FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Antonín Otmar

Generalized chess engine

Department of software and computer science education

Supervisor of the bachelor thesis: Klára Pešková

Study programme: Computer science

Study branch: Software engineering

Prague 2024

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I want to thank my supervisor Klára Pešková for all the help I needed while writing this thesis.

Title: Generalized chess engine

Author: Antonín Otmar

Department: Department of software and computer science education

Supervisor: Klára Pešková, Department of software and computer science education

Abstract:

Background: While chess playing software is currently leagues above how the best human players play the game, engines playing less known variants of chess (known as *fairy chess*) are much less explored.

Objectives: Come up with an exact definition of fairy chess, then develop a general engine capable of playing any such game and evaluate which strategical approaches taken from chess generalize the best onto its variations.

Methods: We developed a minimax based engine capable of playing multiple games based on its configuration with a modular evaluation function, built multiple versions of this engine differing only in the static evaluation used and let them play chess variants against each other to see which was the strongest. First different approaches to evaluating pieces to count material were tried against each other, then different approaches to evaluating positions atop the way of counting material that turned out the best.

Results: Evaluator that estimated the values of pieces by their mobility won the most games against its counterparts (160 of 280 as opposed to 133.5 and 126.5 of the two other approaches tried). Then the evaluator evaluating positions by counting available moves won the most out of all its possible enhancements with 91.5 of 120 games.

Conclusions: The best approach to play any chess variant if we have to use one strategy for all of them we found was based on mobility of pieces, using a combination of both how much mobility they're likely to have in a random position and how much mobility they actually have in the particular position that is evaluated.

Keywords: fairy chess, chess engine, chess variants, minimax

Contents

Introduction

Over the course of chess's long history, starting with the Indian Chaturanga, the game has evolved into many different variants united by a few core concepts, in part due to a natural evolution of the rules and in part due to the work of chess enthusiasts creating new variants for the joy of it - those variants eventually becoming known as *fairy chess*. Later, chess's more popular variants became a challenge for computer programmers – first the goal was to make computers beat the best of human players and once that was accomplished, the challenge changed to creating the best chess engines for games of computers against computers.

This work focuses on generalizing the principles of chess-playing software onto a larger variety of chess-like games. In the beginning I will look at known chess-like or fairy chess games and identify some of their common characteristics to define what I will consider a chess variant for the purpose of this work.

The next goal of this work is to create an engine capable of reading the rules for a variant of chess and then playing the variant against a human player or another engine. Most importantly the engine's purpose is experimentation with fairy chess variants, so its implementation has to have enough modularity to allow building different engines applying different strategies (given them as notions used by the evaluation function) to the games played. The last goal of this work then is to experimentally test those engines in games against each other and infer which concepts from chess programming generalize best into similar games.

The first chapter will focus on describing variants of fairy chess. The second chapter will talk about existing works related to the topic. The third chapter will define fairy chess pieces and therefore fairy chess for the purpose of this work. The fourth chapter will describe the workings of the engine and the fifth chapter will go into more depth on its technical implementation. In the sixth chapter we will describe the particular engines we actually built and in the seventh chapters we will present the results of them playing against each other.

1. Fairy chess games

Before making a generalized chess engine, it is necessary to describe what generalized chess is. Historically, there have been many board games similar to chess (from here referred to as *FIDE chess* whether due to a coincidence, sharing its ancestor with *FIDE chess* i.e. being derived from *chaturanga* – or explicitly being invented as a variation to make chess more interesting. In this chapter I will attempt to describe the commonalities of chess-like games.

1.1 What is Fairy Chess

Fairy chess is a loosely defined term used for games that are similar to *FIDE chess* but differ in some rules. For the needs of this thesis, I will use the phrase to also mean *FIDE chess*, *shogi*, *xianqgi* and other games that are for historical reasons not called fairy chess despite fitting this definition. Commonalities of fairy chess variants include a playing board of discrete cells (typically square) and being played on a turn by turn basis with moves consisting mainly of moving pieces from place to place (how exactly they may move is further discussed in Chapter 3). In the following sections I will describe the most common modifications.

1.1.1 Different starting positions

The most popular example of chess with different starting position is *chess960* – a variant using the standard set of pieces with all the non-pawn pieces' ranks shuffled randomly (requiring only that bishops are of opposite colours and rooks are on opposing sides of the king) [4]. A similar variant *shuffle chess* relaxes those restrictions and includes games where that may not be the case – both rooks may end up on one side of the king or both bishops may be of the same color.

Variants not included among those are ones where some pieces start on rows other than the first and second/seventh and eight, like *advance chess* 1.1 or *corridor chess* 1.2. Furthermore, there are games with different sets of standard pieces – Pritchard mentions *double knight chess* 1.3 where bishops are replaced with knights and *knight supreme chess* 1.4 where both bishops and rooks are replaced by knights [13].

Figure 1.1: Advance chess starting position

Figure 1.2: Corridor chess starting position

Figure 1.3: Double knight chess starting position

Figure 1.4: Knight supreme starting position

Lastly Pritchard describes variants that start with an empty board and pieces are placed on it by players in an alternating fashion for the first sixteen moves until they are all on the board and the game begins.

1.1.2 Different pieces

Some fairy chess variants introduce new pieces, either defined as combinations of orthodox pieces (amazon as the combination of the queen and the knight or the chancellor as the combination of bishop and knight) or introducing patterns that are not present in the orthodox pieces at all.

Movement patterns of new pieces are usually defined by a list of atomic patterns (see Betza's notation 12) where that is possible. More elaborate ways of combining the atomic patterns have been developed to describe complicated pieces such as Betza's crooked bishop [2] or Aronson's rhinos [1].

In many cases however the atomic patterns are not descriptive enough – notably if the piece does not capture in the standard way of moving to a place where an opposing piece stood or if the validity of its moves depends on other factors than pieces standing in its way. Examples include pieces that have to leap over others to make a move such as the grasshopper or the xiangqi cannon as well as pieces whose movement patterns are influenced by where on the board they are such as the xiangqi soldier or the edgehog [13].

Furthermore, there are variants in which each player has different set of pieces to play with, most notable one of them being the 64-variant (8 possible choices of armies for either side) *Betza's Chess with different armies* [3].

Of special note are games where the pawn behaves noticeably different. Being the least valuable piece, i.e. one not worth being traded for anything other than a pawn, a well-protected pawn structure in *FIDE chess*is difficult to alter and as such in a way defines the character of the position more so than the positions of more mobile pieces. As Kramnik has written about the sideways pawns variant: "Even after having looked at how AlphaZero plays Pawnside chess, the principles of play remain somewhat mysterious—it is not entirely clear what each side should aim for. The patterns are very different, and this makes many moves visually appear very strange, as they would be mistakes in classical chess." [15]

1.1.3 Different game objective

While the most common goal in chess-like games is to capture (or checkmate, as the game ends right before the inevitable capture is to take place) the opponent's king, this needn't always be the case.

Very popular variants include loser's chess (which is also interesting for its mechanic of capturing pieces being compulsory, i.e. all non-capturing moves being illegal if at least one capture is possible) with the goal of losing all your pieces or the triple check chess where the player who manages to check opponent's king three times is victorious.

Another common goal is to capture all of the opposing pieces – the king typically has no special value in such variants. An interesting variant is *extinction chess*, where the goal is to capture all pieces of a particular kind – here the king is not a royal piece (player doesn't have to evade checks and it is even possible to

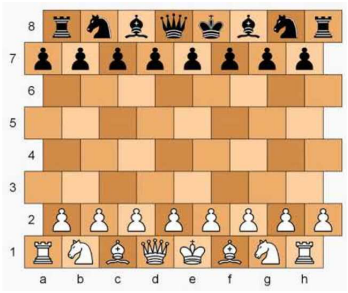

Figure 1.5: Masonic chess

promote a pawn to a king) but it's unimpressive mobility along with being only one of its kind at the start make it a natural target.

1.1.4 Board shapes

Aside from the boards being bigger – either rectangular as in *superchess* or *capablanca chess* or larger squares as in *shogi* – there are variants utilizing more alien topologies. Besides the cylindrical boards where pieces are allowed to move from one side to the other, Pritchard mentions a *Möbius* variant played similarly to *cylindrical chess* but with the pieces' position being mirorred whenever they pass from column a to h or vice versa over what would be the edge of the board in FIDE chess – this variant has challenges regarding pawn orientation when they get mirrored.

A natural extension of the cylinder variant is a torus variant, connecting the bottom and the top edges as well as the left and the right edge. This is of course unplayable with the orthodox starting position, as black would immediately be in check – the solution given by Philip Cohen is to start the game with board empty and let the players place pieces in an alternating fashion, each restricted to their first half of the board [13].

There are also variants that don't restrict themselves to square cells, commonly using hexagonal or triangular cells – those games of course have to use a piece set completely different from pieces used on a square board. Games retaining some semblance of square cells but rearranging them in a way that makes the board's properties very different include *masonic chess* 1.5 and *circular chess* 1.6.

Lastly, there are games where instead of one 2D dimensional board, the pieces move over a set of boards stacked vertically, making the game three-dimensional. *Raumschach* [5] is a variant attempting to translate the movement patterns of 2D pieces into 3D – rook and knight are generalized into 3D easily, but the bishop is split into two pieces – the bishop moving over planar diagonals and the unicorn, moving over spatial diagonals. Other 3D games, such as *peruvian army chess* separate pieces into two groups – one restricted to the bottom board, one restricted to the top.

1.1.5 More alien game mechanics

Many games were invented that are similar to chess yet contain rules that don't fit well into any of the above categories. Examples include games with hidden

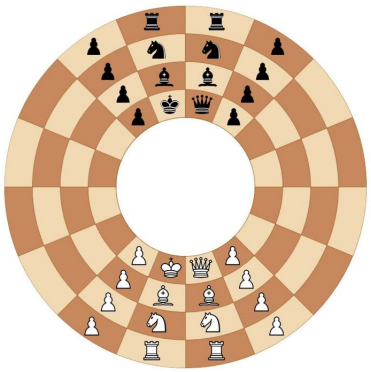

Figure 1.6: Circular chess

information such as *chess in disguise* where each player assembles their own starting position and doesn't reveal which piece is which until moving it.

Other games employ move mechanics that are not played by any piece in particular, such as putting pieces into play from hand in *shogi* or *crazyhouse*, pivoting rows and columns in *pivot chess* or the central squares in *twist chess*, or mirroring pieces in mirror chess.

Not even the rule of players alternating after each move is always abided by – *Marseilles chess* lets a player play two moves before letting the opponent answer with several variants specifying which kinds of moves can be played in succession (including dividing the board into halves, dividing pieces by which side they started at, allowing only a non-pawn and a pawn move in succession...)

1.2 Analysis of requirements – Fairy Chess for the purpose of this work

The goal I have set for this work is to create an engine that would be capable of playing fairy chess. However, due to how wide the space of such games is, it'd be unreasonable to attempt to support all of them – especially considering that there isn't even a truly exhaustive definition of a fairy chess game.

