly address any square therefore avoiding potential performance loss. Moving around the board is little less intuitive, nevertheless still simple. To move up add +8 to the index, to down add -8. Similarly add +1 to move right, -1 to move left. The diagonal movements are then combination of the vertical and horizontal movements. For example to move southeast: *-8+1=-7.*

There are some potential pitfalls like boundary checking and it's less intuitive but overall this approach is more effective.

The third and last board representation discussed here are **Bitboards**. This board representation was first applied to chess in 1970. One bitboard is 64-bit word and can store the positions of all pieces of specific type, because each bit of bitboard indicates whether piece occupies the corresponding index. 12 separate bitboards are kept for every piece type (6 pieces for every side). Since almost all modern processors use 64 bits for their instructions, this technique can be very efficient. [7][8]

In terms of navigating the board it's similar to navigating in 1d array board representation however instead of addition bit shifts are used. To move up on the board calculate bit shift of 8 position. Because of these bit operations, moving around the board can be very efficient. Overall, from these three possibilities, the bitboards are the best option for chesboard representation.

2.4 Searching chess game tree

Chess is two player zero-sum game, which means that what one player gains, the other loses. For example, if white takes black's rook, white gains 5 points while black loses 5 points. As a result, the total sum of the gains is zero. Thus, this is a purely competitive game and no form of cooperation makes sense (if I help the other, I hurt myself).

In order to solve the chess algorithmically, one needs to create and traverse the **game tree**. But as already mentioned, the number of possible traversals of the tree grows exponentially with each move and it is not computationally feasible to traverse the whole tree. **Shannon number** suggest that is not possible to use brute force in order to find the best move. For this reason, game tree search algorithms have limited lookahead. There are two basic algorithms: **Monte Carlo tree search** and **Minimax**

2.4.1 Minimax

It's a backtracking algorithm which uses recursion to search though the tree until specified depth. It looks at all possible sequences of moves, evaluates the chess positions and selects the best move accordingly. When searching, it assumes that the opponent will play his best move. In other words, for the opponent it always chooses the **Min**imum value move and for the player on the move it chooses the **max**imal value. As he searches through the depths he alternates these 2 functions and therefore it is called Minimax:

Algorithm 1 MiniMax Algorithm

```
1:  function MAXI(depth)
2:      if depth = 0 then
3:          return evaluate()
4:      max ← $-\infty$
5:      for all moves do
6:          score ← Mini(depth $-$ 1)
7:          if score > max then
8:              max ← score
9:      return max
10:
11: function MINI(depth)
12:     if depth = 0 then
13:         return $-$evaluate()
14:     min ← $+\infty$
15:     for all moves do
16:         score ← Maxi(depth $-$ 1)
17:         if score < min then
18:             min ← score
19:     return min
```

Instead of minimax, usually the **Negamax** algorithm is implemented. It's a simplified variation of minimax algorithm. It's based on the fact that the evaluation of the player's position is equivalent to the negation of this evaluation from the opponent's point of view:

$$min(a, b) = -max(-b, -a) \tag{2.1}$$

This is precisely because chess is a zero-sum game. So instead of using two separate functions for *min* and *max*, negamax is used for simplicity.

Algorithm 2 NegaMax Algorithm

```
1: function NegaMax(depth)
2:     if depth = 0 then
3:         return evaluate()
4:     max ← $-\infty$
5:     for all moves do
6:         score ← $-$NegaMax(depth $-$ 1)
7:         if score > max then
8:             max ← score
9:     return max
```

2.4.2 Monte Carlo

It's a heuristic search algorithm that is based on random exploration. It consists of 4 phases:

1. Selection - starting from the root node the tree is traversed until a leaf node that has not yet been added to the tree is selected
2. Expansion - adds the leaf node to the tree
3. Simulation - plays against itself until the end of the game. It can be as simple as choosing random moves.
4. Backpropagation - the result is propagated through the tree

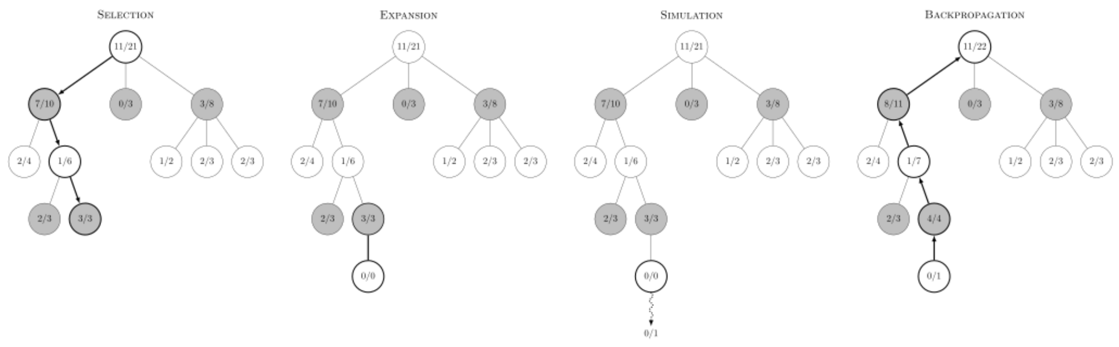


Figure 2.7 One step of Monte Carlo tree search. [9]

Pure Monte Carlo may prefer moves that lead to a loss. To solve this problem, Monte Carlo is often used with **UCT**, which stands for **U**pper **C**onfidence bounds applied to **T**rees. This combination has been successful in several games such as Go, Checkers, and **Chess** when it was first used in DeepMind's AlphaZero project, which won against Stockfish. [9]

2.5 Neural Network

First I will discuss the theory behind different types of neural networks that have been successfully used to find the (sub-)optimal move in chess.

2.5.1 Types of Neural Networks

2.5.1.1 Convolutional Neural Networks (CNN)

A convolutional neural network, also known as convnet, is a network inspired by biological processes and based on mathematical operations **convolution**. It consists of several layers, the three main types are:

- Convolutional layer
- Pooling layer
- Fully-Connected layer

The first one is **convolutional layer** and the network can contain several of them consecutively. It successively applies a filter, called a kernel, to each part of the image to see if the feature is present. This process is called convolution. A kernel (also called a feature detector) is a matrix usually 3x3 in size. It is applied to a location in the image, the dot product is calculated, and it is shifted by a fraction determined by **stride**. This is repeated until all parts of the image are exhausted. The same kernel is applied to all these parts. This is called parameter sharing and helps to reduce the total number of parameters. The output is thus a 2d array of computed point products. [10]

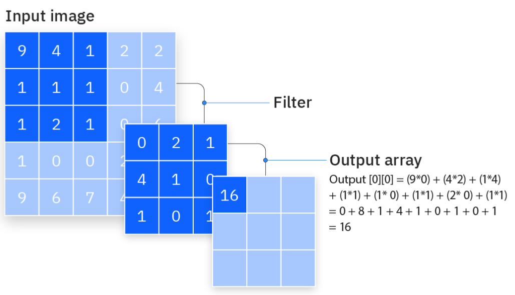


Figure 2.8 An example of convolution operation in Convolutional Layer. [11]

The generally used activation function is ReLU to increase the nonlinearity.

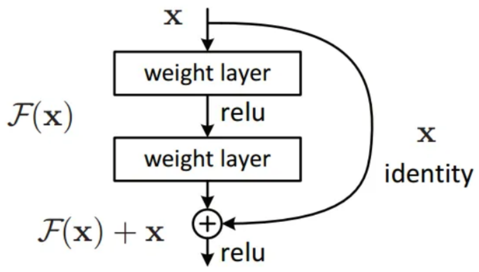
Pooling layer may be inserted after convolutional layer. It reduces the dimension of data by applying a filter across the input. There is a lot of information lost in this layer thanks to dimensionality reduction, but the main benefits are faster computation, less memory usage and overfitting prevention. Overfitting typically occurs when there isn't enought data to train neural network and the network may start memorizing the training data instead of generalizing them. However pooling helps prevent it by reducing the spatial dimensions of the input data. Two commonly used types of pooling layers are:

- Max pooling - applies a filter to parts of the image, always selecting the pixel with the highest value. It is used more often than average pooling.
- Average pooling - in contrast to the previous type, it calculates the average of the values of the pixels.

The final layer is fully-connected which computes the final classification or regression task based on the features extracted from the previous layers.

2.5.1.2 Residual Neural Networks

Residual Neural Network was introduced by paper ”Deep residual Learning for Image Recognition” in 2015. Its basic element is **Residual Block**. It is a subnetwork whose input is summed



x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 2.9 Single Residual block. [12]

with the output of this block. Thus, the output of the residual block $H(x)$ can be expressed as:

$$H(x) = F(x) + x,$$

where $F(x)$ represents the residual mapping learned by the network. An illustration of the Residual Block is shown in Figure 2.9. Several residual blocks are usually used consecutively. It also uses skip connections, which as the name suggests skips a layer in the neural network and feeds the output of one layer as the input to the next layers. That leads to better generalization on the unseen data as the network can learn identity function and skip unnnecessary or irrelevant information. [12]

One problem of very deep networks is that when training the training error gradually decreases until it hits some minimum and then the more data I add the more this error often increases. Which is counterintuitive, since the more data I use for training, the better the neural network should learn. The reasons for this are the vanishing gradient problem and the fact that it is actually hard for the neural network to learn the identity function $f(x) = x$. But the latter is needed when there already is a small training error and when there are passed data that contain no new information during training. The weights actually don't need an update and the neural network then learns the identity function. [13]

2.5.2 Input

An important aspect of a successful neural network is, of course, input in the right form. The input is most often just the chess position, but sometimes extra information can be added. There are two basic representations of the chess position: the **bitmap** and **algebraic** representations.

The first, **bitmap representation**, describes each square of the chessboard using 12 binary features. Each of these describes one particular type of piece. There are 6 types of pieces in chess (king, queen, rook, bishop, knight and pawn) and each side has its own, so $6 * 2 = 12$. A particular piece is denoted by *1* if it is standing on the square, *0* if it is not on the square, or *-1* if it is standing but belongs to the opponent. There are 64 squares, so the entire chess position is described by 768 bits in total $(12 * 8 * 8 = 768)$.

The second type of representation is **algebraic input**. It is slightly different from the bitmap representation. In addition to a description of whether the figure is on the array, it carries information about the value of the figure type. Pawns are represented by the value *1*, knights and bishops by *3*, rooks by *5*, queens by *9* and kings by *10*. The opposite values are used for opponent pieces, i.e. *-1* for pawns, *-3* for knights and bishops, *-5* for rooks, ... Otherwise each square is described by 12 flags, differing only in the values themselves. This method theoretically contains more information, because it also contain the values of each figure.

However several experiments suggest that **bitmap** representation gives better results than algebraic one. [14][15]

The term **plane** is in the next section used to identify one feature of 8x8 size in input. For example that means the if one would use **bitmap representation** which $12 * 8 * 8$ bits in input, it has 12 features.

2.5.3 Architecture of Neural Network for chess

The authors of the article "Predicting Moves in Chess using Convolutional Neural Networks" recommend Convolutional Layers for solving chess problems, as they are useful in pattern recognition of small, local tactics. However, the authors argue that CNNs should also be used with other methods such as looking several moves ahead. This is because there are important concepts in chess (such as pawn chains) that are better characterized by a heuristic function. [16]

This turns out to be true since successful systems like AlphaZero, Leelo Chess Zero use a convolutional network and combine it with a limited depth Monte Carlo tree search of the game tree, where a convolutional neural network is used to evaluate the searched position.