1.2.1 Supported chess games

Based on the games mentioned above and while trying to make programming such engine feasible, we have decided the engine will support chess games that satisfy the following requirements:

- Players alternate move by move.
- Starting position is given, i.e. it is not created by players after the game starts.
- Every possible move belongs to a particular piece that plays it.
- Board is composed of square cells arranged in a square or rectangular grid.

- The goal of the game is checkmating the enemy king (stalemate can be redefined to be another victory condition for the side whose opponent has no moves as it is in *xiangqi*).

2. Related works

In this chapter I will describe existing works related to the topics of my thesis, namely systems used to describe generalized chess pieces as well as existing engines capable of playing multiple chess variations if given some description of its rules.

2.1 Generalized chess notations

There are several systems to describe fairy chess pieces based on how they can move. I will describe how they work and what – if any – are their limitations.

2.1.1 Betza's notation

Betza's notation is a system developed by Ralph Betza [8] that divides chess moves into nine atomic patterns plus two extra patterns for staying in place and moving anywhere.

The patterns can – except for U – be written as tuples of *(a,b)* where a is the number of squares moved in a given direction and b is the number of squares in a direction orthogonal to it.

- W – (1,0)
- F – (1,1)
- D – (2,0)
- H – (3,0)
- N – (2,1)
- A – (2,2)
- C – (3,1)
- Z – (3,2)
- G – (3,3)
- O – (0,0) – piece that can stay in place as a valid move
- U – universal leaper, can move to any square on the board

A piece in Betza's notation is described as a set of atomic pieces with modifiers for each atom.

An atom has to specify the number of moves in one direction they can do in repetition as long as they don't run into any obstacles. It can be further modified by specifying:

- which directions the atomic piece can move in

- whether the given atomic piece is only applicable for captures or only for moving onto unoccupied places.
- whether the atomic piece can jump over other pieces (it can also be specified that the piece has to jump over another to move at all)
- that the atomic piece treats the board as a cylinder
- that the atomic piece must change the direction of its movement every step – it is also possible to define a piece that alternates between several movement patterns

The drawback of this notation with regards to standard chess is its inability to describe the following:

- moves with conditions regarding moves played in the past, such as castling, en passant capture or pawn's double jump
- moves summoning new pieces on the board such as pawn promotion
- double moves such as castling
- moves capturing a piece not on the moving piece's destination square such as en passant capture

2.1.2 XBetza notation

XBetza notation [9] is an evolution of Betza's notation that is used by the GNU XBoard chess interface. It includes additional features that allow the description of standard chess:

- modifier 'i' to specify an atom only applies if the piece hasn't yet moved
- modifier 'e' to specify a piece can take another piece if that one has just passed a square that it could take, reserved to pawns
- castling is reserved for the king, but it doesn't necessarily have to be with a rook

furthermore it includes features that aren't relevant to standard chess, but allow more expressive definitions of pieces:

- modifier 'a' to allow connecting different atoms into a double move
- annotation '@' for putting new pieces on the board from hand – allowing games like bughose and crazyhouse
- modifier 't' to specify that the atom can't be used to check

XBetza solves most drawbacks of Betza's notation with regards to standard chess, but it does so in a way that is self-admittedly ad hoc invented for standard chess – it still doesn't have a way to describe a move of two pieces that isn't castling or an indirect capture that isn't en passant. [9]

2.1.3 Betza 2.0

Betza 2.0 [9] is a further evolution of XBetza, that allows chaining arbitrary number of atomic moves into a large move done at once.

As such it allows quite complex description of side effects as the move can be arbitrarily long and on each step the piece can move somewhere (and thus assert a condition to play this move), capture a piece or drop a previously captured piece. It can even check for the piece's absolute position on the board by including in the path a place that'd be outside the chessboard in case the piece is in a place where this movement isn't allowed.

2.1.4 David Parlett's notation

A notation for fairy pieces developed by David Parlett [12]. The system divides movement patterns into 5 directions – forward, diagonally forward, sideways, backward and diagonally backward. More complex moves can be created by chaining atomic moves using the . (dot) operator or multiplying a direction (possibly by n, meaning unlimited distance in a given direction).

A suffix & can be added if a compound movement can be done repeatedly in the same direction.

Unlike Betza's notation, the notation assumes all pieces to be symmetrical over the forward/backward axis.

A chess piece in Parlett's notation is a string of available moves separated by commas.

2.1.5 Extensions to Parlett's notation

Parlett's notation has been extended with conditional modifiers to tell:

- that the move can only be played if the piece has not yet moved
- that the move can only be used for capture
- that the move can only be used for non-capture
- that the move captures pieces by moving past them as in checkers
- whether the piece can be blocked by obstacles in its path (unlike in Betza notation, the path is always well-defined)

2.1.6 Fairy Max notation

This notation was developed for Fairy Max (chess engine) ini files, as an extension of Micro Max's internal representation of standard chess pieces. A move is defined by the relative position of a tile and a move descriptor in the format of five hexadecimal digits, where the last two digits are a descriptor:

- whether the move can be used for capture
- whether the move can be used for non-capture
- whether the move can be used repeatedly in one turn

- whether the move can hop over an nonempty square without capturing
- if the penultimate digit is nonzero, it is a magic number with an orthodox chess specific meaning – pawn double step, castling

FairyMax pieces with repeatable move can alternate between two different atomic patterns, the first two digits being the toggle for the target square position and the third digit being the toggle for the last digit.

The notation uses absolute orientation of the board and for that reason any asymmetric piece has to be defined twice – once for black and once for white. Null move isn't definable.

2.2 Generalized chess engines

Generalized chess engines are programs capable of playing multiple chess variants, usually using a configuration file of some sort that they can interpret as rules of a chess variant.

2.2.1 Fairy Max

The Fairy Max chess engine is an engine derived from the micro-Max chess engine extended to be able to play with user defined pieces. It understands pieces given in an engine-specific notation.

Fairy Max is compatible with the GNU XBoard/WinBoard interface, with its definable pieces being a subset of all those definable by XBetza notation. The engine evaluation method requires the user to input the value of every custom piece for the engine to use it.

All chess variants playable with Fairy Max have to have the winning condition be the capture of all opponent's royal pieces (in orthodox chess, there's only one per side, but Fairy Max supports several)[11].

2.2.2 Fairy Stockfish

Fairy stockfish is an engine derived from the Stockfish chess engine. It allows definitions of custom pieces in Betza notation with support for using orthodox chess pieces as shorthands for their Betza definitions.

In addition to Betza notation, the pieces can be further specified by defining their mobility regions, ability to promote or to take/be taken en passant in configuration flag. The engine has a very rich support for changing game rules that aren't dependent on any particular piece, such as:

- non standard victory/draw/loss conditions – including capture the flag type games, extintion chess, toggling draw by repetition or victory by achieving a special position
- promotion areas and promotable pieces
- castling ranks
- dropping pieces and rules specifying how to obtain pieces to drop

- and many more settings typically borrowed from traditional chesslike games but allowed to be used in non-traditional combinations

The engine is based on Stockfish's neural network architecture modified to represent positions in a way that allows for richer definitions of the game and its state. Multiple Fairy Stockfish neural networks trained for particular games can be downloaded. The engine also has a handcrafted evaluation function that it will use if no neural network is available.

2.3 XBoard chess interface

The XBoard interface, originally developed by Tim Mann and currently maintained by the GNU project is a graphical interface for chess engines supporting FIDE chess as well as chess variants.

2.3.1 XBoard communication protocol

When an engine is selected to run under XBoard, it's started in a way that XBoard's commands become its standard input and the program's standard output is piped to XBoard. The interface sends messages such as:

- `xboard` – initial message sent on start
- `new` – reset the board to the default position
- `variant NAME` – play the following variant
- `quit` – the chess engine should exit
- `force` – engine should play neither color, only to be used for checking legality of moves
- `go` – engine should play the color that is to move
- `white/black` – set white/black to move – engine should play the opposite side than was sent
- `st TIME` – set time per move, time is a number of seconds
- `level MOVES BASE INC` – set time control with `BASE` amount of time at start, an increment of `INC` (in seconds) per move, incrementing by `BASE` after each `MOVE` moves
- `sd DEPTH` – cap the depth of the engine's search tree at `DEPTH`
- `time N` – set engine's time to `N` (in centiseconds)
- `otim N` – set opposite engine to N (in centiseconds)
- `MOVE` – opponent played move – given as `[startSquare][endSquare]` in algebraic chess notation, e.g. a1h8; pawn promotion as `[startSquare][endSquare][pieceString]`, e.g. a7a8q;
  castling as e1g1/e1c1/e8g8/e8c8; null move as @@@@

- `draw` – opponent offers a draw – if engine replies with offer draw, it is accepted and game ends
- `result RESULT` – end of game, result takes values of 1-0, 0-1, 1/2-1/2 or * (unfinished)
- `setboard FEN` – set board to the given position
- `edit` – put engine into edit mode where it accepts editing commands – `[pieceString][square]`: place piece on square; `x[square]`: remove piece; `#`: clear board; `c`: change color of pieces places; `.`: leave edit mode
- `hint` – suggest a move to the user
- `bk` – send some text to the user (the documentation doesn't specify what text)
- `undo` – revert the last move
- `hard` – turn on engine thinking in opponent's time
- `easy` – turn off pondering during opponent's time
- `post/nopost` – turn on/off pondering input

There are more commands that XBoard can send, relating to things like player identification if xboard is running as an internet chess server, management of resources (how much memory can the engine use, etc.) or playing the four–player variant bughouse. Full documentation can be found here [14].

The engine sends messages back to the interface as:

- `Illegal move: MOVE` – inform the interface that what the opponent played was illegal
- `Error (errortype): COMMAND` – inform the interface that the engine can't understand/doesn't support a command it received
- `move MOVE`: play a move, given in the same format as XBoard sends it to the engine
- `RESULT Reason` – set the result of the game – the result has to be validated by the opponent to be valid
- `resign` – resign from the game
- `offer draw` – offer draw to the opponent, XBoard will send them a draw command
- `telluser/tellusererror MESSAGE` – display a message dialog
- `askuser REPTAG MESSAGE` – show user a dialog with a textfield – if user replies, xboard will return `REPTAG REPLY` command

3. Fairy chess pieces

In this chapter I will describe what I consider a fairy chess piece for the purpose of this work, i.e. what pieces will the engine developed be able to play with. Then I will introduce a definition general enough to capture most of pieces used in existing fairy chess variants and provide a notation for describing pieces satisfying that definition.

Unless specified otherwise, the chess variants mentioned in this chapter are taken from David Pritchard's Classified Encyclopedia of Chess Variants [13].

3.1 Description of chess pieces

A chess piece is in essence described by how it can move – each move is defined by its consequences and by the conditions required to make the move legal.

One consequence is almost always moving the piece from one place on the board, possibly on the square where another piece was, removing it from the game. In *FIDE chess*, the target square is given by its relative position to the piece's original square but variants such as Pritchard's *start-again chess* utilize moves defined by absolute coordinates. Usually this is the only consequence, but for a general definition of a piece, there may be side effects:

- Putting a new piece on the board – as is the case during pawn promotion in *FIDE chess* pawns or *shogi* rook, bishop, silver, knight, lance and pawn – this is not restricted to the target square (i.e. a piece promotion as it gets replaced by the new), the king and the locust in *locust chess* can spawn new pieces without destroying themselves.

- Capturing a piece that is not on the target square – as is the case during en passant capture in *FIDE chess* or during any move of the locust in *locust chess*.

- Moving another piece from place to place – as is the case during castling in *FIDE chess* (this can of course be represented as a combination of capturing a piece and putting a new piece onto another place).

In *FIDE chess* the conditions deciding where a piece can move (aside from the rule that at the end of one's move, their king must not be in check) typically require the squares between the source and target to be free and the target to be either free or occupied by a piece of the opposing color (in some case only one of those is allowed, such as for pawn capture in *FIDE chess*). Less commonly used conditions include:

- Requirements on squares that are not in the path – in Betza's anti-runners

- Requirement that the moved piece has not yet been moved – in castling or pawn double move

- Requirement that a given square is not under attack – in castling

- That a piece is in the way – in the *xiangqi* chariot or the popular fairy chess piece grasshoper (a piece that only moves by jumping over another piece and landing right behind it)
- That the piece is in a particular part of the board – in *xiangqi* soldier and in pawn promotion in *FIDE chess* as well as all the promotions in *shogi* (for the standard pawn, not being moved yet and being on the second rank is equivalent in *FIDE chess*, on the other hand the fish in Moeser's *fish chess* is a pawn that can move backwards and it can use its double advance move any time it is on the second rank)
- That another piece is in a given place and has not been yet moved – in *FIDE chess* castling
- That another piece has just played a particular move – in en passant capture
- This piece has not yet played a particular move – in Schmittberger's *teleport chess* (or this piece has not yet played it a given number of times in Kevin Lawless's *super queen chess*)

It's important to note that checking whether a given square is under attack may in a generalized view lead to situations without a correct resolution – consider two pieces in opposing corners of a 2x2 board with move "can move 1 forward if the square 1 to the left isn't under attack".

3.1.1 Fairy piece moves for the purpose of this work

To describe as many of the above mentioned mechanics as possible while making the engine's implementation feasible, the generalized definition of a move we will use will consist of:

- target square – can be defined either in relative position to the piece's starting square or in absolute position regardless of where the piece is
- list of conditions that have to be satisfied to make the move legal – conditions may be in the form of:
  - a certain square is empty, occupied by an opponent, occupied by a friend, attacked or unattacked
  - a certain square is occupied by a piece with a particular flag
  - the piece starts in a particular area of the board
  - the piece ends in a particular area of the board
- list of consequences of the move aside from moving the playing piece – those effects may be in the form of:
  - piece on a certain square is moved to another square
  - a new piece is put onto a given square
  - a piece on a certain square is taken

– a flag is set/unset for a certain piece that is already on the board

The flags mentioned in this definition are set at the start of a game according to the variant's rules (in *FIDE chess* kings, rooks and pawns would start with a flag that indicates they're unmoved) and then handled according to the move effects (in FIDE chess moving a pawn, rook or king unsets the unmoved flag and advancing a pawn two squares sets the en passant takeable flag). This feature can be very powerful – it is even possible to forget piece types altogether and only use flags to distinguish pieces from each other – in most variants that only makes the variant description more obfuscated for no benefit, but it is useful for describing variations such as *cannibal chess* where a piece obtains all the moves of any piece it captures.

3.2 Notation

To describe any move fitting the description given above, I will use the following notation.

- A square can be denoted as *(rowIndex colIndex)* or *($rowIndex $colIndex)* – with dollar signs indicating it is absolute value.
- A condition requiring a square to be unoccupied can then be written as *[square] unoccupied* – and equivalently for *occupied*, *opponent*, *friend*, *attacked* and *unattacked*.
- A condition requiring a piece on a square to have a certain flag can be written as *[square] *[flag]*.
- A condition requiring the playing piece to be in a certain part of the chess board can be written as *ROW < [number] or COL < [number]*.
- An effect moving a piece onto a certain square can be written as *[pieceString] -> [square]*.
- An effect setting a flag of a piece can be written as *[square] +flag*.
- An effect unsetting a flag of a piece can be written as *[square] -flag*.
- An effect moving a piece from one square to another can be written as *[square] -> [square]*.

A move can then be denoted as *{ -target [square] -conditions [[condition]] -effects [[effect]] }*. A piece can be denoted as a list of moves in the previous notation followed by a list of flags this piece can have. If a flag is to only last for the duration of one move (like the flag indicating a pawn can be captured en passant) its last character must be an underscore, otherwise the flag is to last until unset by another effect. If a flag should be set to true at the start of the game, its name should be preceded by a + symbol.

Since it is very inconvenient to write pieces in this rather verbose notation, there are tools to enable translation from Betza's notation (described in section 2.1.1) into this notation as well as shorthands to describe pieces indescribable in Betza's notation more conveniently – those shorthands include:

- defining variables to iterate over a set of values and create multiple moves that way
- using a / operator to describe multiple options in one effect, i.e. multiple moves differing in the effect only or multiple options in one condition, i.e. a move that requires at least one of the |-separated conditions satisfied to be legal
- using *promotion [ROW </> n COL </> n] -> [piecesStrings]* to indicate a promotion zone – use *opromotion* if the promotion is optional like in Shogi

A variant can then be defined by a config file consisting of:

- one line per piece in the format of *[pieceName] {move1}{move2}... flag1 flag2...*
- an empty line followed by 8 lines describing the default position by piece names separated by spaces

3.3 Definitions of some popular chess games

In this section we will show how the rules of some chess variants can be written using the notation described above.

For convenience, the management of the "unmoved" flag is ignored, the program assumes it automatically.

3.3.1 FIDE chess

Fide chess can be defined (with explanation given piece by piece) as:

```
p fmW1fcF1 promotion t[ROW = 7] -> b|n|r|q
{-square (2 0) -conditions (1 0) empty (2 0) empty (0 0) *unmoved
-effects (2 0) +en_passant_takeable}
{-square (1 1) -conditions (0 1) *en_passant_takeable
        (1 1) empty -effect . -> (0 1)}
{-square (1 -1) -conditions (0 -1) *en_passant_takeable
        (1 -1) empty -effect . -> (0 -1)}
_en_passant_takeable
```

`fmW1cF1`

is the Betza notation for a piece that has a forward move as the atom *W* restricted to a distance of 1 and a capture move as the atom *F*, the whole expression expands to

```
{-square (1 0) -conditions (1 0) empty}
{-square (1 1) -conditions (1 1) opposite}
{-square (1 -1) -conditions (1 -1) opposite}
```

describing three moves – one straght forward onto an empty square and two forward and to the side onto a square occupied by an opponent. The `promotion` specification makes it so that the whole line actually expands into

```
{-square (1 0) -conditions (1 0) empty t[ROW != 7]}
{-square (1 1) -conditions (1 1) opposite t[ROW != 7]}
{-square (1 -1) -conditions (1 -1) opposite t[ROW] != 7]}
{-square (1 0) -conditions (1 0) empty t[ROW = 7]
-effect b|n|r|q -> (1 0)}
{-square (1 1) -conditions (1 1) opposite t[ROW = 7]
-effect b|n|r|q -> (1 1)}
{-square (1 -1) -conditions (1 -1) opposite t[ROW] = 7]
-effect b|n|r|q -> (1 -1)}
```

to assert that if the pawn's target is on the eigth row, it has to promote and if it isn't, then it can not promote.

The last three moves describe respectively the pawn's double move with setting the `en_passant_takeable` and indirectly capturing a piece with that flag set on the left or the right to the pawn.

`n N1`

```
N1
```

is Betza's *N* atom restricted to distance of 1. It expands to

```
{-vars a in [1,2] -square (a 3-a) -conditions (a 3-a) empty|opposite}
```

to describe a move in any direction in the shape *(1, 2)*.

```
r W
```

```
W
```

is Betza's unrestricted *W* atom. On an 8x8 board it expands to

```
{-vars a in <1,7> b in <1,a) -square (a 0)
-conditions (b 0) empty (a 0) empty|opposite}
{-vars a in <1,7> b in <1,a) -square (-a 0)
-conditions (-b 0) empty (-a 0) empty|opposite}
{-vars a in <1,7> b in <1,a) -square (0 a)
-conditions (0 b) empty (0 a) empty|opposite}
{-vars a in <1,7> b in <1,a) -square (0 -a)
-conditions (0 -b) empty (0 -a) empty|opposite}
```

to describe the moves of a rook. Note that the *(0 -b) empty* conditions get united over all moves with the same target square, i.e. over all combinations of *a* and *b* with the same *a*, asserting that all squares in the rook's path are empty.

```
b F
```

```
F
```

is Betza's unrestricted *F* atom. On an 8x8 board it expands to

```
{-vars a in <1,7> b in <1,a) -square (a a)
-conditions (b b) empty (a a) empty|opposite}
{vars a in <1,7> b in <1,a) -square (-a a)
-conditions (-b b) empty (-a a) empty|opposite}
{vars a in <1,7> b in <1,a) -square (a -a)
-conditions (b -b) empty (a -a) empty|opposite}
{-vars a in <1,7> b in <1,a) -square (-a -a)
-conditions (-b -b) empty (-a -a) empty|opposite}
```

to describe the moves of a bishop.

```
q WF
```

The queen is just the combination of a rook and a bishop.

```
k W1F1
{-square (0 2)
-conditions (0 1) empty (0 1) unattacked (0 2) empty (0 3) *unmoved
        (0 0) *unmoved (0 0) unattacked
-effects (0 3) -> (0 1)}
{-square (0 -2)
        -conditions (0 -1) empty (0 -1) unattacked (0 -2) empty (0 -2)
        unattacked (0 -3) empty (0 -4) *unmoved (0 0) *unmoved
        (0 0) unattacked
-effects (0 -4) -> (0 -1)}
```

`F1`

is Betza's *F* atom restricted to distance of 1. It expands to

```
{-vars a in [1, -1] b in [1, -1] -square (a b)
-conditions (a b) empty|opposite}
```

`W1`

is Betza's *W* atom restricted to distance of 1. It expands to

```
{-vars a in [1, -1] - square (a 0) -conditions (a 0) empty|opposite}
{-vars a in [1, -1] - square (0 a) -conditions (0 a) empty|opposite}
```

The last two moves describe short and long castling, asserting neither king nor rook have moved and that none of the squares the king will move over are attacked.