2.5.3.1 AlphaZero

AlphaZero is a successful project of DeepMind company with the goal to master chess and other games such as go and shogi. The program itself only after 24 hours of training gained superhuman skills to beat the chess engine programs such as Stockfish, which was until then ranked as best among chess engines.

AlphaZeo's neural network architecture is a Deep Residual Neural Network. The input is the current position and the last 7 positions represented by a bitmap representation. The reason why there are also 7 last positions on the input is because of three-fold repetitions, i.e. to prevent forced draw after 3 consecutive equal repetitions of the last position. Another 2 planes are used for number of repetitions. The first plane is set to ones if the position has occurred a second time and the second plane similarly, i.e. if the position has occurred three times, it is set to ones. Another plane is used to code who is on the move: if white, it is set to 0, 1 otherwise. Next 4 Planes are set all too 1 to if White/Black can castle queenside/kingside. The next 2 planes are used as counters. One encodes the number of moves where no progress has been made. That means where no capture happened nor no pawn moved. It is for 50 moves rules. The other counter contains the total number of moves. It is not clear what good this is, in Leelo Chess Zero it is omitted. These counters are not represented as planes, but are directly provided as numbers.

The first layer is convolutional which is than followed by multiple residual blocks. Each one has two convolutional layers followed by batch normalization and rectifier nonlinearity. It also has skip connection as it is residual. Experimented with 19 blocks and also with 39. Then there are two split heads: **policy head** and **value head**. The first one has one convolutional layer followed by batch normalization and the last layer is fully connected. The latter outputs the probability distribution of each possible move. The value head adds ReLU, another fully-connected layer terminated by a tanh activation function. It provides a single number in the range of $[-1, 1]$ indicating who will win in the current position if both play optimally. [13]

2.6 Chess and Constraint satisfaction

Chess is a competitive game, the player tries to checkmate and win. But the main aim of this work is to generate a position that satisfies some constraint. It is not an attempt to find a better position for white (or black), on the contrary, strategic concepts are not very important when trying to satisfy the given constraint. It is rather a collaborative problem. Players theoretically cooperate in order to get into a position that satisfies the constraint, and it doesn't matter how much any player is actually at a disadvantage. They have a common goal that they can achieve by cooperating. From the perspective of the given problem, this is a variant of chess in which the players win if they get into a constraint-satisfying position and, conversely, both lose if they do not. Thus, it is not possible to use a Minimax algoritmus that searches for the best move for one player or, equivalently speaking, the move that most harms the other player. However, one can simply modify the Negamax algorithm to search the game tree in cooperative play instead.

However, it would be reasonable to want a sequence of valid moves (proof game) which lead to the generated position, but which are at least to some extent **rational** in terms of competitiveness. For example, if a player can take a piece for free, he will usually take it unless it is a trap or he would lose a strategic advantage. This means that, in order to get into a position that satisfies the constraint, one might use a similar/same style of play as players trying to win (or at least draw in the case of a losing position). A possible reason for this could be that the user wants to practise different positions, so that in the course of actual play he can recognise them quickly, realise the different chess strategy properties of a given position and devise the best possible moves. He would not need to practice positions that he could never get into in competitive play. But there could theoretically be many such positions if the system were designed with the sole purpose of producing a valid chess position that meets the specifications. An example of such an "unrealistic" position, where the constraint is that White loses 3 pawns and 1 bishop, is shown in figure number 2.10. The constraint is indeed 100% satisfied, but only if White offers his pieces to Black for free and does not attempt to checkmate and thus win.

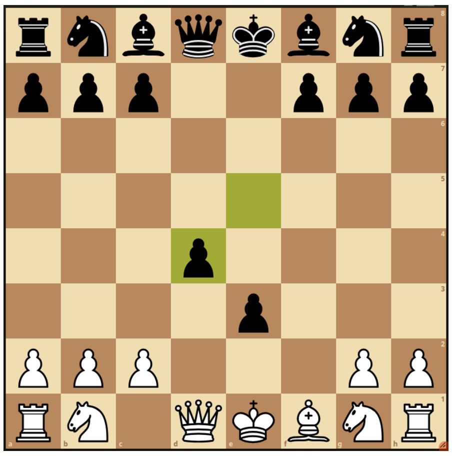


Figure 2.10 Constraint satisfied with "unrealistic" type of play.

Thus, it would require combining 2 goals: reasonable competetive moves and constraint satisfaction. Where, of course, satisfying the constraints should have a higher priority. This would of course be a more complex problem. It is not enough to make simpleminded moves that easily lead to constraint satisfaction. However, this is not the assignment of my thesis and this simplifies the whole problem.

No literature on the problem of generating valid chess positions satisfying the constraint or any similar **was found**.

Chapter 3

Data analysis

3.1 Dataset

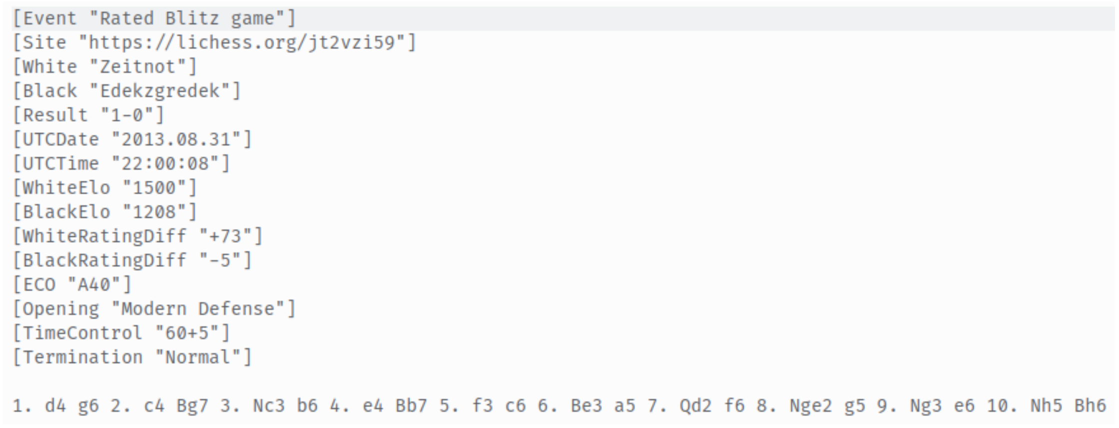
As mentioned earlier dataset was taken from `https://database.lichess.org`, where one can download hundreds of million of played chess games. The games are played by players with different skill level, from complete beginners to highly skilled professionals and they are of different time control (Bullet = 1min, Blitz = 3min, Rapid = 10 min, ...). Due to higher computational complexity and lack of performance, different sized datasets were chosen for constraint analysis and neural network training. A dataset of ∼10,000,000 chess games was used for data analysis and a dataset of ∼300,000 games was used as the input dataset for neural network training.

3.1.1 Dataset format

Dataset file has a .pgn file extension, which stands for **P**ortable **G**ame **N**otation (**PGN**). It is a standard plain text format for recording chess games. Created by Steven J. Edwards, it's structured for easy reading and writing by human users and for easy parsing and generation by computer programs. Each game contains played moves and some game-related information, such as the winner. The preview of PGN file format is shown on figure **figure 4.1**. [17]

Game-related information are in tags, enclosed [...] brackets. Downloaded pgn file contains these tags:

- **Event**: Name of the match, e.g. blitz, bullet, rated, ...
- **Site**: URL address leading to played game on Lichess website
- **White**: White's username
- **Black**: Black's username
- **Result**: The result of the played game, e.g. 1-0, 1/2-1/2, 0-1
- **UTCDate**: UTC date when game was played
- **UTCTime**: UTC time when game was played
- **WhiteElo**: Glicko2 white's raiting Elo
- **BlackElo**: Glicko2 black's rating Elo
- **WhiteRatingDiff**: The number of Elo points white gained/lost from the game result



```
[Event "Rated Blitz game"]
[Site "https://lichess.org/jt2vzi59"]
[White "Zeitnot"]
[Black "Edekzgredek"]
[Result "1-0"]
[UTCDate "2013.08.31"]
[UTCTime "22:00:08"]
[WhiteElo "1500"]
[BlackElo "1208"]
[WhiteRatingDiff "+73"]
[BlackRatingDiff "-5"]
[ECO "A40"]
[Opening "Modern Defense"]
[TimeControl "60+5"]
[Termination "Normal"]

1. d4 g6 2. c4 Bg7 3. Nc3 b6 4. e4 Bb7 5. f3 c6 6. Be3 a5 7. Qd2 f6 8. Nge2 g5 9. Ng3 e6 10. Nh5 Bh6
```

Figure 3.1 Preview of .pgn file.

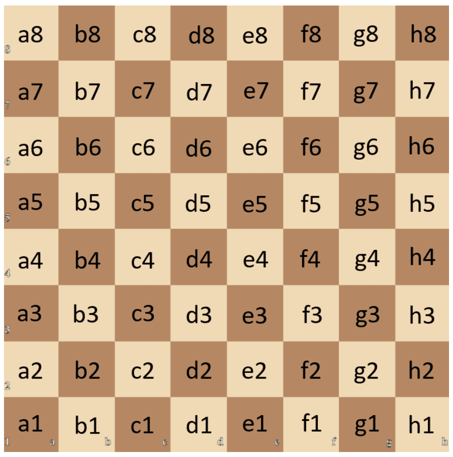


Figure 3.2 Algebraic square coordinate. [18]

- **BlackRatingDiff**: The number of Elo points black gained/lost from the game result
- **ECO**: - **E**ncyclopaedia of **C**hess **O**penings is a classification system for the chess openings move - Code of the corresponding chess opening
- **Opening**: The name of the chess opening
- **TimeControl**: Time control game variant - how much time each player has to finish the game, e.g. 60+5 means 60 seconds and extra 5 seconds for each played move
- **Termination**: Type of game termination, e.g. Time forfeit, Normal, ...

Chess moves are recorded using **Algebraic Notation**, more precisely **Short Algebraic Notation**. Algebraic Notation is a chess notation which represents chess moves in human-readable way. It's the most used chess notation among chess players. It describes the target square of the move with algebraic coordinates (except for the king/queen castle). Algebraic coordinate is a cartesian coordinate specified by two characters, a letter for the file (a-h) and a digit for the column (1-8). There are different ways to note the origin square of the moves, such as Short Algebraic Notations, Long Algebraic Notation, Smith Notation, ...

Short Algebraic Notation (SAN), also known as **Standard Algebraic Notation** is a commonly used system for recording chess moves in computer programs and official notation in all international chess competitions. SAN describes the origin square of the move with an abbreviation of piece (K - King, Q - Queen, R - Rook, B - Bishop, N - Knight) if there is no other pieces of same type which can also move to the same target square (e.g. two knights). In case of such ambiguity, rank(row), file(column) or the complete square algebraic coordinate is added after the descriptive letter of the piece. For pawn pieces, no abbreviation is needed. Specifying only the target square is sufficient to unambiguously determine the origin square of each pawn, because pawns go only forward.