3.3.2 Shogi

Shogi without placing pieces from hand can be defined as:

```
玉   W1F1
飛   W opromotion [ROW > 5]-> 竜
竜   WF1
角   F opromotion [ROW > 5]-> 馬
馬   FW1
金   W1fF1
銀   fW1F1 opromotion [ROW > 5]-> 金
珪   ffN1 opromotion [ROW > 5]-> 金
香   fW opromotion [ROW > 5]-> 金
歩   fW1 opromotion [ROW > 5]-> 金
```

The `opromotion` shorthand expands into new moves that replace the piece by its promoted version. `F opromotion [ROW > 5]-> 馬` then expands into

```
{-vars a in <1,7> b in <1,a) -square (a a)
-conditions (b b) empty (a a) empty|opposite}
{vars a in <1,7> b in <1,a) -square (-a a)
-conditions (-b b) empty (-a a) empty|opposite}
{vars a in <1,7> b in <1,a) -square (a -a)
-conditions (b -b) empty (a -a) empty|opposite}
{-vars a in <1,7> b in <1,a) -square (-a -a)
-conditions (-b -b) empty (-a -a) empty|opposite}
```

```
{-vars a in <1,7> b in <1,a) -square (a a)
-conditions (b b) empty (a a) empty|opposite [ROW > 5]
-effects 馬  -> (a a) }
{-vars a in <1,7> b in <1,a) -square (-a a)
-conditions (-b b) empty (-a a) empty|opposite [ROW > 5]
-effects 馬  -> (-a a) }
```

```
{-vars a in <1,7> b in <1,a) -square (a -a)
-conditions (b -b) empty (a -a) empty|opposite [ROW > 5]
-effects 馬  -> (a -a) }
{-vars a in <1,7> b in <1,a) -square (-a -a)
-conditions (-b -b) empty (-a -a) empty|opposite [ROW > 5]
-effects 馬  -> (-a -a) }
```

If there was `promotion` instead of `opromotion`, the non-promoting moves would have a condition `[ROW <= 5]` to assert that a non-promoting move into/out of the promotion zone is illegal, i.e. the promotion is obligatory.

3.3.3 Locusts chess

Locusts chess can be defined using the following rules. Out of all the mentioned variants, this may be the one most alien to traditional chess, none of its pieces can be even partially written using Betza's notation.

```
k {-vars a in <2,11> b in <2,11> -square ($a $b) -conditions ($a $b) empty
-effects l -> (0 0)}
```

The king can move to any unoccupied place on the board, leaving behind a locust (l).

```
l
{-vars a in <1, 11> b in <1,a-1) -square (a 0)
        -conditions (a-1 0) opposite
        (a 0) empty (b 0) empty -effects . -> (a-1 0) e -> (0 0)}
{-vars a in <1, 11> b in <1,a-1) -square (0 a)
        -conditions (0 a-1) opposite
        (0 a) empty (0 b) empty -effects . -> (0 a-1) e -> (0 0)}
{-vars a in <1, 11> b in <1,a-1) -square (-a 0)
        -conditions (-a+1 0) opposite
        (-a 0) empty (-b 0) empty -effects . -> (-a+1 0) e -> (0 0)}
{-vars a in <1, 11> b in <1,a-1) -square (0 -a)
        -conditions (0 -a+1) opposite
        (-a 0) empty (0 -b) empty -effects . ->  (0 -a+1) e -> (0 0)}
{-vars a in <1, 11> b in <1,a-1) -square (a a)
        -conditions (a-1 a-1) opposite
        (a a) empty (b b) empty -effects . -> (a-1 a-1) e -> (0 0)}
{-vars a in <1, 11> b in <1,a-1) -square (-a a)
        -conditions (1-a a-1) opposite
        (-a a) empty (-b b) empty -effects . -> (1-a a-1) e -> (0 0)}
{-vars a in <1, 11> b in <1,a-1) -square (a -a)
        -conditions (a-1 1-a) opposite
        (a -a) empty (b -b) empty -effects . -> (a-1 1-a) e -> (0 0)}
{-vars a in <1, 11> b in <1,a-1) -square (-a -a)
        -conditions (1-a 1-a) opposite
        (-a -a) empty (-b -b) empty -effects . -> (1-a 1-a) e -> (0 0)}
```

The locust's move `-vars a in <1, 11> b in <1,a-1) -square (a 0) -conditions (a-1 0) opposite (a 0) empty (b 0) empty -effects . -> (a-1 0) e -> (0 0)` jumps over an opposing piece (`(a-1 0) opposite`) to an empty place (`(a 0) empty`), captures it indirectly (`. -> (a-1 0)`) and leaves a leo (e) behind. The condition `(b 0) empty` asserts there are no pieces between the starting square and the piece jumped over.

Its other moves are the same pattern in different directions.

```
e
{vars a in <1,11> b in <1,a-1> c in <b+1,a-1> d in <1,b-1> -square (a 0)
        -conditions (b 0) opposite|friend (a 0) opposite
        (c 0) empty (d 0) empty
        -effects (b 0) -> (b 0)}
{vars a in <1,11> b in <1,a-1> c in <b+1,a-1> d in <1,b-1> -square (-a 0)
        -conditions (-b 0) opposite|friend (-a 0) opposite
        (-c 0) empty (-d 0) empty
        -effects (-b 0) -> (-b 0)}
{vars a in <1,11> b in <1,a-1> c in <b+1,a-1> d in <1,b-1> -square (0 a)
        -conditions (0 b) opposite|friend (0 a) opposite
        (0 c) empty (0 d) empty
        -effects (0 b) -> (0 b)}
{vars a in <1,11> b in <1,a-1> c in <b+1,a-1> d in <1,b-1> -square (0 -a)
        -conditions (0 -b) opposite|friend (0 -a) opposite
        (0 -c) empty (0 -d) empty
        -effects (0 -b) -> (0 -b)}
{vars a in <1,11> b in <1,a-1> c in <b+1,a-1> d in <1,b-1> -square (-a a)
        -conditions (-b b) opposite|friend (-a a) opposite
        (-c c) empty (-d d) empty
        -effects (-b b) -> (-b b)}
{vars a in <1,11> b in <1,a-1> c in <b+1,a-1> d in <1,b-1> -square (a a)
        -conditions (b b) opposite|friend (a a) opposite
        (c c) empty (d d) empty
        -effects (b b) -> (b b)}
{vars a in <1,11> b in <1,a-1> c in <b+1,a-1> d in <1,b-1> -square (-a -a)
        -conditions (-b -b) opposite|friend (-a -a) opposite
        (-c -c) empty (-d -d) empty
        -effects (-b -b) -> (-b -b)}
{vars a in <1,11> b in <1,a-1> c in <b+1,a-1> d in <1,b-1> -square (a -a)
        -conditions (b -b) opposite|friend (a -a) opposite
        (c -c) empty (d -d) empty
        -effects (b -b) -> (b -b)}
```

The leo's move `vars a in <1,11> b in <1,a-1> c in <b+1,a-1> d in <1,b-1> -square (a 0) -conditions (b 0) opposite|friend (a 0) opposite (c 0) empty (d 0) empty -effects (b 0) -> (b 0)` jumps over any piece (`(b 0) opposite`) onto an opposing piece (`(a 0) opposite`). Unlike the locust that jumps to the square right after the piece jumped over (the platform), the leo can jump any distance as long as there are no pieces between its starting position and the platform (`(c 0) empty`) and no pieces between the platform and its ending position (`(d 0)`

`empty`). The effect `(b 0) -> (b 0)` is there to differentiate moves with same target square but different place of the platform from each other.

Its other moves are the same pattern in different directions.

4. Engine

To compare different strategies applicable to fairy chess games, I needed an engine capable of understanding their rules and playing those games.

4.1 Analysis of requirements for the engine

The engine required the following features:

- It had to be able to learn to play (in terms of playing moves in accordance to the rules) a variant from its configuration file as described in section on notation (Section 3.2).
- It had to be able to play a game both against a human opponent and against another engine.
- It had to have a simple way of changing its evaluation function.

Existing engines either don't support all the movement patterns I wanted to support like FairyMax 14 or don't allow me to experiment with their evaluation function like FairyStockfish 14.

4.2 High level overview

The engine wrapper if given a valid configuration file is able to set the starting position and then play one side of the game – it's notified of opponent's moves and replies with moves of its own, trying to win the game – it can use multiple communication protocols for receiving/sending moves as long as they're implemented using the engine's interface.

The chess engine I've written consists of several interconnected parts. An image can be seen in 4.1. In the following sections I will describe what those parts are and how the engine uses them.

4.3 Game rules representation

Once initialized by a game configuration file, the engine contains an array of all pieces used in the game. The pieces are each associated with moves they can play – those are then used to figure out which positions can follow from another in a game tree as well as when the game is over due to a checkmate or a stalemate. This way the same engine can play multiple different games if it is instantiated with different list of pieces.

Furthermore the engine knows the starting position of the variant it is currently playing that is to be set at the start or any subsequent restart of the game.

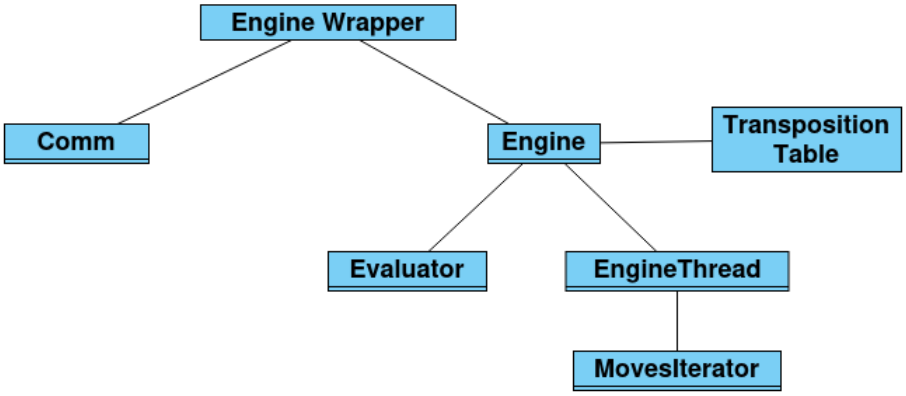

Figure 4.1: Overview of the engine components – connections indicate that the component closer to Engine Wrapper owns the one further away

4.4 EngineThread

The engine thread searches through the position tree using a depth first min-max algorithm. Until the maximal depth, each discovered position is passed to a MovesIterator to iterate over all possible continuations. Positions at max depth are evaluated statically by the engine's Evaluator.

4.5 Evaluator

The engine utilizes an Evaluator class to provide static evaluations of positions at the bottom of the search tree – changing the implementation of this class will have large impact on how the engine plays in a game – later in this work I will be comparing different Evaluator classes in games against each other to see which principles are useful for which variants.