3.1.2 Parsing games

Next step was to parse downloaded games, for this purpose I used C++ library **PGNP**: `https://gitlab.com/manzerbredes/pgnp/`. It parses one game at time, extracting all the game-related information from tags and all the moves. Unfortunately moves are in String SAN format, therefore I had to write SAN parser. The syntax of SAN isn't complicated:

1. **Piece moves**: <Piece symbol>[<from file> | <from rank> | <from square>]['x']<to square>
2. **Pawn captures**: <from file>[<from rank>] 'x' <to square>[<promoted to>]
3. **Pawn push**: <to square¿[<promoted to>]
4. **King or Queen castle**: 0-0 or 0-0-0

3.1.3 Feature Selection a Data Preprocessing

The dataset contains several items in the game headers that do not provide any relevant information for our specific problem and can be removed. These items are `Event`, `Site`, `White`, `Black`, `UTCDate`, `UTCTime`, `WhiteRatingDiff`, `BlackRatingDiff`, `Termination`.

`Event` contains information whether the games are rated. However, rated and unrated games by similarly advanced players in the same time format do not necessarily differ in game quality. Also, there is no point in filtering games from the dataset by this flag, since unrated games can also be used in constraint analysis, hence refining the results of the statistics. It also includes the type of time format (**Bullet** = 3 min, **Rapid** = 10 min, ...), but this is also stored in `TimeControl` and additionally gives information about the time increment (60**+5**). Thus, the `Event` partly contains duplicate information and in general does not bring any additional essential information.

The URL in `Site` leading to the game played on the Lichess page is of course useless, as are the names of the players `White`, `Black`. Also the date and time of the game played (`UTCDate`, `UTCTime`) does not matter. The `WhiteRatingDiff` `BlackRatingDiff` items indicating how many points a player gained/lost based on the outcome of the game are irrelevant, as the only thing I am interested in is the approximate level of play of the players. This is provided by `WhiteElo`, `BlackElo`. If the resulting difference in points is dramatic (not in units), it means that the player is new to the server and thus his Glicko2 rating is not sufficiently representative of. These are only minor cases, however. [19]

And the last removed item `Termination` indicates the type of game termination. I may be interested in whether a position meeting the constraint leads more often to a win, or perhaps whether games meeting the constraint lead more often to a win, but how the win was achieved is no longer relevant information. For this reason this item was also not used and therefore removed.

Table 3.1 Summary of magnitudes of change in datasets.

	Dataset for analysis	Dataset for training
Original size	**10,194,940**	**325,099**
Low ranked games	129,297	1,933
Unallowed time control	4,340,018	141,554
Too short games	106,652	2,604
Final size	**5,692,409**	**180,414**

The dataset of course contains different types of games, whether beginner, advanced, masters or different length games with different data formats, as well as different incomplete games.

For example very beginner players may overlook potential threats or opportunities and get into unnatural positions that more experienced players wouldn't get into. It could negatively affect the results obtained from analysis and possibly let neural network to learn to favour such positions - of course that depends on the architecture of the neural network, selected loss function and so on. For this reason games with players Glicko2 rating (`WhiteElo` and `BlackElo`) < 1000 were considered as beginner and so were removed.

Moreover one could expect shorter `TimeControl` game variants to differ from longer one in game play. Players have less time to notice potential threats, think it throw thoroughly and came up with decent strategy. Thus it doesn't make sense to mix them in analysis or use them together as input data for neural network training. Hence game variants with time control set to 60 seconds, 120 seconds, 180 seconds a 240 seconds were removed, regardless of whether it's also with extra seconds per move.

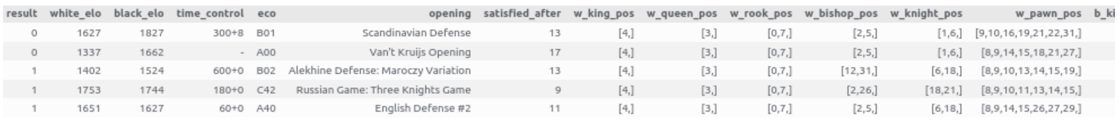
The last change I made was to remove games that were too short. In online chess games, it often happens that one of the players gives up after the second, third, ... move without losing a piece, getting into a disadvantageous position or losing strategically in any way. It is also quite common for a player to stop playing (also after a short number of moves) and either run out of time or the other player surrenders rather than to wait. Such games could make the results of the analysis more imprecise - for example, the number of games that meet the specified constraint. Of course, there are also many unfinished longer games in the dataset. Even in these cases, of course, it happens that players drop out of games. Since these are minority cases (there were only ~35 such games among the ~400,000 games when analyzed), they were left in the dataset. Moreover, in a training dataset they can be potentially beneficial. Namely, for example, to theoretically better teach the model what positions do not (or instead do) lead to satisfying a given constraint.

The table **3.1** shows the sizes of the original and final datasets and also the numbers of all removed games. After all the changes, the final datasets contained approximately one half of the games. Specifically, the training dataset contained **180,414** games and the analysis dataset contained **5,692,409** games.

3.1.4 Dataset for analysis

After all the modifications, I obtained two datasets in binary format. Prior to the analysis, I extracted the constraint information from the analysis dataset and saved it in `.csv` format files - one separate file for each constraint examined. In addition to the information I already had (in the game headers), I extracted additional data from each game. The position (if it exists) that satisfies the constraint was found and I extracted from it:

1. move number in which the constraint is satisfied
2. the position of each white piece
3. the position of each black piece



result	white_elo	black_elo	time_control	eco	opening	satisfied_after	w_king_pos	w_queen_pos	w_rook_pos	w_bishop_pos	w_knight_pos	w_pawn_pos	b_ki
0	1627	1827	300+8	B01	Scandinavian Defense	13	[4,]	[3,]	[0,7,]	[2,5,]	[1,6,]	[9,10,16,19,21,22,31,]	
0	1337	1662	-	A00	Van't Kruijs Opening	17	[4,]	[3,]	[0,7,]	[2,5,]	[1,6,]	[8,9,14,15,18,21,27,]	
1	1402	1524	600+0	B02	Alekhine Defense: Maroczy Variation	13	[4,]	[3,]	[0,7,]	[12,31,]	[6,18,]	[8,9,10,13,14,15,19,]	
1	1753	1744	180+0	C42	Russian Game: Three Knights Game	9	[4,]	[3,]	[0,7,]	[2,26,]	[18,21,]	[8,9,10,11,13,14,15,]	
1	1651	1627	60+0	A40	English Defense #2	11	[4,]	[3,]	[0,7,]	[2,5,]	[6,18,]	[8,9,14,15,26,27,29,]	

Figure 3.3 Csv dataset for analyzing.

In addition to this extracted information, the resulting dataset also contained the result of the game, the Elo of both players, the name of the opening and the `eco` code and time format of the game. These datasets were analyzed to find the properties of the constraints. Figure number 3.3 shows a piece of the `.csv` dataset used for constraint analysis. The part that is no longer visible in the figure contains additional flags - the positions of all black figures and in the same format as the white ones (flags starting with **b**).

3.2 Analysis

As mentioned above, a constraint is defined as an ordered quintuple of numbers (Q, R, B, N, P), where:

$$\begin{aligned}
Q &: \text{number of Queens} \in \{0, 1\},\\
R &: \text{number of Rooks} \in \{0, 1, 2\},\\
B &: \text{number of Bishops} \in \{0, 1, 2\},\\
N &: \text{number of Knights} \in \{0, 1, 2\},\\
P &: \text{number of Pawns} \in \{0, 1, 2, \ldots, 8\}.
\end{aligned}$$

Each number represents the number of pieces of that type that must be on the board for the constraint to be satisfied.

It is allowed to have more than one queen in a chess game. The pawn that reaches the last rank is promoted to queen, rook, bishop or knight. Thus, in theory, a player can have (on the Lichess platform) up to 9 queens and similarly for the other mentioned pieces. For the purposes of this thesis, I have limited the maximum number of pieces of each type to only as many as there are pieces of that type at the start of the game. That is, a maximum of 8 pawns, 2 knights, 2 bishops, 2 rooks and 1 queen. This is to simplify the analysis and generation of positions. The main purpose of is **not** to thoroughly understand the relationships between the constraints and to generalize them, but rather to gain basic insight into the generation of positions satisfying the constraints defined above.

Just for our definition of this constraint, there are quite a lot of different combinations - more precisely, $2 \times 3 \times 3 \times 3 \times 9 = 486$ **combinations**. Since I am not trying to generalize the relations between all constraints, I have chosen only **17 constraints**. I also selected a few more constraints that I want to see if they behave like some of the constraints already mentioned. In fact, one might assume that there are some classes of constraints that share similar properties. For example, after how many moves are most often satisfied, similar-looking positions, ... As an example, two situations (constraints) can be given: 1st - player lost 1 knight, 2nd - player lost 1 bishop. These situations can be very similar in their progression and can be reached after a similar game development. I can assume that both situations can be reached (sometimes) after the same number of moves. This will be one of the points of analysis.

With each successive move, the number of different games grows exponentially. Thus, in the shorter number of moves the constraint is satisfied, the fewer possible such positions exist. If I choose constraints that are assumed to be satisfied in the fewest number of moves (most often), I assume that such a large number of data will not be needed for the analysis - there are fewer positions. It is also possible that such constraints will be more tightly coupled since there are fewer possible developments of the game. For these reasons, constraints that are assumed to be satisfied most often in the early stages of the game were predominantly selected for analysis. [20]

The selected constraints are listed in the following table:

Table 3.2 Chosen constraints and their meaning.

	Constraint	Missing pieces
1.	(1, 2, 2, 2, 7)	1 Pawn
2.	(1, 2, 2, 1, 8)	1 Knight
3.	(1, 2, 1, 2, 8)	1 Bishop
4.	(0, 2, 2, 2, 8)	1 Queen
5.	(1, 2, 2, 1, 7)	1 Knight and 1 Pawn
6.	(1, 2, 1, 2, 7)	1 Bishop and 1 Pawn
7.	(0, 2, 2, 2, 7)	1 Queen and 1 Pawn
8.	(1, 2, 1, 1, 7)	1 Bishop 1 Knight and 1 Pawn
9.	(1, 2, 2, 0, 7)	2 Knights and 1 Pawn
10.	(1, 2, 0, 2, 7)	2 Bishops and 1 Pawn
11.	(1, 1, 2, 2, 8)	1 Rook
12.	(1, 2, 2, 2, 6)	2 Pawns
13.	(1, 2, 1, 1, 8)	1 Knight and 1 Bishop
14.	(1, 2, 2, 2, 5)	3 Pawns
15.	(1, 2, 1, 1, 6)	1 Bishop, 1 Knight and 2 Pawns
16.	(1, 2, 2, 0, 6)	2 Knights and 2 Pawns
17.	(1, 2, 0, 2, 6)	2 Bishops and 2 Pawns

3.2.1 Objects of analysis

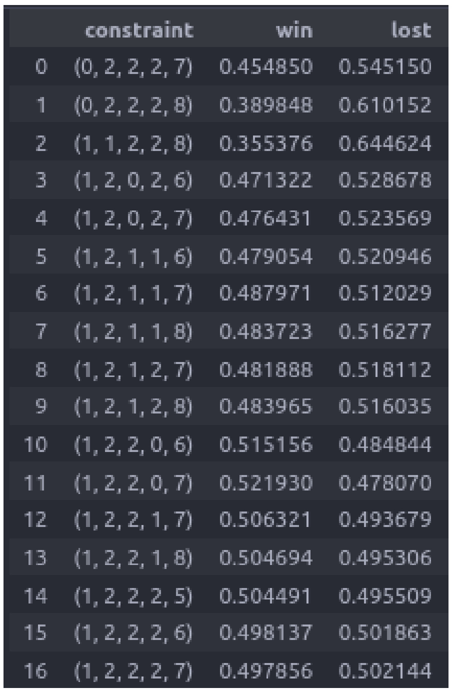
The first thing examined was whether there are any constraints that, if met, increase the chance of winning/losing or whether the constraints have no effect on winning. Figure 3.4 lists the calculated probability of winning and losing for each constraint.

It is evident that certain constraints result in a winning chance of less than 50%. For instance, if White loses their queen, the chance of winning decreases by a $\sim$ 11%.

It was also examined whether the constraints shared any similar characteristics. For each constraint, the probability distribution of the values determining the half-move in which the constraint was satisfied was calculated. Each move consisted of two half-moves - one white move, one black move. A threshold of 5% was chosen to filter out values that are rather exceptional. For example, if a given constraint was satisfied in 3% in the 23rd half-move, this number was not considered. Probability curves were created for each constraint and these curves were added to a single graph, which can be seen in Figure 3.5.

In the graph, there are always only even numbers of half moves, and this is because each constraint refers by definition to the white player. In order for the constraint to be satisfied, the white player must always lose some pieces, so it can only be satisfied by a half move by the black player. The exceptions would be two situations. The first would be when the ”constraint” is (1, 2, 2, 8), i.e. no pieces are missing. But this is trivially satisfied in the initial position before the game starts. And the other situation is when the white player reaches the last rank with a pawn, where he can promote it to any piece, thus also fulfilling the restriction with his half-move. However, I have chosen to examine constraints that I expect to be satisfied in the basic phases of the game. Thus, it is very unlikely that in our case the constraint would be satisfied by White’s half-move, and is more likely to be a minority case filtered by the chosen 5% threshold.

The graph displays multiple overlapping curves with similar courses, suggesting the existence of constraint classes with shared properties. There are a total of four ’hills’, each composed of



	constraint	win	lost
0	(0, 2, 2, 2, 7)	0.454850	0.545150
1	(0, 2, 2, 2, 8)	0.389848	0.610152
2	(1, 1, 2, 2, 8)	0.355376	0.644624
3	(1, 2, 0, 2, 6)	0.471322	0.528678
4	(1, 2, 0, 2, 7)	0.476431	0.523569
5	(1, 2, 1, 1, 6)	0.479054	0.520946
6	(1, 2, 1, 1, 7)	0.487971	0.512029
7	(1, 2, 1, 1, 8)	0.483723	0.516277
8	(1, 2, 1, 2, 7)	0.481888	0.518112
9	(1, 2, 1, 2, 8)	0.483965	0.516035
10	(1, 2, 2, 0, 6)	0.515156	0.484844
11	(1, 2, 2, 0, 7)	0.521930	0.478070
12	(1, 2, 2, 1, 7)	0.506321	0.493679
13	(1, 2, 2, 1, 8)	0.504694	0.495306
14	(1, 2, 2, 2, 5)	0.504491	0.495509
15	(1, 2, 2, 2, 6)	0.498137	0.501863
16	(1, 2, 2, 2, 7)	0.497856	0.502144

Figure 3.4 The winrate of games satisfying the constraints.

several similar curves.

1. (0, 2, 2, 2, 8), (1, 2, 1, 2, 8), (0, 2, 2, 2, 7), (1, 2, 2, 1, 8) - součet **14**, kromě (0, 2, 2, 2, 7)
2. (1, 2, 2, 2, 6), (1, 2, 1, 2, 7), (1, 2, 2, 1, 7), (1, 1, 2, 2, 8) - součet vždy **13**
3. (1, 2, 2, 0, 7), (1, 2, 1, 1, 7), (1, 2, 0, 2, 7) - součet vždy **12**
4. (1, 2, 1, 1, 6), (1, 2, 2, 0, 6), (1, 2, 0, 2, 6) - součet vždy **11**

If I count the sum of the constraint pieces in each group, I find that (with one exception) the constraints in the group always have **same point sum** pieces. This suggests that constraints with the same sum of points usually form groups with similar properties. At least in the early stages of the game where the constraints in this thesis were studied. However, there are a few exceptions, such as (1, 2, 1, 1, 8), where the bishop and the knight are missing. The sum of the points of the pieces is 13, but this constraint does not belong to this group. Further research would be needed to generalise to constraints from other phases.

Finally, I analysed what chess positions look like that satisfy the constraints. I am interested in the most common placement of each piece type. I will not present the results for every constraint, but only a few selected ones. The probability distribution of the placement for each piece type has been calculated, in addition to information about how many moves it took on average and on median to get to a given position, and what White's chances of winning are likely to be if a given piece is on that square. Let's look at the table for the constraint **(1, 2, 1, 1, 8)** (grey colour in the diagram), which was a bit "lonely" in the previous diagram. The first five results for **bishop** can be seen in the table in figure 3.6, ordered by highest probability. All the results have also been visualised in the form of a chess heatmap with a **logarithmic scale**, so

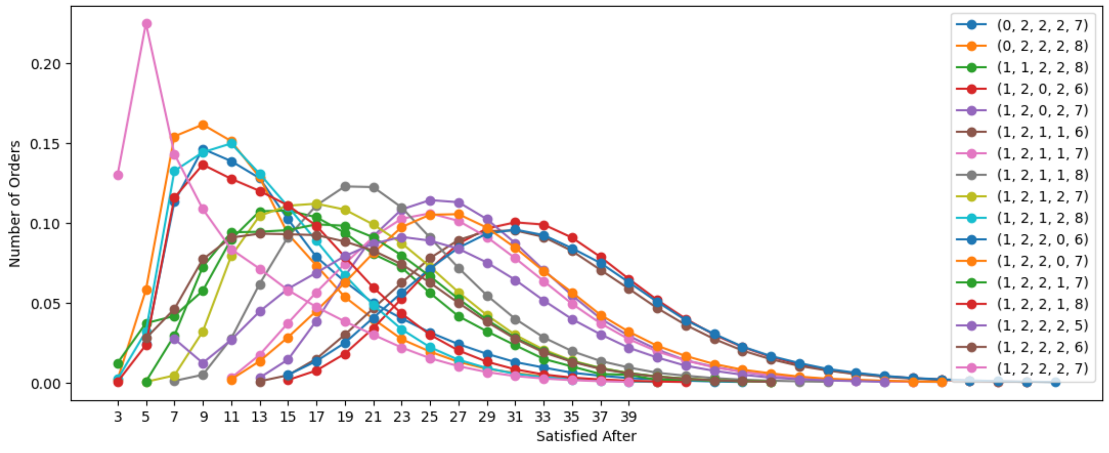


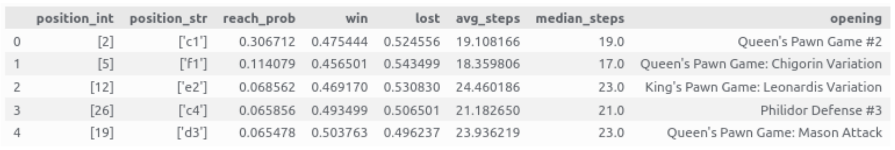
Figure 3.5 Comparison of constraints.

that even less probable placements can be seen more easily (in the second figure 3.7 also for the bishop):

Next, let's try to compare those constraints that seemed to have similar properties from the curves - they were most often satisfied in the same half move. For example, let's look at the group whose "hill" is furthest to the right. These are the following constraints: (1, 2, 1, 1, 6), (1, 2, 2, 0, 6), (1, 2, 0, 2, 6). Let's see if they have more in common. I will no longer provide a table, but just a chess heatmap for each type of piece. Comparisons of the individual pieces of these constraints are shown in the figures: king 3.8, queen 3.9, rook 3.10, bishop 3.11, knight 3.12, pawn 3.13. Indeed, the positions look very similar. The only difference that can be seen at first glance is for knight 3.12, where the constraint (1, 2, 1, 1, 6) has the left knight on the square **c3** more often than (1, 2, 0, 2, 6).

3.2.2 Conclusion of the analysis

In this section, I analyzed the dataset and for each constraint found the most frequent fields of piece types and White's chance of winning. However, the primary effort was to see if there might be any connection between certain constraints, whether they share any properties. The analysis suggests that there could indeed be certain classes of constraints that share some characteristic, such as in the presented constraints (1, 2, 1, 1, 6), (1, 2, 0, 2, 6), (1, 2, 2, 0, 6). Only basic research was conducted and more in-depth and detailed research would be necessary to confirm the results.



	position_int	position_str	reach_prob	win	lost	avg_steps	median_steps	opening
0	[2]	['c1']	0.306712	0.475444	0.524556	19.108166	19.0	Queen's Pawn Game #2
1	[5]	['f1']	0.114079	0.456501	0.543499	18.359806	17.0	Queen's Pawn Game: Chigorin Variation
2	[12]	['e2']	0.068562	0.469170	0.530830	24.460186	23.0	King's Pawn Game: Leonardis Variation
3	[26]	['c4']	0.065856	0.493499	0.506501	21.182650	21.0	Philidor Defense #3
4	[19]	['d3']	0.065478	0.503763	0.496237	23.936219	23.0	Queen's Pawn Game: Mason Attack

Figure 3.6 The first 5 items of sorted table of the most frequently used squares for bishop for (1, 2, 1, 1, 8).

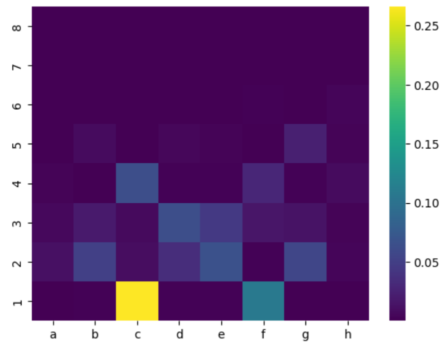


Figure 3.7 Chessboard heatmap of the most frequently used squares for bishop for (1, 2, 1, 1, 8).

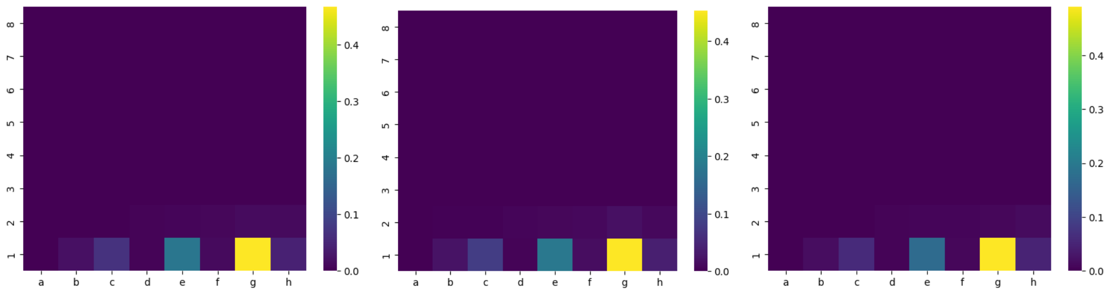


Figure 3.8 Comparison of **king** squares for constraints (1, 2, 1, 1, 6), (1, 2, 0, 2, 6), (1, 2, 2, 0, 6).