Heuristics that have been used by chess players such as Ralph Betza [2] and Steve Mayer [10] to estimate the value of chess pieces and some of which I will implement in the engine's evaluators include the following:

- mobility over an empty board – in one move or in several moves, placed at a random square
- mobility over a board in terms of which squares can be reached regardless of the number of moves it takes – discounting colorbound pieces or even pieces like the Alfil (Betza's *(2,2)* atom) that can only ever reach about one eighth of the chessboard
- mobility over a board with randomly placed obstacles – in one move or in several moves, placed at a random square
- checkmating potential – a piece that can attack multiple squares immediately next to each other is more dangerous in terms of delivering a checkmate than one that can't – this changes if the king has a different movement pattern, in such cases mirroring the king's movement may be the better heuristic for checkmate potential.

- special abilities potential – pieces that can place new pieces on board can be very powerful, as can be pieces that can destroy opposing pieces without jumping on their square. Both the likelihood that the move with a given special effect can be played and the actual magnitude of its impact when played are to be taken into account.

Furthermore, the evaluator should take into account how the value of a piece differs based on where on the board it is:

- mobility over an empty board in one or several moves when starting on a particular square
- mobility over a board with randomly placed obstacles in one or several moves when starting on a particular square
- proximity to places where moves with a positional condition given in absolute numbers can be played, e.g. promotion zones in Shogi or FIDE chess
- relative safety especially in case of the king but to some extent all valuable pieces

Lastly, an evaluator may benefit from taking into account concepts that can't fully be expressed in terms of what was mentioned above, such as proximity of pieces to the king, teamwork of pieces in a position (protection of each other, jointly controlled squares...), positional weak points and more.

More on evaluators will be in the following chapter on their technical realization.

4.6 MovesIterator

In its simplest implementation just a wrapper over a vector of positions, the MovesIterator is a class that given a position iterates over all the positions that may follow from it. Various implementations of MovesIterator can use heuristics regarding the order in which they iterate over the child positions to attempt to maximize the benefit of alpha/beta pruning in minmax search done by the engine thread.

Heuristics employed by the iterators will consist of the following:

- strongest piece heuristic – the most valuable piece is most likely to have the best move
- monotonous static evaluation heuristic – if a child position has a good static evaluation, it is likely that it will turn out as a good move even as examined into more depth
- capture move heuristic – a move that captures an opposing piece is more likely to be a good move, the more powerful that piece is the more likely it is to be a good move – this is in fact very similar to the monotonous static evaluation heuristic, since the moves that change the static evaluation the most are usually capturing moves.

4.7 EngineWrapper

Technically speaking not a part of the engine – it'd be more accurate to say the engine is a part of the wrapper – the wrapper encapsulates the engine and the communication protocol used to communicate with the opponent (depending on the protocol either over a server or in a peer to peer fashion). It also keeps information about the game's current position as well as tables of pieces, symbol tables used to translate positions as understood by the protocols into positions as understood by the engine and tables used for hashing of positions.

4.8 TranspositionTable

Hashtable used by engine as a cache of previously seen positions. The table stores entries of either exact evaluation, upper bound evaluation or lower bound evaluation and optionally the best available move. Zobrist hashing [17] is used to map positions to entry buckets.

5. Technical solution

In this chapter I will describe how the engine's components mentioned in the previous chapter are implemented using C++ and how they actually work when playing a game.

5.1 EngineWrapper

The EngineWrapper is an object that encapsulates the engine itself and a communication object capable of sending/receiving moves from the opponent. It is also responsible for maintaining the game state, both in terms of storing data related to the variant being played (more on it in the following section) as well as storing the current board for the game being played. A diagram of its component can be seen in 5.1.

5.2 Game rules representation

The engine wrapper remembers a vector of pieces. Each piece is represented by a vector of its possible moves and each move is represented by a vector of conditions that have to be satisfied for it to be playable and a vector of effects that take place if it's played.

Since the conditions for different moves of one piece are very often shared among several pieces (all pieces described by the slider pattern, such as rooks which will for example share the condition *(0 1) empty* for all moves to *(0 ?)*), the fetching of all available moves of a piece is done at once. The vector of all conditions relating to the moves of a piece is stored in the piece with individual moves only have a list of indices into the vector – when the engine looks for available moves of a piece, it creates an array of truth values for this list of conditions, evaluating each of them only once. A diagram of how pieces are represented can be seen in 5.2.

5.3 Transposition Table

The transposition table is implemented using an unordered map from positions to cache entries. Each cache entry contains the following:

- evaluation value for the given position
- indicator of exactness of the evaluation – either *EXACT* if the position was evaluated by examining all of its subtree to a given depth, or *UPPER_BOUND* if the position was evaluated until it was found that it's evaluation was lower than the current alpha and it wasn't examined further or *LOWER_BOUND* if it was found that it's evaluation was higher than the current beta and it was not examined further
- Depth to which this particular position was evaluated when this cache result was obtained

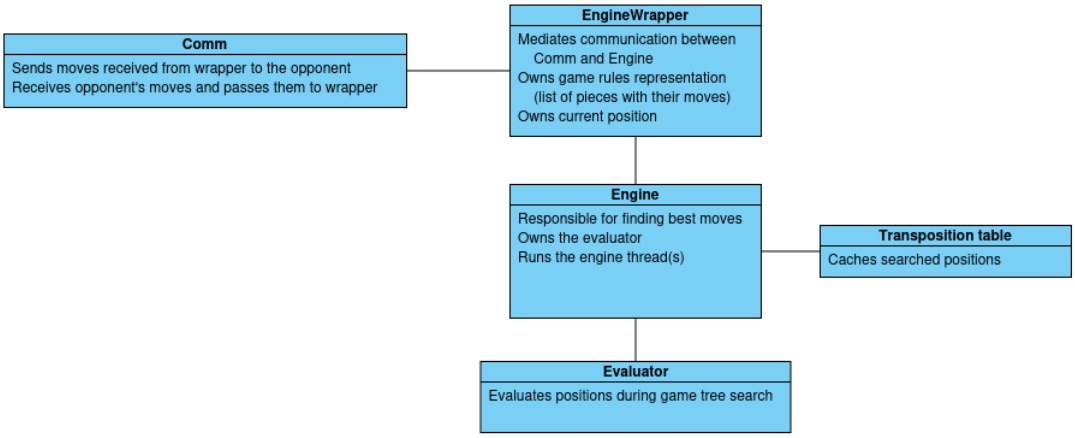

Figure 5.1: Diagram of EngineWrapper and its main constituents

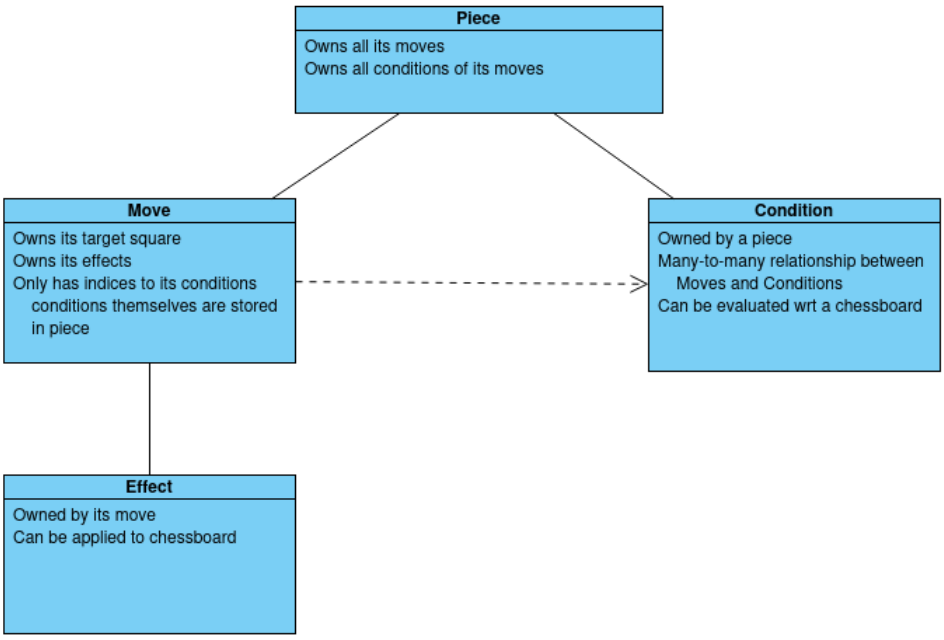

Figure 5.2: Diagram of objects representing the game rules

- Best move playable in this position – only if the evaluation is *EXACT*

When cleaning up, the transposition table removes those entries that have the lowest deep, i.e. those whose evaluation is likely least accurate and was least expensive to obtain.

5.4 EngineThread

The engine thread uses a depth-first search in the following fashion:

- receive a node – a position, the player to move (*MIN*/black or *MAX*/white) and its associated depth, alongside the alpha and beta of the relevant part of the tree traversed so far.
- If the node is in cache with indicator *EXACT*, return its evaluation.
- If the node is in cache with indicator *UPPER_BOUND* and the current beta is higher than this upper bound, return the upper bound so the parent call can continue.
- If the node is in cache with indicator *LOWER_BOUND* and the current alpha is higher than this lower bound, return the lower bound so the parent call can continue.
- If the node is not at the maximal depth, the search will expand it. If it is at maximal depth, the engine will evaluate it statically using its Evaluator object.

5.4.1 Expanding a node

To expand a node of the search tree, the depth evaluation method is parameterized by an implementation of MovesIterator. It uses the iterator to generate children of the original node and then (if the child position isn't in cache, in which case it can just return the cached value) call itself recursively on them until either:

- The player to move is *MIN* and the current child's evaluation is lower than the current alpha – going further would be a waste of resources – return the current node's eval as this child's eval with the sign that it is an upper bound.
- The player to move is *MAX* and the current child's evaluation is higher than the current beta – going further would be a waste of resources – return the current node's eval as this child's eval with the sign that it is a lower bound.
- The child's evaluation is MAXINT and current player is *MAX* or MININT and the current player is *MIN*, signifying that a king was taken and the node has been an illegal position. Return child's evaluation with the sign that it was exact.
- there are no more children to investigate – return the minimal evaluation of a child encountered if playing as *MIN*or the maximal if playing as *MAX* with the exactness sign of the child node.

Note on checkmates and stalemates: The engine saves resources by not checking legality of moves with regards to king's safety. Instead, every valid evaluator implementation is required to evaluate a position with the king missing as MAXINT if the black king is missing or MININT if the white is missing and the engine thread will never investigate such position further – this means the engine will never play a move leading to a possibility of immediate capture of the king if it has any other option. Checkmates and stalemates can then be detected as nodes where all of their children have MAXINT or MININT evaluation.

5.5 MoveIterator

Moves iterator is given a starting position and provides a way to iterate over its children using a `getNext()` and a `hasNext()` method. Aside from just being a container of the positions, its goal is to maximize the benefit of the alpha/beta pruning by iterating over the child positions from best to worst. The interface allows both eager and lazy ways of iteration.

5.5.1 Eager iterators

Eager iterators are ones that generate all positions at once and then iterate over them. Their advantage is the ability to use heuristics based on sorting all the generated positions, such as the monotonous static eval assumption. They can implement all the heuristics of lazy evaluators and some more, their disadvantage being that if they're actually good at putting more important positions first, they'll waste most of their effort creating ones that will never be examined.