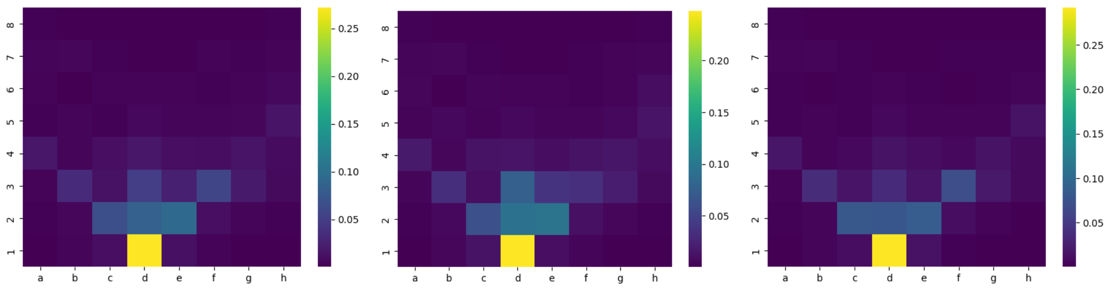


Figure 3.9 Comparison of **queen** squares for constraints (1, 2, 1, 1, 6), (1, 2, 0, 2, 6), (1, 2, 2, 0, 6).

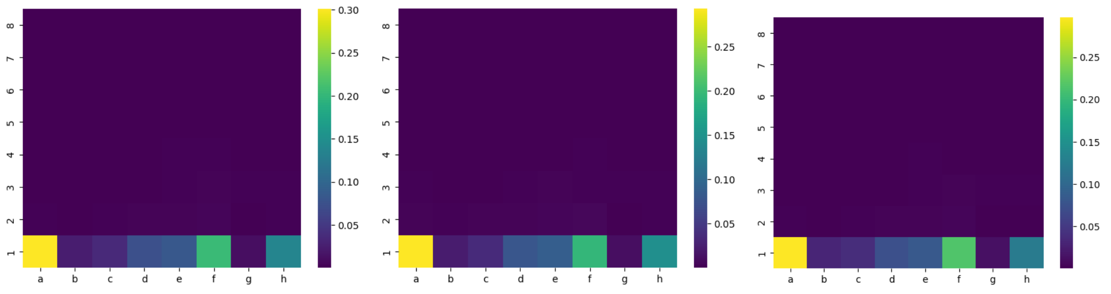


Figure 3.10 Comparison of **rook** squares for constraints (1, 2, 1, 1, 6), (1, 2, 0, 2, 6), (1, 2, 2, 0, 6).

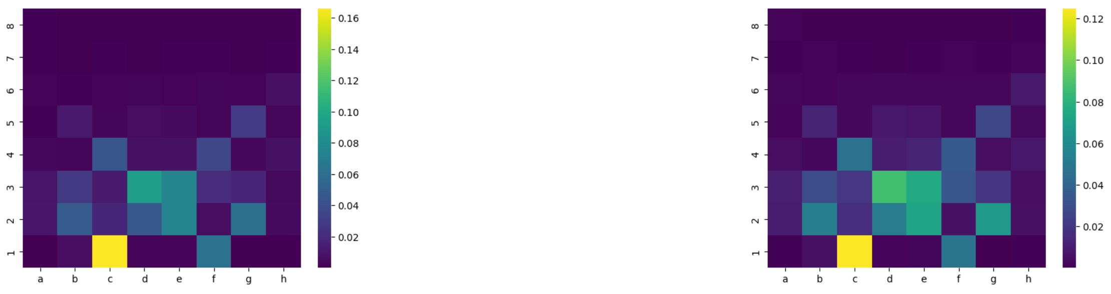


Figure 3.11 Comparison of **bishop** squares for constraints (1, 2, 1, 1, 6), (1, 2, 2, 0, 6).

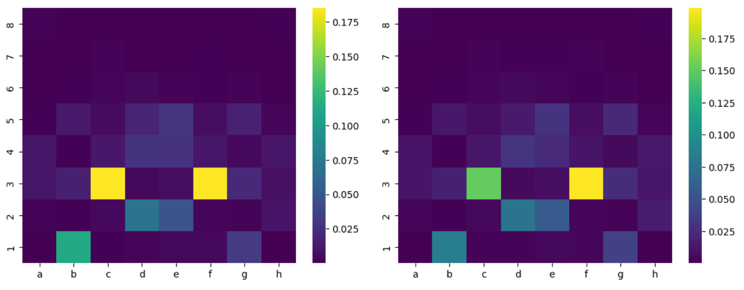


Figure 3.12 Comparison of **knight** squares for constraints (1, 2, 1, 1, 6), (1, 2, 0, 2, 6).

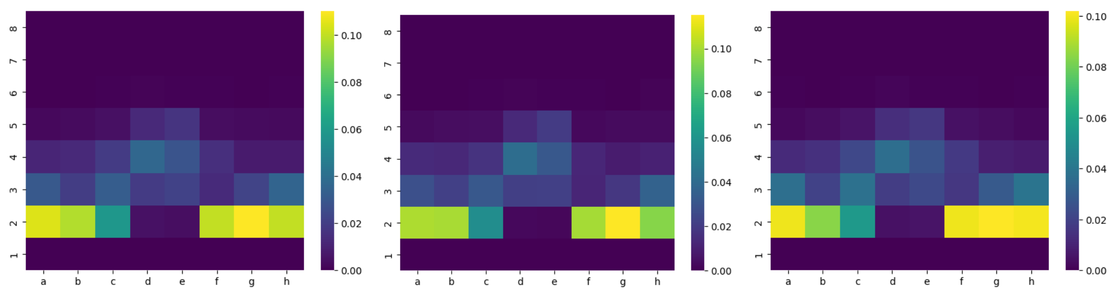


Figure 3.13 Comparison of **pawn** squares for constraints (1, 2, 1, 1, 6), (1, 2, 0, 2, 6), (1, 2, 2, 0, 6).

Chapter 4

Current status of the solution

4.1 Solution approaches

In the chapter 3 for each constraint the probability distribution of squares for each piece type for both white and black was obtained. However, I could make the analysis of the positions more detailed. For example consider the probability distribution of the square of the white king. For each placement of the king, I could then compute the conditional probability distribution of the placement of the white queen. By successive condition adding, I would compute the information for all other pieces (white and black). In this way, I would obtain complete information (within the data) about the distribution of the pieces in order to construct a legal position satisfying the constraint. It would then be sufficient to iteratively select square of a piece based on the criterion of how frequent it should be, and under the condition of that selected square, continue with the selection. Then surely the acquired data alone would be sufficient to generate valid chess positions.

This way, one could only generate positions that have been played by human players in the past. But there are many more possible positions in chess. The American mathematician Clause Shannon estimated the number of possible chess positions to be 10^{43} (this includes some illegal cases). The goal of this work is to try to create a system that is capable of generating more general chess positions, not just those previously recorded in a database of chess games. This is the main reason why it is not optimal to use this solution method. [3]

So if I want a more general system capable of generating as many positions as possible, there are two basic approaches to solving the problem of generating valid positions that satisfy the constraint:

1. Generate position satisfying constraint, then prove it's valid
2. Generate valid moves which leads to position satisfying constraint

The first one first generates a position that satisfies the constraints to 100%. In this way it would be relatively easy to satisfy the constraint chosen in this thesis (number of pieces). The position should be generated reasonably with respect to the rules of chess, and it should certainly satisfy all legality properties that can be easily verified. For example, I certainly do not want to generate a position that has both kings right next to each other or has a pawn on the last rank (it should have been promoted to another piece). This would guarantee that most positions are likely to be legal. But as discussed in Chapter 2 in the *Problem of verifying valid chess position* section, proving this is not so easy. This is an NP-complete problem. The solution is to recursively search backwards for possible valid moves up to the initial position. You only need to find one such sequence leading to the initial position to prove that the position is valid. But proving the opposite is generally harder. In the worst case, one would need to backtrack through all possible move sequences to prove that the position is indeed **not** legal. One would have to deal with the possible problem that after several moves some backtrack sequences may return to the same position. It would probably want to favor back moves in which the number of pieces increases (i.e., close to the full number of pieces, as it is at the beginning). [21]

Conversely, the second approach selects successive **valid** moves, from the initial position, aiming to satisfy the constraint. Some metric is needed to evaluate the position in terms of constraint satisfaction. In each round, all valid moves are searched, the one with the best score is selected, and the process continues. Thus, the resulting generated position is implicitly valid. The disadvantage may be that if an inappropriate evaluation function is chosen, the constraint may not be satisfied at 100%.

Thus, the first approach makes it easy to satisfy the constraint, but very hard, often computationally impossible, to prove the validity of the position. The second approach, on the other hand, makes it very easy to prove the validity of a position, but harder to satisfy the constraint. Because of the complexity of creating a proof game, I decided to use the **second approach**. This way every generated position is automatically valid, because at each step the algorithm chooses one of the valid moves. In this way a sequence of valid moves is created from the initial position to the generated position.

4.2 Chess engine

When implementing the chess engine I decided to use a one-dimensional array board representation for position representation instead of the more efficient bitboard representation. The latter would be faster, but what matters is not so much the speed of generating a position that satisfies the constraints, but rather that such a position is generated at all.

4.3 Game tree search

There are 2 widely used algorithms for searching the chess game tree: Minimax and Monte Carlo. As mentioned in Section 2, modern successful systems have abandoned the traditional Minimax algorithm and use **Monte Carlo Tree Search**. I have decided to try Minimax instead, specifically its more widely used variant **Negamax**. The reason is that I assume the problem of satisfying the constraints is considerably easier than the problem of finding the best move that has the best chance of leading to mate.

By definition, the constraint satisfaction problem is more about player cooperation than competence. For this reason, it is not possible to use a Negamax algorithm that finds the best move for one player or, equivalently, the move that most harms the other. However, it can be easily modified to search the game tree in a cooperative game. The only thing I change is that I remove the **minus** evaluation when evaluating all possible moves.

Algorithm 3 Modified NegaMax Algorithm for cooperative chess

```
1: function CoopNegaMax(depth)
2:     if depth = 0 then
3:         return evaluate()
4:     max ← $-\infty$
5:     for all moves do
6:         score ← CoopNegaMax(depth − 1)          ▷ Note the missing minus sign
7:         if score > max then
8:             max ← score
9:     return max
```

When the algorithm runs recursively to the last move, it calls the *evaluate* function to evaluate the current chess position. But how do we do this so that the position that comes closest to satisfying the constraint is always chosen? This is evaluated by the trained neural network model.

4.4 Algorithm of generating a position satisfying given constraint

My position generation algorithm first stores all possible legal moves that can be made when choosing the next move. Then it calls a modified Negamax algorithm on each one. Next it assigns to each move its evaluation in terms of satisfying the given constraint. Consequently it selects the move with the best evaluation, executes that move and, if the constraint is not satisfied, moves on. Of course the generation algorithm starts from the initial chess position.

However, in this way the algorithm could run for a long time (making many moves) and it would still end up generating a position that does not satisfy the constraint. For this reason I have added a condition to the algorithm that checks when choosing the next move whether the number of half moves made has exceeded certain chosen limit. If so, the algorithm will terminate even though the constraint has not been satisfied, since I do not expect it to be satisfied any further. I set this limit to **101 half moves**.

Since the best **valid** move is always selected, a sequence of valid moves is generated from the initial position. So the generated position **is always valid**.

4.4.1 Randomness of generating positions

If the algorithm selects the move with the best evaluation, it will always be directed to the same position. Thus, such a deterministic approach of selecting moves when generating a position will not be able to generate different positions. However, the aim of this thesis is to generate potentially many positions, ideally as different as possible, for the chosen constraint. Therefore I have introduced randomness into the choice of moves. The algorithm first evaluates each move with a modified Negamax algorithm and puts it and its score into a sorted array. Then it chooses the lowest possible score that the selected move can have: $minScore = bestScore - criterium$. I set this criterion to 0.1. Finally it randomly chooses any move with a score in the interval $[minScore, bestScore]$. For example, if the best move has a score of 0.9, then any move with a score in the interval $[0.8, 0.9]$ can be chosen instead. In this way the algorithm is theoretically able to generate different positions that satisfy the constraint.