The iterators implemented like this are:

- Monotonous moves iterator: Orders child nodes by their static evaluation (ascending or descending dependent on which player is expanding the node).
- Faster monotonous moves iterator: Orders child nodes by the cumulatively computed part of their static evaluation (more on this in section about evaluators – operation during playing moves), saving time by forgoing the possibly expensive full evaluation.
- Strongest piece iterator: Orders child nodes by the overall evaluation (mean over all its positional evaluation, as described in chapter on static evaluators) of the piece playing the move.
- Capture move iterator: iterates first over nodes reached by a capturing move, then over nodes reached by a non-capturing move.

5.5.2 Lazy iterators

Lazy move iterators generate children positions on demand. While they can't sort the children vector, they can implement heuristics using information available in the moves themselves (capture move heuristic, strongest piece heuristic...) and if they're actually able to maximize the alpha/beta pruning, they'll save extra work that'd be spent on creating positions that weren't worth exploring.

While the iterator interface is designed to allow such iterators, I didn't implement any iterator like that.

5.6 Evaluator

The evaluator is a class implementing static evaluation of positions. Since we want to use one general implementation for multiple chess variants, the evaluator does a lot of work at its initialization when it's given the rules and is now trying to figure out how to evaluate positions of this particular variant.

5.6.1 Initialization

On initialization, the evaluator receives a vector of pieces with their moves and it has to fill an unordered map with entries in the form of `piece_id, row_index, col_index` assigning numeric values to how a piece of the given id on the given square impacts the position's evaluation. This unordered map also has to have entries with negative values for `piece_id` to evaluate black pieces – typically it will be symmetric but it doesn't have to be (there may be games where black and white share some pieces but their armies differ in others making the shared piece's usefulness in combination with others different). Furthermore, if the piece definitions include flags, the evaluator will fill another unordered map with entries in the form of `piece_id, flag_id, row_index, col_index` mapped to numerical values indicating how a piece having a given flag will impact the position's evaluation.

After filling those tables, the evaluator may do any other work depending on how its non-static evaluation should work – notably the *KingSafetyEvaluator* creates a table for each piece that tells how long it'll need to go from any given square to any other square on an empty board.

5.6.2 Operation during playing moves

Whenever a move is played (either in the actual game or in its simulation during the engine's minmax search), the engine will call evaluator's methods
`placePiece(piece_id, row_index, col_index)`
`removePiece(piece_id, row_index, col_index)`
`setFlag(flag_id, piece_id, row_index, col_index)`
`unsetFlag(flag_id, piece_id, row_index, col_index)`.
The evaluator doesn't actually change the position in any way, it only updates its evaluation accordingly. The evaluator doesn't check whether the piece is in fact on the given square.

5.6.3 Static evaluation

To evaluate a position at the bottom of a search tree, the engine uses an Evaluator class. The evaluation calculation consists of two parts:

- sum of the values of pieces at their positions

- position based inferences that can't be measured just by the such as proximity to the king, rooks on free files, piece coverage...

This distinction is of particular importance due to the fact that the first part is recalculated from the previous value whenever a piece is taken away or put onto the board, lessening the demand on resources. On the other hand, recalculating the second part would be expensive and in fact pointless, as there are no guarantees about how exactly a particular move changes it – it has to be done anew whenever the search tree bottoms out.

5.6.4 Composite evaluator

To make comparing different approaches to evaluation easier, there is a class template that allows combining multiple evaluators. The *CombinedEvaluator<T...>* is parameterized with a list of evaluators, summing their partial normalized evaluations of a position to arrive at a conclusion.

The class *ScaledEvaluator<typename T,int Up, int Down>* is an evaluator that simply takes another evaluator *T* and returns its evaluation scaled by *Up/Down* – by itself it'd play exactly the same as *T*, but it is useful to give different weights to partial evaluations in the context of a *CombinedEvaluator*.

Lastly, any evaluator that tries to account for the promotion potential (or any other move that places new pieces on the board, if the variant has such) of pieces has to know the evaluation of those pieces – and it can't provide them itself, as it'd get stuck in an infinite recursive call if it tried to – for that reason, it has to be parameterized by an evaluator that doesn't take values of other pieces into account. The drawback of an evaluator like that is that it isn't likely to work well for pieces with multiple levels of promotion (ones that promote to a piece that promotes to another) – this could be to an extent mitigated by a *PotentialEvaluator* parameterized by another *PotentialEvaluator* with the latter one (or only the last one if we wanted more layers) parameterized in a way that evaluates pieces in a way independent of other pieces' evaluations. On the other hand, such promotion patterns aren't something I've ever seen in any chess variant.

5.6.5 Normalization of evaluators

To make *CombinedEvaluator* more useful, it is needed that their member evaluators give values within the same order of magnitude. For that reason, the evaluators using a table of pieces' evaluations have them scaled in a way that the magnitude of the vector *[evalOfPiece1, evalOfPiece2, evalOfPiece3...]* is 1 000 000. Evaluators using a table of pieces' positions will scale them in a similar way, using the mean over all squares of a piece as its evaluation for the purpose of figuring out the scaling factor. Bottom evaluators that don't use a table at all will return values rescaled to be in range from *-(1 000 000)/sqrt(num_pieces) to +(1 000 000)/sqrt(num_pieces).*

5.7 Communication Protocol

The engine supports two communication protocols, the XBoard interface's protocol and my own communication protocol with one caveat – the XBoard communication protocol was not designed to support all the variations the engine can implement, so there are variants that can't be played using this protocol.

5.7.1 XBoard protocol

This is the protocol implemented by the XBoard interface [14] as specified in chapter on Related Works. It is used if the engine is started via the XBoard chess interface, which sets the program to receive XBoard's input on stdin and to send its stdout to XBoard.

The limitation of this protocol is ignoring special effects of moves played (aside from pawn promotion and castling) – the engines receiving its output therefore can't differentiate between different moves with equal final square differing only in their effects.

5.7.2 Custom communication protocol

This protocol is a peer to peer communication protocol that allows all variants playable with pieces defined using the notation given in previous chapters.

5.7.3 Game initiation

One instance of engine is to run listening for incoming connection at a given port. Once another engine connects to it, the connecting engine sends a message setting up the variant and the position.

New variant message is as follows:

```
{
type: "new_variant",
variant: string
}
```

New position message is as follows:

```
{
type: "new_position",
board:{
pos: {
squares: [[{piece_id: int, flags: long_int}]]
},
white: bool,
moves: int
}
}
```

Then the connecting engine tells the other one which side to play.

```
{
type: "play_color",
white: bool
}
```

If `white` was set to `false`, the connecting engine will start playing as white, otherwise it will await the opponent's move.

5.7.4 Playing moves

Moves are sent in the form of:

```
{
type: "move",
move: string,
effects: string
}
```

Where `move` is a string like `(a b) -> (c d)` and `effects` string is a string of the move's effects as they're given in definition of the move, i.e. `(a b) -> (c d)` for a moving effect `[piece_string] -> (a b)` for a new piece effect, and `+-[flag_string] -> (a b)` for setting/unsetting a flag.
For example white's short castling would be denoted as

```
{
type: "move",
move: "($0 $4) -> ($0 $6)",
effects: "($0 $7) -> ($0 $5)"
}
```

A piece utilizing its once-per-game teleport move in Schnittberger's *teleport chess* would look like

```
{
type: "move",
move: "($0 $2) -> ($3 $7)",
effects: "-teleport -> ($3 $7)"
}
```

6. Evaluators and iterators

So far I have outlined how the engine will work and what strategies it may use in playing a game. Now the goal is to find out what effect will those approaches have on the engine's decision making, which heuristics will prove superior over others in actual games.

In this chapter I will describe in more detail the approaches to evaluation of chess variant positions, as well as approaches to iteration heuristics that will be used by the engine.

6.1 Evaluators to be compared

In this section I will describe the logic implemented by the evaluators that (or the combinations thereof) will be compared to each other during the experiments.

Note: "Board randomly populated by obstacles" is a board with randomly placed pieces that have black or white color (i.e. they can be captured or be the platform for a grasshopper piece), but are not any kind of piece defined by the variant and don't have any flags. "Random valid position" is a position reached from the starting position of the variant by a random sequence of valid moves.

6.1.1 Static mobility evaluator

Evaluator that assumes the piece's value is dependent on how many different squares it can go to. Specifically it looks at:

- How many squares the piece can on average reach when placed on an empty board.
- How many squares the piece can on average reach when placed on a board randomly populated by obstacles.
- How many squares the piece can on average reach when placed into a random valid position.

The evaluator considers sequences of 1, 2 and 3 moves of the piece.

The equation used for each square is:

```
P_1 + P_2/2 + P_3/4
```

where `P_n` is the number of different positions that reached by all sequences of n moves of the given piece.

This evaluator calculates one number for each piece and copies it to all `piece, row, col` fields for that piece – it doesn't take into account the pieces' positions.

A similar evaluator was proposed by *I. J. Good in Five Year Plan for Automatic Chess* [7].

6.1.2 Location-dependent mobility evaluator

Evaluator that works like the static mobility evaluator, but it stores the value of the piece for each square separately.

6.1.3 Position-dependent mobility evaluator

Evaluator that only looks at the number of available moves in the current position – it uses no evaluation tables.

The evaluator considers only the single next move (it has to do the brunt of its work during game time rather than during initialization and considering more would be akin to adding an extra depth to the search tree).

The evaluation returned is:

`[number of moves white's pieces have - number of moves black's pieces have]`

This evaluation method was mentioned *I. J. Good in A five-year plan for automatic chess* [7].

6.1.4 Static capturing ability evaluator

Evaluator that works similar to the static mobility evaluator, but instead of which squares it can go to, it looks at which pieces it can capture. Specifically it looks at:

- How many pieces on average it can capture when placed on a random square on a board randomly populated by obstacles.
- How many pieces on average it can capture when placed into a random valid position.

The evaluator considers the total number of captured pieces in 1, 2 and 3 moves of the piece.

The equation used for each square is: `C_1 + C_2/2 + C_3/3`,

where `C_n` is the total number of captures in all combination of n moves the piece could play.

This evaluator calculates one number (the mean result over all squares) for each piece and copies it to all `piece, row, col` fields for that piece – it doesn't take into account the pieces' positions.

6.1.5 Location-dependent capture ability evaluator

Evaluator that works like the static capturing ability evaluator, but it stores the value of the piece for each square separately.

6.1.6 Position-dependent capture ability evaluator

Evaluator that only looks at the number of available capture moves in the current position – it uses no evaluation tables.

The evaluator considers only the single next move (it has to do the brunt of its work during game time rather than during initialization and considering more would be akin to adding an extra depth to the search tree).

The evaluation returned is:

`[number of capturing moves white's pieces have - number of capturing moves black's pieces have]`,

6.1.7 Static potential evaluator

Evaluator that estimates the strength of the piece by estimating how much the piece's moves can change the position's evaluation.