4.5 Neural Network

The neural network is therefore designed to evaluate the chess position to what extent it satisfies the given constraints. I decided to use a convolutional neural network. The architecture of the tested convolutional neural networks is described in chapter 6 Experiments.

4.5.1 Training dataset

To train the neural network, I did not directly use the dataset described in chapter 3, but created another one from it. Over all the games in this dataset, I found for each position the probabilities of satisfaction of the constraints. And I did this for these 17 constraints (the same ones used in the second experiment - described in chapter 6): $(1, 2, 2, 2, 7)$, $(1, 2, 2, 1, 8)$, $(1, 2, 1, 2, 8)$, $(0, 2, 2, 2, 8)$, $(1, 2, 2, 1, 7)$, $(1, 2, 1, 2, 7)$, $(0, 2, 2, 2, 7)$, $(1, 2, 1, 1, 7)$, $(1, 2, 2, 0, 7)$, $(1, 2, 0, 2, 7)$, $(1, 1, 2, 2, 8)$, $(1, 2, 2, 2, 6)$, $(1, 2, 1, 1, 8)$, $(1, 2, 2, 2, 5)$, $(1, 2, 1, 1, 6)$, $(1, 2, 2, 0, 6)$, $(1, 2, 0, 2, 6)$. I saved this as the data set for training the neural network. So for these 17 constraints, the number of data points in the dataset was: $9,803,755 * 17 = 166,663,835$.

In the actual training, I converted then each position into a **bitmap representation** whose tensor was 12x8x8. Along with the position, I needed to add constraints to the neural network input. Since the constraint has 5 numbers, I expressed it as 5x8x8, where all the numbers of the given 8x8 array were the same and set to the given number from the constraint. For example, if the constraint was (1, 0, 2, 1, 7), the first 8x8 was set to all ones, the second 8x8 was set to all zeros, ..., the last 8x8 in the tensor was set to all sevens. I added this constraint representation to the bitmap representation of the position, together the input tensor to the neural network was (12+5)x8x8 = **17x8x8**.

Chapter 5
Implementation

This chapter describes the structure of the project and the technologies chosen for development. Section 5.1 discusses the programming and scripting languages chosen and the motivation for their selection. Section 5.2 covers the libraries used in the project and Section 5.3 further describes the tools used. Finally, Section 5.4 gives a brief description of the structure of the project.

5.1 Programming and scripting languages

For programming, I chose the **C++** and **Python** programming languages. In C++ I implemented the whole chess engine, the neural network training and the experiments themselves. When a position satisfying a given constraint is generated sequentially, all possible moves are searched, evaluated and a move is selected accordingly. Due to the use of a modified Negamax, which recursively searches for the best move, the search can take a long time if the generation of all legal moves is slow. For this reason I chose C++ for the implementation. I used Python mainly in Jupyter Notebook for data analysis because of its ease of creating graphs and data analysis, where speed is no longer an issue.

For a more comfortable and easier implementation of the project, I used the **xonsh** scripting language, which allows to combine Python and Bash scripting languages.

5.2 Libraries

In total I used 3 libraries in my project: **PGNP**, **PyTorch**, **SDL2** and **Catch2**. The first PGNP library is used to parse **P**ortable **G**ame **N**otation `.pgn` files containing chess games (see chapter chapter 3 Analysis). I used this to parse the dataset so that I could further analyse the games and use them to train the neural network. The reason I used this library was that it allows you to parse the file in parts without loading the whole dataset into memory. This was necessary because the dataset was ∼9 GB and I did not know in advance how much data would be needed to train the neural network to give satisfactory results. Another reason was that it was several times faster than other libraries I had tried. The only drawback was that it parsed the moves in SAN format and not in algebraic notation (Nc3 vs c1c3), so I had to write my own SAN parser. I copied the source files of the library into a src file in the pgn subdirectory, as I was having trouble linking the library and it was faster than resolving the issue. The library can be downloaded from GitHub `https://github.com/manzerbredes/pgnp`.

The PyTorch library is used for tensor computation and deep neural networks. It is primarily designed for Python, but provides a C++ API, which I used since I chose C++ as my main language. I used the library to create and train neural networks. It also allows GPU training, which can be faster in some cases, and I took advantage of this. I chose the library because it met all my requirements and was easy to install and use. It is available from `https://pytorch.org/`.

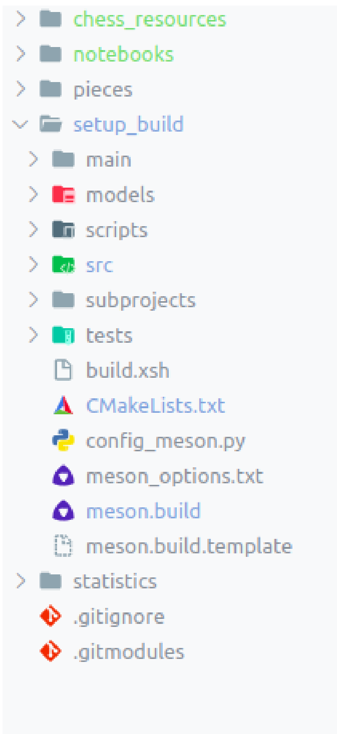
SDL2 stands for **S**imple **D**irect **M**media **L**ayer **2** and is a cross-platform library that provides low-level access to audio, keyboard, mouse, joystick and graphics hardware. I used it to graphically render the chessboard and pieces. I installed the library using the Linux tool apt, but it is also available on GitHub `https://github.com/libsdl-org/SDL`.

The last library I used is Catch2, which is also available on GitHub `https://github.com/catchorg/Catch2`. It is mainly used as a unit testing framework. I chose it because it allows for easy unit test programming and it provided everything I needed.

5.3 Tools

I used two tools to develop the project. The first is **CMake**, which I used to build the whole C++ part of the project. First I tried using another build system, Meson, but encountered complications when adding the PyTorch library, so I switched to the recommended build system, CMake. However, the project still contains Meson files, but they are no longer used. I also used the **git** tool for versioning and change tracking.

5.4 Project structure



```
> chess_resources
> notebooks
> pieces
v setup_build
  > main
  > models
  > scripts
  > src
  > subprojects
  > tests
    build.xsh
    CMakeLists.txt
    config_meson.py
    meson_options.txt
    meson.build
    meson.build.template
> statistics
  .gitignore
  .gitmodules
```

Figure 5.1 Structure of the project.

The structure of the project is influenced by the fact that the goal is only to create a project that experimentally verifies to what extent it is possible to generate chess positions that should satisfy the given constraints. It is not designed as a software application, but as a collection of code to implement the experiments.

The project is divided into several folders, as shown in the figure 5.1. The first folder **chess_resources** contains data for the analysis and training of neural networks. The folder **notebooks** contains Jupyter notebooks. To create the graphical chessboard I downloaded chess pieces from `https://commons.wikimedia.org/wiki/Category:PNG_chess_pieces/Standard_transparent` where they are freely available. They are stored in the folder **pieces**. The main folder for the whole project is **setup_build**, which contains all .cpp files, tests, scripts, saved models of trained neural networks and files for building the project. And the last folder **statistics** contains the saved results from experiments along with the generated positions. Below are the two files for the git tool.

5.4.1 Source Code

The project contains several programs whose .cpp files with the main() function are stored in the **setup_build/main** folder. The two most important ones are *Train_NN* and *GeneratePositions*. The former trains the neural network and saves the best model, the latter uses it to generate one or more positions.

The rest of the code is in the **setup_build/src** directory. The core of the chess engine is in the subdirectory **engineCore**. It contains the main chess components such as *Bitboard*, *Board* (which is a less efficient way of representing the board, but can be converted to *Bitboard* class), *Move* structure for representing chess moves and *Piece* class to represent the type of piece and its colour. It also contains the logic of the chess engine, i.e. the classes *BoardController*, which can make individual moves on the chess board, *Rules* to get pseudo-legal moves of individual pieces (not necessarily legal moves), and the main class *Game*, which controls the whole game and is able to return all possible legal moves (as opposed to pseudo-legal moves).

As for the training of neural networks, these are stored in the subfolder **neural**. It contains the classes *ConvNet* for the convolutional network, *ResNet* for the residual network (which didn't work), and also SimpleNet, which is a simple neural network with several fully connected layers.

The classes for loading and parsing the dataset are stored in the **dataset** subfolder. The C++ API of the PyTorch library requires a so-called data loader when training the neural network. This is an object that takes care of loading the dataset, e.g. in batches or even multithreaded. It takes as a parameter a dataset that can load data with the corresponding target values. This is done by my *ChessDataset_analysed* class. The project also contains other variants of this class, which were used for experimentation but did not give good enough results. Furthermore **dataset/parsing/***SanGame* and **dataset/parsing/***SanParser* can convert chess games from a PGN file (using the PGNP library) into games of the **dataset/parsing/***ParsedGame* class, which I can work with better and more efficiently.

The **Evaluation** subfolder contains the constraints defined for the whole project in the *Constraints* file, each of which uses the *Constraint* class. The *Evaluator* class is used to evaluate whether a position satisfies a constraint, using a trained neural network model. This is used by the **Search** class, which implements the minimax and alpha-beta pruning algorithms. The main class for finding the best move that satisfies the constraint is *MinimaxGame* in the **game** folder, which uses *Search* and thus indirectly *Evaluator*.

The *terminal* subdirectory is used for colored logging. It contains many files, but most of them are not used as they are not needed. Some general functionality, such as checking if a file exists, is in **common**. And the last subfolder **graphics** contains the *GraphicPieces* class for loading .png images of the pieces, the *Chessboard* class for rendering the chessboard with the pieces, and the *GraphicsGame* class for starting and displaying the game and making moves, which was mostly used for debugging.

5.4.2 Licence

Library licences are included in the digital attachment.

Chapter 6

Experiments

This chapter presents the experiments conducted and their results. Section 6.1 describes the first experiment, while section 6.2 covers the second. Section 6.3 then compares the results of these experiments.

6.1 Experiments

In total, I conducted 2 experiments. In the first, I trained the model for fewer constraints than in the second. In each case, several positions were generated, 300 in the first and 125 in the second, as it required much more computation time. The success of the generation was measured by the following formula:

$$satisfaction\_ratio = \frac{\#satisfied}{\#positions} * 100 \tag{6.1}$$

where $\#satisfied$ is the number of positions that fully satisfy the constraint, and $\#positions$ is the total number of positions generated. This means that $satisfied\_ratiois \in [0, 100]\%$. A position fully satisfies a constraint if it contains the same pieces of the same type as in the constraint. A way to determine this is to calculate the difference between the pieces on the board and the pieces in the constraint. The result will also be a set of five numbers. If the result is a zero vector, then the constraint is fulfilled . For example, suppose the chosen constraint is $(1, 0, 1, 1, 1)$ and White, in the current chess position I want to evaluate, has 1 queen, 1 rook, no bishop, 2 knights and 3 pawns (and of course a king). I convert this to the same five number format as the constraints, which would be $(1, 1, 0, 2, 3)$. After subtracting, $(1, 1, 0, 2, 3)-(1, 0, 1, 1, 1) = (0, 1, -1, 1, 2)$ remains. This means that the constraint is not satisfied because it contains some extra pieces and too few.

However, if the system generated, for example, 100 positions with $satisfaction\_ratio = 100\%$, but only two positions were unique, the system actually generated only 2 positions that satisfied the constraint. For this reason I also calculated the percentage of unique positions for each generation.

$$unique\_positions\_ratio = \frac{\#unique\_positions}{\#positions} * 100 \tag{6.2}$$

where $\#unique\_positions$ is the number of unique positions. A position is unique if none of the previously generated positions are the same. It also has the same range: $unique\_positions\_ratio \in [0, 100]\%$.

Thus, the success rate of generating positions that satisfy a given constraint is determined by these two measures. I have decided to round the results of these two measures to one decimal place, as this is sufficiently indicative of the success of the generation.

I also calculated how many average and median half moves it took to get to the **fully satisfied** position. Next, I computed the ***k*-PDR**, where $k \in \{1, 2, ..., 14\}$, which stands for *k*-**P**iece **D**ifference **R**atio:

$$k\text{-}PDR = \frac{\#k\text{-}positions}{\#positions} * 100 \qquad (6.3)$$

where $\#k\text{-}positions$ is the number of generated positions that have k pieces less or more than the constraint. So it gives the percentage of positions that are k pieces away from satisfying the constraint. For example, for $k = 1$, **1-PDR** gives the percentage of generated positions that do not satisfy the constraint, but have only one more or one less piece. In the tables below, for both experiments, I have listed ***k*-PDR** for $k \in 1, 2, 3, 4$, since any higher k is already considered too large an error, the position is far from satisfying the constraint. These numbers are also rounded to one decimal place, same as $unique\_positions\_ratio$ and $satisfaction\_ratio$.

Both experiments test whether it is possible to successfully generate constraint satisfying positions and what properties they have.

6.1.1 First experiment

The first experiment was designed for a smaller number of constraints to see if it is possible for the proposed algorithm and neural network to generate positions that satisfy the constraint at all. Another thing I wanted to check was whether these generated constraint satisfying positions were different or all the same. The constraints chosen for the first experiment were $(1, 2, 2, 2, 7)$, $(1, 2, 2, 1, 8)$, $(1, 2, 1, 2, 8)$, $(0, 2, 2, 2, 8)$, $(1, 2, 2, 1, 7)$, $(1, 2, 1, 2, 7)$ and $(0, 2, 2, 2, 7)$.

The convolutional neural network had two convolutional layers and two fully connected layers. The first convolutional layer had 17 input channels, 64 output channels, a 3x3 kernel and a stride set to 1. The second had 64 input channels, 32 output channels and the same kernel and stride. Both were terminated with a Rectified Linear Unit (ReLU) function. This was followed by a 1 max pooling layer, then the first fully connected layer of size 128x64 and the second of size 64x1. The network was terminated with a sigmoid activation function. The batch size was set to 128. The loss function chosen was binary cross entropy. The network was trained for 7 epochs and the best model has an average training error of 0.069, rounded to three decimal places.

The first experiment did not use the modified Negamax algorithm, but simply tried all moves to a depth one in each iteration. So no recursion was needed, it just tried each move, evaluated it and chose the best one.

The table 6.1 shows the results of the first experiment. The table contains the $satisfaction\_ratio$ and $unique\_position\_ratio$ values for each constraint. However, you can see from the table that most of the positions are unique.

Table 6.1 Results of the 1. experiment.

Constraint	Satisfaction ratio	Unique positions ratio
(1, 2, 2, 2, 7)	100%	96.7%
(1, 2, 2, 1, 8)	97.3%	87%
(1, 2, 1, 2, 8)	97.7%	75.3%
(0, 2, 2, 2, 8)	69%	99%
(1, 2, 2, 1, 7)	88.7%	99.7%
(1, 2, 1, 2, 7)	93%	100%
(0, 2, 2, 2, 7)	66%	100%

As shown in the table, the worst results are given by the constraints where the queen is missing, more precisely by the constraints $(0, 2, 2, 2, 8)$ and $(0, 2, 2, 2, 7)$. In both cases the ratio of satisfied to unsatisfied constraints is ∼70%. The other constraints do better, the best being $(1, 2, 2, 2, 7)$, where only one pawn has to be missing, which is 100% satisfied. As for the unique position ratio, most constraints are close to 100%, only the constraints $(1, 2, 2, 1, 8)$ and $(1, 2, 1, 2, 8)$ are much lower.

Table 6.2 Additional results of the 1. experiment.

Constraint	Mean steps	Median steps	1-PDR	2-PDR	3-PDR	4-PDR
(1, 2, 2, 2, 7)	9.1	9	-	-	-	-
(1, 2, 2, 1, 8)	14.8	9	0.3%	0.7%	0.7%	0.3%
(1, 2, 1, 2, 8)	9.5	7	0.3%	1%	0.3%	-
(0, 2, 2, 2, 8)	22.7	15	3.3%	3.3%	2.7%	0.3%
(1, 2, 2, 1, 7)	35.9	33	0.7%	3.7%	3.3%	2.3%
(1, 2, 1, 2, 7)	26.4	21	0.3%	2%	1.7%	1.7%
(0, 2, 2, 2, 7)	31.9	25	1.7%	4.3%	6.3%	7%

The table 6.2 shows more statistics about the generation results. For each constraint it shows how many average and median half moves (steps) it took to satisfy the constraint. The results show that when one piece is missing, the number of half moves is lower than when two pieces are missing. The only exception is when there is only one queen in the constraint, which has the highest number of half moves, both on average and median among the constraint where only 1 piece should be missing.

The table also shows the results of **1-PDR**, **2-PDR**, **3-PDR** and **4-PDR**. It can be seen that the worst results are again in the constraints $(0, 2, 2, 2, 7)$ and $(0, 2, 2, 2, 8)$ where a queen is missing. These had the lowest $satisfaction_ratio$, so it makes sense that they would also have the worst results.

Figure 6.1 shows the three generated positions for the constraints $(0, 2, 2, 2, 8)$, $(1, 2, 1, 2, 8)$ and $(1, 2, 2, 1, 7)$. The images are included as examples to give the reader an idea of what the generated position looks like. These are one of many possible positions that can be generated and they are only indicative.

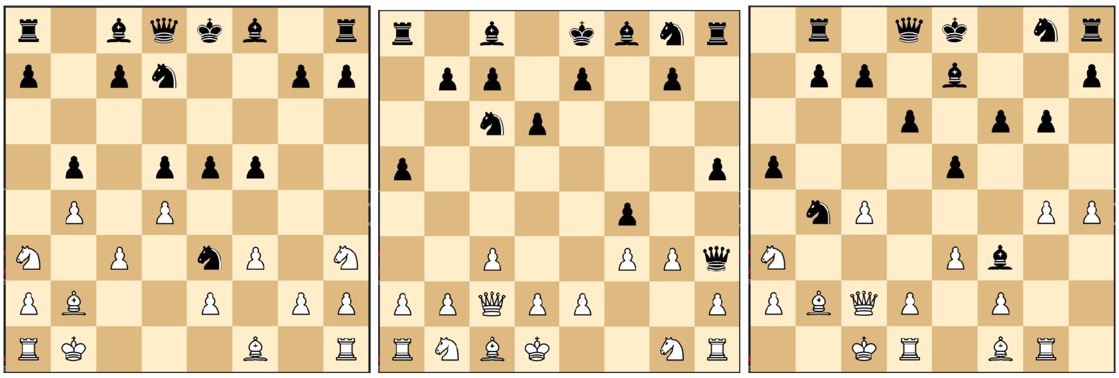


Figure 6.1 Examples of generated positions for constraints (0, 2, 2, 2, 8), (1, 2, 1, 2, 8), (1, 2, 2, 1, 7).

6.1.1.1 Conclusion of the experiment

The table 6.1 shows that the system is able to successfully generate positions that satisfy the constraints for 7 selected constraints. And this with a relatively high $satisfaction\_ratio$. In one case (constraint $(1, 2, 2, 2, 7)$) it even has a 100% constraint satisfaction success rate. The experiment also verified the ability of the system to generate different positions. Thus, the generated constraint satisfying positions are not all the same, but largely different.

6.1.2 Second experiment

The second experiment had more constraints than the first, with a total of 17 constraints, 10 of which were added. These were: $(1, 2, 2, 2, 7)$, $(1, 2, 2, 1, 8)$, $(1, 2, 1, 2, 8)$, $(0, 2, 2, 2, 8)$, $(1, 2, 2, 1, 7)$, $(1, 2, 1, 2, 7)$, $(0, 2, 2, 2, 7)$, $(1, 2, 1, 1, 7)$, $(1, 2, 2, 0, 7)$, $(1, 2, 0, 2, 7)$, $(1, 1, 2, 2, 8)$, $(1, 2, 2, 2, 6)$, $(1, 2, 1, 1, 8)$, $(1, 2, 2, 2, 5)$, $(1, 2, 1, 1, 6)$, $(1, 2, 2, 0, 6)$, $(1, 2, 0, 2, 6)$. This experiment was designed to test whether the neural network could be extended to a larger number of constraints and the system would still be able to generate constraint-satisfying positions. It also tests whether deeper recursion helps to improve the $satisfaction\_ratio$ for constraints where the queen shouldú be missing. These had worse results in the first experiment. As in the first experiment, it is also tested whether the generated successful positions are different or all the same.

The used neural network has been slightly modified. I added an extra convolutional layer, also terminated with a ReLU function, anticipating that more neurons would be needed to overcome more constraints. It had 64 input and output channels, a 3x3 kernel and a stride of 1. I also removed the max pooling layer. The batch size was the same, 128. The network was trained for a total of 8 epochs and the average training error of the best model was 0.073, which is also rounded to 3 decimal places as in the first experiment. It is slightly worse than in the first experiment.

This time a modified Negamax algorithm was used to find the best move. The depth chosen was only two, due to the time consuming computations in evaluating the positions. The difficulty is exponential compared to the first experiment where moves were only explored to depth one. The results are in the table 6.3.

Table 6.3 Results of the 2. experiment.

Constraint	Satisfaction ratio	Unique positions ratio
(1, 2, 2, 2, 7)	100%	78.4%
(1, 2, 2, 1, 8)	98.4%	87.2%
(1, 2, 1, 2, 8)	97.6%	96.8%
(0, 2, 2, 2, 8)	89.6%	98.4%
(1, 2, 2, 1, 7)	97.6%	99.2%
(1, 2, 1, 2, 7)	99.2%	100%
(0, 2, 2, 2, 7)	95.2%	100%
(1, 2, 1, 1, 7)	90.4%	100%
(1, 2, 2, 0, 7)	95.2%	100%
(1, 2, 0, 2, 7)	49.6%	100%
(1, 1, 2, 2, 8)	85.6%	100%
(1, 2, 2, 2, 6)	100%	97.6%
(1, 2, 1, 1, 8)	92%	100%
(1, 2, 2, 2, 5)	97.6%	100%
(1, 2, 1, 1, 6)	82.4%	100%
(1, 2, 2, 0, 6)	87.2%	100%
(1, 2, 0, 2, 6)	57.6%	100%

The results show that most constraints are satisfied with a large $satisfaction\_ratio$, the constraints $(1, 2, 2, 2, 7)$ and $(1, 2, 2, 2, 6)$ are even satisfied 100% of the time. Only two constraints give significantly worse results: $(1, 2, 0, 2, 7)$ and $(1, 2, 0, 2, 6)$. What they have in common is that they both miss two bishops. However, many of the constraints had all generated positions different ($unique\_positions\_ratio = 100\%$). The table 6.4 shows after how many moves, on average and median, were these constraint satisfied. Obviously, the more moves it took to satisfy the constraint, the higher the $unique\_positions\_ratio$ it has. This makes sense since there are many more possible sequences of moves.