This evaluator has to be parameterized by an inner evaluator class to know how to evaluate the aftermath of effects taking place. Specifically this evaluator looks at the maximal change in evaluation according to its inner evaluator over all possible sequences of moves of the piece.

The evaluator considers sequences of 1,2 and 3 moves of the piece.

The equation for each square is:
`maxDE_1 + maxDE_2/2 + maxDE_3/4`, where `maxDE_n` is the expected maximal evaluation change over all sequences of n moves the piece could play.

This evaluator calculates one number (the average of square values) for each piece and copies it to all `piece, row, col` fields for that piece – it doesn't take into account the pieces' positions.

6.1.8 Position-dependent potential evaluator

Evaluator that works similarly to the static potential evaluator, but stores the evaluation for each square separately.

6.1.9 Position soundness evaluator

Evaluator that looks at how many pieces in the current position are protected by another piece, i.e. pieces positioned so that if they were captured, a friendly piece would be able to capture the attacker – it uses no evaluation tables. The king doesn't count as protected, as capturing its capturer would be pointless.

The evaluation value is:

```
 [number of white's capture moves with target on a white non-king piece]
-[number of black's capture moves with target on a black non-king piece].
```

6.1.10 King safety evaluator

Evaluator that assumes a position is better if many friendly pieces are close to the king and few opposing pieces are.

The evaluator has to be parameterized by another to know how dangerous which pieces are.

It looks at how far (in terms of number of moves needed to get onto the king's square as well as number of squares to get within 1 square of it) the pieces are from the king and how dangerous they are.

The evaluation value is:

```
[sum over all white's pieces:
 p_D * (2/(p_PO) + 1/(p_PO1)
 + 2/(p_PM) + 1/(p_PM1)]
 - [sum over all black's pieces:
 p_D * (2/(p_PO)+ 1/(p_PO1)
 + 2/(p_PM) + 1/(p_PM1)],
```

where:

- `p_D` is the piece's dangerousness according to a provided inner evaluator,
- `p_PO` is how many moves it'd take to get to the opponent's king over an empty board,
- `p_PO1` is how many moves it'd take to get within one square from opponent's king over an empty board,
- `p_PM` is how many moves it'd take to get to own king over an empty board,
- `p_PM1` is how many moves it'd take to get within one square from own king over an empty board.

6.1.11 Simple king safety evaluator

Evaluator that assumes a position is the better the further our king is from in the center.

The evaluation value is:

```
([row of black king] * [number rows - row of black king]
  - [row of white king] * [number rows - row of white king])
 / [number rows]^2 / 4
 + ([col of black king] * [number columns - col of black king]
 - [col of white king] * [number columns - col of white king]
 / [number columns^2] /4.
```

6.2 Iterators to be tested

In this section I will describe the heuristics used by the iterators that will be tested to see which provide the biggest benefit in terms of minimizing the number of nodes that have to be resolved until the EngineThread arrives at a result.

6.2.1 Monotonous evaluation iterator

The iterator assumes that the evaluation is likely to change monotonously, i.e. the move that has the best desired impact at depth 0 will likely have the best desired impact at depth 1 and so on until the bottom of the search.

6.2.2 Faster monotonous evaluation iterator

The iterator assumes that the evaluation is likely to change monotonously, i.e. the move that has the best desired impact at depth 0 will likely have the best desired impact at depth 1 and so on until the bottom of the search. Unlike the Monotonous evaluation iterator, this iterator only looks at the cumulative part of the evaluation function, saving time that'd be needed to go through the possibly expensive static evaluation.

6.2.3 Captures first iterator

This iterator assumes capture moves are more likely to be good moves than noncapture moves, it iterates over capturing moves first.

6.2.4 Strongest piece moves iterator

This iterator assumes the strongest piece (according to a given inner evaluator) is the most likely to have the best move, it iterates over the moves of pieces according to their value in its evaluator's piece table.

7. Experiments

In this section, I will present the results of the evaluators playing games against each other, to see which playing strategies generalize soundly into chess variations and create the best generalized chess engine and which – if any – are misleading or inferior to other approaches. Furthermore, I will present results of different iterators helping engines find the best moves in the same positions to see which of them provide the best speedup.

7.1 Experiment settings

In this section I will describe how exactly the conditions of the engines' games against each other were set up.

7.1.1 Evaluators comparing method

The comparing of evaluators consists of having an engine using one evaluator play against another engine utilizing a different evaluator. Each pair of engines is to be tested on multiple chess variants.

Besides the results of the games themselves, we will take note of how long different evaluators took to play a move when using different moves iterators.

7.1.2 MovesIterators comparing method

Since a different MovesIterator implementation by itself won't change how the engine plays (except in the case the depth evaluation of two positions is the exact same), their goal is just to maximize how much benefit is gained from A/B pruning. The metrics we will focus on are the iterator's impact on the branching factor of the search tree as well as the actual wall-time per move when using an iterator to account for iterator's own overhead.

7.1.3 Used evaluators

Since the amount of possible evaluators created by parameterizing a CombinedEvaluator is very high, it was necessary to choose a select few evaluators to compare experimentally.

All the evaluators used in this chapter are a combination of a static table evaluator (from here referred to as *base evaluator*), chosen from:

- Static Mobility Evaluator – from here referred to as *SME*
- Static Potential Evaluator (using Static Mobility Evaluator as the inner evaluator) – from here referred to as *SPE*
- Static Captures Evaluator – from here referred to as *SCE*

and an evaluator to evaluate the pieces' positions (from here referred to as *position evaluator*), chosen from:

- none
- position mobility/potential/captures evaluator (always using the one derived from the combination's static evaluator)
- King Safety Evaluator (using the combination's static evaluator as its inner evaluator)
- Simple King Safety Evaluator
- Immediate Mobility Evaluator
- Immediate Captures Evaluator
- Position Soundness Evaluator

This gives us a total of 21 evaluators.

7.1.4 Variants played

The variants played were as follows:

- FIDE chess
- Nutty Knights from Ralph Betza's chess with different armies [3]
- Colorbound Clobberers from Ralph Betza's chess with different armies [3]
- Locusts by Yu Ren Dong [6] – modified by being played at an 8x8 board to make the games faster
- Shogi – without putting pieces into play from hand

7.2 Results

In this section I will present the results of the experiments.

7.2.1 Comparison of base evaluators

To figure out which of the base evaluators gives us the best estimate of the relative values of fairy chess pieces, I had them play multiple games against each other using different position evaluators.

Specifically, each combined evaluator mentioned above played 4 games of each variant against two opponents that used an alternative static evaluator and the same position evaluator, making it 420 games in total with each base evaluator present in two thirds of them. After each game, the engines were awarded 1 point for a victory, half a point for a draw and 0 points for a loss. The results can be seen in table 7.1.

Full breakdown of the games played and their results for each engine pair can be found in attachments 54.

Evaluator	Wins / Games played
SME	160 / 280
SPE	133.5 / 280
SCE	126.5 / 280

Table 7.1: Results of games with different base evaluators

Engine	Wins
By itself	53 / 120
Position Mobility	78 / 120
King Safety	63.5 / 120
Simple King Safety	32.5 / 120
Immediate Mobility	91.5 / 120
Immediate Captures	43.5 / 120
Position Soundness	58 / 120

Table 7.2: Results of games with different position evaluators

7.2.2 Comparison of position evaluators

In this experiment, seven variations of Static Mobility Evaluator were used:

- Static Mobility Evaluator without any enhancement
- Static Mobility Evaluator + Position Mobility Evaluator scaled by 1/100
- Static Mobility Evaluator + King Safety Evaluator scaled by 1/100
- Static Mobility Evaluator + Simple King Safety Evaluator
- Static Mobility Evaluator + Immediate Mobility Evaluator
- Static Mobility Evaluator + Immediate Captures Evaluator
- Static Mobility Evaluator + Position Soundness Evaluator

ehe scalings for Position Mobility Evaluator and King Safety Evaluator were chosen somewhat arbitrarily – using them without downscaling led to a very reckless styl of play where they'd sacrifice a lot of material for very slight positional benefit. Each played 4 games of each variant against every evaluator in the group. Results can be seen in table 7.2.

Full breakdown of the games played and their results for each engine pair can be found in attachments 58.

7.2.3 Speed of evaluator-iterator pairs

In this experiment, I let each evaluator-iterator pair in the following table play first three moves in a game of fide chess (my moves were always e4 first, followed by developing the queen onto an unattacked square and then moving it to another unattacked square) and measured the average branching factor of each node that

Engine/Iterator	Base	Monotonous	Monotonous faster	Strong piece	Capture move
SME by itself	13,14,13	8,7,7	8,6,5	12,11,12	11,8,7
SME position mobility	9,10,11	6,5,6	6,5,6	9,7,8	5,4,4
SME King safety	8,8,11	5,5,6	4,3,3	9,8,8	5,5,5
SME Simple King safety	11,12,17	8,8,8	7,4,5	11,12,13	10,5,6
SME ImmediateMobility	10,15,15	7,10,9	6,5,5	6,8,7	6,6,5
SME ImmediateCaptures	8,8,10	7,8,8	5,4,4	8,9,11	7,5,4
SME PositionSoundness	10,15,15	7,10,8	6,5,5	6,8,7	6,6,5
Average	11.57	7.29	5.10	9.05	5.95

Table 7.3: Branching factors of iterator-engine pairs on each move

was searched as well as the actual time it took the engine to play its three moves. The branching factor can be seen in table 7.3 and the resulting wall times in table 7.2.3.

Engine/Iterator	Base	Monotonous	Monotonous faster	Strong piece	Capture move
SME by itself	58s	44s	21s	35s	28s
SME position mobility	40s	36s	20s	20s	21s
SME King safety	400s	640s	54s	192s	109s
SME Simple King safety	78s	58s	20s	64s	23s
SME ImmediateMobility	400s	676s	70s	150s	79s
SME ImmediateCaptures	210s	460s	51s	160s	81s
SME PositionSoundness	5028s	8100s	1320s	1745s	1200s

Table 7.4: Wall speeds of iterator-engine pairs over all three moves

7.3 Discussion

7.3.1 Comparison of evaluators

According to the results shown in table 7.1, the base evaluators seem relatively evenly matched, with the mobility one having slight edge over its counterparts.

As shown in table 7.2, all enhancements of the base evaluator except Simple King Safety and Immediate Captures have shown to give improvements over just using the base evaluator. The most successful ones are ones focused on mobility – the ImmediateMobility evaluator actually counting the moves each player can play in a position and the PositionMobility evaluator estimating their number based on where each piece is placed, echoing the idea that mobility is the most important factor in chess-like games.

The differences between combined evaluators differing in base evaluators are much smaller than those between combinations differing in position iterators, likely because while different base evaluators are relatively similar (they all work by counting pieces, just differing in how they estimate value of each of them), the approaches employed by position evaluators differ vastly from each other.