The table 6.4 also shows the errors as expressed by **1-PDR**, **2-PDR**, **3-PDR** and **4-PDR**.

Obviously the highest values are for the constraints $(1, 2, 0, 2, 7)$ and $(1, 2, 0, 2, 6)$, which were satisfied about 50% of the time. The highest values are for **2-PDR**. For example, up to 18.4% of the generated positions for the constraint $(1, 2, 0, 2, 7)$ have two more or fewer pieces. Or for a constraint where 1 rook is missing, it is 8.8%.

Table 6.4 Additional results of the 2. experiment.

Constraint	Mean steps	Median steps	1-PDR	2-PDR	3-PDR	4-PDR
(1, 2, 2, 2, 7)	5.6	5	-	-	-	-
(1, 2, 2, 1, 8)	14.1	11	-	1.6%	-	-
(1, 2, 1, 2, 8)	14.4	9	1.6%	0.8%	-	-
(0, 2, 2, 2, 8)	13.3	11	0.8%	-	-	-
(1, 2, 2, 1, 7)	14.8	11	-	2.4%	-	-
(1, 2, 1, 2, 7)	15.7	11	0.8%	-	-	-
(0, 2, 2, 2, 7)	17.5	13	0.8%	0.8%	1.6%	0.8%
(1, 2, 1, 1, 7)	32.9	25	1.6%	5.6%	1.6%	0.8%
(1, 2, 2, 0, 7)	26.9	21	-	-	-	0.8%
(1, 2, 0, 2, 7)	43.5	39	2.4%	18.4%	4%	2.4%
(1, 1, 2, 2, 8)	33.9	25	1.6%	8.8%	-	-
(1, 2, 2, 2, 6)	8.9	9	-	-	-	-
(1, 2, 1, 1, 8)	25.3	21	1.6%	4%	1.6%	-
(1, 2, 2, 2, 5)	24.7	19	0.8%	0.8%	-	-
(1, 2, 1, 1, 6)	43.7	43	0.8%	8.8%	7.2%	0.8%
(1, 2, 2, 0, 6)	39.8	35	2.4%	7.2%	-	3.2%
(1, 2, 0, 2, 6)	47.9	45	4%	13.6%	6.4%	1.6%

Figure 6.2 shows again the three generated positions for the constraints $(0, 2, 2, 2, 7)$, $(1, 2, 1, 1, 7)$, $(1, 2, 2, 0, 7)$. As with the first experiment, the positions in the pictures are just one of many possible positions. They are included here to give the reader an idea of what the generated positions look like.

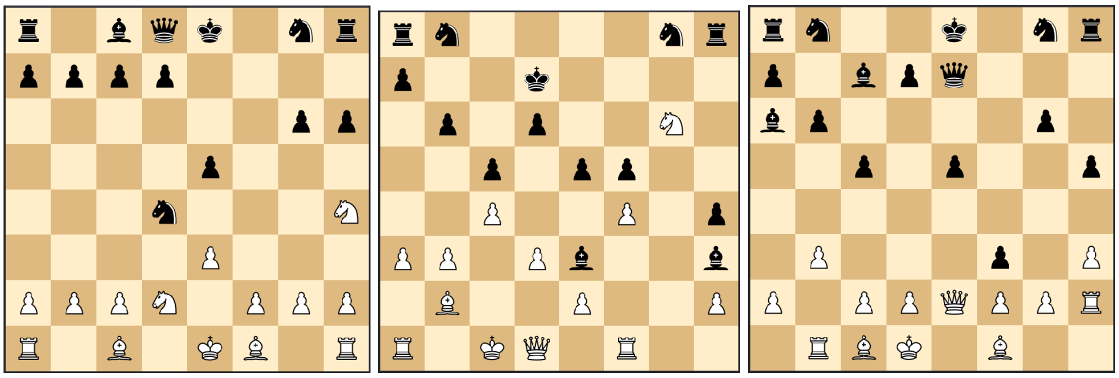


Figure 6.2 Examples of generated positions for constraints (0, 2, 2, 2, 7), (1, 2, 1, 1, 7), (1, 2, 2, 0, 7).

6.1.2.1 Conclusion of the experiment

The second experiment produced better results, suggesting that deeper recursion aids in solving constraint satisfaction problems. The biggest improvement can be seen in the constraints where the queen should be missing. These had worse results in the first experiment, but are much better in the second, with a 20% increase in satisfaction for $(0, 2, 2, 2, 8)$ and 29% for $(0, 2, 2, 2, 7)$.

This experiment showed that it is possible to extend the neural network to a larger number of constraints so that the generation algorithm is still able to produce constraint-satisfying positions. The experiment also showed that even with a larger number of constraints, the generated positions are still more likely to be different and only a small percentage of positions are the same.

6.2 Conclusion

The chapter presented two experiments and proved that it is possible to generate positions that satisfy the constraints in the described way and at the same time are mostly not the same. The first experiment extended the neural network by one convolutional layer and increased the recursion depth of the generation algorithm by 1, to a total depth of 2. It showed that it is possible to successfully extend the described algorithm and neural network to a larger number of constraints. In Experiment 2, it was successfully extended from 7 to 17. Increasing the recursion depth also helped to improve the $satisfaction\_ratio$ values, i.e. the generation success rate, for constraints that performed worse in the first experiment. This suggests that a higher recursion depth helps to successfully find a position that satisfies the given constraint.

Chapter 7

Conclusion

The aim of this thesis was to develop a system capable of generating valid chess positions that satisfy given constraint. The type of constraint was chosen as number and type of pieces which are in the resulting position. In total there can be 486 combinations of the chosen constraint. A limited number of combinations of this constraint type were chosen to fit the scope of the bachelor thesis, namely 17 constraints.

Two possible approaches to solving the constraint were explained. Since it is difficult to prove that a given position is valid, the **gradual position generation** approach was chosen. The algorithm starts from the initial position and sequentially selects the moves that are most likely to satisfy the constraint. It stops when it encounters a position that satisfies the constraint, or when the number of moves exceeds a specified maximum. Since a valid move is chosen at each step, the generated position is **always valid**. This approach has been implemented using a modified Negamax algorithm and a neural network. The Negamax algorithm is modified for black and white cooperation, since the constraint satisfaction problem is non-competitive. It recursively searches the game tree to find the best move, and the neural network evaluates each position in terms of constraint satisfaction. To ensure that the algorithm does not always generate the same position, the best move is not necessarily chosen, but a move with a slightly lower score may also be chosen.

A total of two experiments were carried out to test the implemented algorithm. The experiments tested whether positions satisfying the constraint could be successfully generated using the proposed algorithm. The first experiment was for 7 constraints only and the recursion depth for finding the best move was set to 1. In total 300 positions were generated for each constraint. One constraint had a success rate as high as 100%, most of the others were around 95%. In the second experiment, the number of constraints was successfully increased to 17, and 125 positions were generated for each. It showed that a higher recursion depth in the search for the best move leads to better results. In both experiments, most of the positions generated were different, which was good result.

Future work would be to extend to more constraint combinations and to improve the generation algorithm to have a higher constraint satisfaction rate. It would also be interesting to investigate whether player level has an effect on constraint satisfaction. For example, whether a professional player satisfies the constraints in fewer moves than a beginner, or whether the positions look different. In any case, the results show that the described approach to generating constraint satisfying positions has potential.

Bibliography

1. *Branching Factor - Chessprogramming wiki* [online]. [N.d.]. [visited on 2024-02-15]. Available from: `https://www.chessprogramming.org/Branching_Factor`.
2. INC., OEIS Foundation. A048987. *On-Line Encyclopedia of Integer Sequences*. 2024. Available also from: `https://oeis.org/A048987`.
3. *Shannon number* [online]. 2024. [visited on 2024-02-15]. Available from: `https://en.wikipedia.org/w/index.php?title=Shannon_number&oldid=1198284115`. Page Version ID: 1198284115.
4. *Ceriani-Frolkin Theme* [online]. [N.d.]. [visited on 2024-02-15]. Available from: `https://www.janko.at/Retros/Glossary/CerianiFrolkin.htm`.
5. HYATT, Robert. Chess program board representations. *University of Alabama at Birmingham*. 2004.
6. *Review of different board representations in computer chess.* [online]. 2022. [visited on 2024-02-15]. Available from: `https://lichess.org/@/likeawizard/blog/review-of-different-board-representations-in-computer-chess/S9eQCAWa`.
7. ADEL'SON-VEL'SKII, Georgii Maksimovich; ARLAZAROV, Vladimir L'vovich; BITMAN, AR; ZHIVOTOVSKII, AA; USKOV, Anatolii Vasil'evich. Programming a computer to play chess. *Russian Mathematical Surveys*. 1970, vol. 25, no. 2, p. 221.
8. BIJL, Pieter; TIET, AP. Exploring modern chess engine architectures. *Victoria University, Melbourne*. 2021.
9. WIKIPEDIA CONTRIBUTORS. *Monte Carlo tree search — Wikipedia, The Free Encyclopedia*. 2024. Available also from: `https://en.wikipedia.org/w/index.php?title=Monte_Carlo_tree_search&oldid=1200466356`. [Online; accessed 15-February-2024].
10. MATSUGU, Masakazu; MORI, Katsuhiko; MITARI, Yusuke; KANEDA, Yuji. Subject independent facial expression recognition with robust face detection using a convolutional neural network. *Neural Networks*. 2003, vol. 16, no. 5-6, pp. 555–559.
11. *What are Convolutional Neural Networks? — IBM* [online]. [N.d.]. [visited on 2024-02-15]. Available from: `https://www.ibm.com/topics/convolutional-neural-networks`.
12. HE, Kaiming; ZHANG, Xiangyu; REN, Shaoqing; SUN, Jian. Deep residual learning for image recognition. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.
13. KLEIN, Dominik. Neural Networks for Chess. *arXiv preprint arXiv:2209.01506*. 2022.
14. DREŻEWSKI, Rafał; WATOR, Grzegorz. Chess as sequential data in a chess match outcome prediction using deep learning with various chessboard representations. *Procedia Computer Science*. 2021, vol. 192, pp. 1760–1769.

15. SABATELLI, Matthia; BIDOIA, Francesco; CODREANU, Valeriu; WIERING, Marco A. Learning to Evaluate Chess Positions with Deep Neural Networks and Limited Lookahead. In: *ICPRAM*. 2018, pp. 276–283.
16. OSHRI, Barak; KHANDWALA, Nishith. Predicting moves in chess using convolutional neural networks. *ConvChess. pdf*. 2016.
17. EDWARDS, Steven J. Portable game notation specification and implementation guide. *Retrieved April*. 1994, vol. 4, p. 2011.
18. *Chess Notation for Beginners - Chessable Blog* [online]. 2021. [visited on 2024-02-15]. Available from: `https://www.chessable.com/blog/chess-notation-for-beginners/`.
19. *Frequently Asked Questions • lichess.org* [online]. [N.d.]. [visited on 2024-02-15]. Available from: `https://lichess.org/faq#ratings`.
20. BRONSTEIN, David. *The Game of Chess*. Dover Publications, 2004.
21. Available also from: `https://github.com/peterosterlund2/texel/blob/master/doc/proofgame.md`.