7.3.2 Speed of iterator-evaluator pairs

The results presented in tables 7.3 and 7.2.3 show that all iterator heuristics managed to lower the branching factor compared to the baseline, with Faster

Monotonous Iterator providing the most significant improvement, followed closely by Capture Move Iterator and then after a bigger gap by Monotonous iterator and Strong Piece iterator.

The impact on wall speed of using a smart iterator has proved quite significant – the Capture Move and Monotonous Faster iterators have both increased the playing speed of all engines at least twice. The Strong Piece iterator is somewhat less effective, but it still delivers significant improvement. The Monotonous Iterator is the only one that has a negative effect on wall speed in some cases.

It seems that while the Monotonous Iterator reduces the branching factor over naive iteration, its overhead for evaluators with more expensive static evaluation is so high it actually worsens the wall-time measurement. It also doesn't seem to reduce branching more than its faster variant that ignores non-cumulative part of evaluation. A plausible explanation may be that the non-cumulative part is less likely to move monotonously over the course of a game than the cumulative part.

Conclusion

In this work, we have described popular variants of chess and fairy chess, identified some of their common characteristics and defined a notation that can describe large subset of theirs.

Then we implemented an engine capable of understanding the defined notation and playing the games described it in a way that allowed easy extensibility and experimentation with its evaluation function.

Next, we built variants of the aforementioned engine and used them in games against each other to test which of the evaluation function concepts are best to figure out a winning strategy. Results suggest the most important thing is piece mobility, both with regards to evaluating individual pieces as well as to evaluating a position of equal material. Experiments with iteration heuristics hinted that approaches using static evaluation of the intermediate nodes or notions strongly linked to it, such as captures of opposing pieces, are the most useful in terms of maximizing the benefit of A/b pruning.

The main drawbacks include the fact that the engine is rather slow and therefore it has to use rather shallow depths to play at a reasonable speed – either more powerful pruning heuristic have to be used (engines such as Stockfish go as low as 1.5 [16] compared to around 5 of our engine with the most successful iteration heuristics tried) or the engine's representation of game rules and subsequently its way of using it to expand game tree nodes would have to be reworked to allow faster operation.

Due to the sheer number of possibilities given by the CombinedEvaluator, there is a lot of space for future work – it could be quite interesting to see what benefit – if any – could more complex combinations of evaluation functions yield over the tuples of one base evaluator and one position evaluator that were tested as well as to use ScaledEvaluators in a more purposeful manner to find out what weights of the partial evaluators provide the best performance.

List of Figures

List of Tables

A. Attachments

The repository with code for the engine used in this thesis can be found at https://gitlab.mff.cuni.cz/teaching/nprg045/peskova/bp-otmar.

A.1 Games between engine pairs - different base evaluators

In this section, detailed results of each match between a pair of engines are shown. Each pair played four games of each variant with each engine playing twice as black and twice as white once with depth 4 and then with depth 5. Each engine was awarded 1 point for each victory, half a point for each draw and 0 points for each loss.

Game	Wins SME	Wins SCE
FIDE chess	2.5	1.5
Nutty Knights	2	2
Colorbound Clobberers	2.5	1.5
Locust chess	2	2
Shogi	1.5	2.5

Figure A.1: Results SME vs SCE

Game	Wins SME	Wins SPE
FIDE chess	1.5	2.5
Nutty Knights	2	2
Colorbound Clobberers	2	2
Locust chess	2	2
Shogi	1	3

Figure A.2: Results SME vs SPE

Game	Wins ScE	Wins SPE
FIDE chess	2.5	1.5
Nutty Knights	2.5	1.5
Colorbound Clobberers	1	3
Locust chess	1	3
Shogi	1	3

Figure A.3: Results SCE vs SPE

Game	Wins Combined ME	Wins Combined CE
FIDE chess	3	1
Nutty Knights	3	1
Colorbound Clobberers	4	0
Locust chess	1	3
Shogi	2	2

Figure A.4: Results Combined ME vs Combined CE

Game	Wins Combined ME	Wins Combined PE
FIDE chess	3	1
Nutty Knights	3	1
Colorbound Clobberers	2.5	1.5
Locust chess	1	3
Shogi	1	3

Figure A.5: Results Combined ME vs Combined PE

Game	Wins Combined CE	Wins Combined PE
FIDE chess	3	1
Nutty Knights	1	3
Colorbound Clobberers	2.5	1.5
Locust chess	2	2
Shogi	2	2

Figure A.6: Results Combined PE vs Combined CE

Game	Wins SCE + ImmM	Wins SME + ImmM
FIDE chess	2	2
Nutty Knights	2	2
Colorbound Clobberers	1.5	2.5
Locust chess	0	4
Shogi	2	2

Figure A.7: Results ImmM CE vs ImmM ME

Game	Wins SME + ImmM	Wins SPE + ImmM
FIDE chess	2	2
Nutty Knights	4	0
Colorbound Clobberers	1.5	2.5
Locust chess	2	2
Shogi	2	2

Figure A.8: Results ImmM ME vs ImmM PE

Game	Wins SCE + ImmM	Wins SPE + ImmM
FIDE chess	3.5	0.5
Nutty Knights	1.5	2.5
Colorbound Clobberers	1.5	2.5
Locust chess	1	3
Shogi	1.5	2.5

Figure A.9: Results ImmM CE vs ImmM PE

Game	Wins SCE + King	Wins SPE + King
FIDE chess	3.5	0.5
Nutty Knights	2	2
Colorbound Clobberers	2.5	1.5
Locust chess	1	3
Shogi	2.5	1.5

Figure A.10: Results King CE vs King PE

Game	Wins SME + King	Wins SPE + King
FIDE chess	2.5	1.5
Nutty Knights	2	2
Colorbound Clobberers	2.5	1.5
Locust chess	2	2
Shogi	3.5	0.5

Figure A.11: Results King ME vs King PE

Game	Wins SCE + King	Wins SME + King
FIDE chess	0.5	3.5
Nutty Knights	2.5	1.5
Colorbound Clobberers	0.5	3.5
Locust chess	2	2
Shogi	3	1

Figure A.12: Results King CE vs King ME

Game	Wins SME	Wins SCE
FIDE chess	3	1
Nutty Knights	3	1
Colorbound Clobberers	2.5	1.5
Locust chess	2	2
Shogi	1	3

Figure A.13: Results SimpleKing ME vs SimpleKing CE

Game	Wins SPE	Wins SCE
FIDE chess	1.5	2.5
Nutty Knights	3	1
Colorbound Clobberers	2	2
Locust chess	4	0
Shogi	2	2

Figure A.14: Results SimpleKing PE vs SimpleKing CE

Game	Wins SME	Wins SPE
FIDE chess	2.5	1.5
Nutty Knights	3	1
Colorbound Clobberers	2	2
Locust chess	2	2
Shogi	1.5	2.5

Figure A.15: Results SimpleKing ME vs SimpleKing PE

Game	Wins SME	Wins SCE
FIDE chess	2.5	1.5
Nutty Knights	3	1
Colorbound Clobberers	2	2
Locust chess	2	2
Shogi	1	3

Figure A.16: Results Position Soundness ME vs PositionSoundness CE

Game	Wins SME	Wins SPE
FIDE chess	2.5	1.5
Nutty Knights	3	1
Colorbound Clobberers	3.5	0.5
Locust chess	4	0
Shogi	2	2

Figure A.17: Results Position Soundness ME vs PositionSoundness PE

Game	Wins SPE	Wins SCE
FIDE chess	0.5	3.5
Nutty Knights	1.5	2.5
Colorbound Clobberers	1.5	2.5
Locust chess	2	2
Shogi	2.5	1.5

Figure A.18: Results Position Soundness PE vs PositionSoundness CE

Game	Wins SME	Wins SCE
FIDE chess	3	1
Nutty Knights	2	2
Colorbound Clobberers	2.5	1.5
Locust chess	1	3
Shogi	2.5	1.5

Figure A.19: Results ImmC ME vs ImmC CE

Game	Wins SME	Wins SPE
FIDE chess	2.5	1.5
Nutty Knights	2.5	1.5
Colorbound Clobberers	3	1
Locust chess	1	3
Shogi	1.5	2.5

Figure A.20: Results ImmC ME vs ImmC PE

Game	Wins SPE	Wins SCE
FIDE chess	3	1
Nutty Knights	2	2
Colorbound Clobberers	1.5	2.5
Locust chess	3	1
Shogi	2	2

Figure A.21: Results ImmC PE vs ImmC CE

A.2 Games between engine pairs - different position evaluators

Game	Wins SME	Wins SME+King
FIDE chess	2.5	1.5
Nutty Knights	2	2
Colorbound Clobberers 0.5	3.5	
Locust chess	2	2
Shogi	2.5	1.5

Figure A.22: Results Base vs King

Game	Wins SME	Wins SME+SimpleKing
FIDE chess	2.5	1.5
Nutty Knights	2	2
Colorbound Clobberers	2	2
Locust chess	3	1
Shogi	3	1

Figure A.23: Results Base vs Simple King

Game	Wins SME	Wins 100:1 SME+PME
FIDE chess	1	3
Nutty Knights	0.5	3.5
Colorbound Clobberers	0.5	3.5
Locust chess	2	2
Shogi	1.5	2.5

Figure A.24: Results Base vs Combined

Game	Wins SME	Wins SME + ImmM
FIDE chess	1	3
Nutty Knights	0	4
Colorbound Clobberers	0.5	3.5
Locust chess	2	2
Shogi	1	3

Figure A.25: Results Base vs ImmM

Game	Wins SME	Wins SME + ImmC
FIDE chess	2	2
Nutty Knights	2.5	1.5
Colorbound Clobberers	2	2
Locust chess	3	1
Shogi	2	2

Figure A.26: Results Base vs ImmC

Game	Wins 100:1	Wins SME + SimpleKing
FIDE chess	1.5	2.5
Nutty Knights	3	1
Colorbound Clobberers	2	2
Locust chess	4	0
Shogi	4	0

Figure A.27: Results Combined vs SimpleKing

Game	Wins 100:1	Wins SME + King
FIDE chess	3	1
Nutty Knights	3.5	0.5
Colorbound Clobberers	3	1
Locust chess	3	1
Shogi	2	2

Figure A.28: Results Combined vs King

Game	Wins 100:1	Wins SME + ImmM
FIDE chess	1	3
Nutty Knights	1.5	2.5
Colorbound Clobberers	1.5	2.5
Locust chess	2	2
Shogi	2.5	1.5

Figure A.29: Results Combined vs ImmM

Game	Wins MixedMEE	Wins SME + ImmC
FIDE chess	3.5	0.5
Nutty Knights	3.5	0.5
Colorbound Clobberers	1	3
Locust chess	3	1
Shogi	3.5	0.5

Figure A.30: Results Combined vs ImmC

Game	Wins SME + SimpleKing	Wins SME + King
FIDE chess	1	3
Nutty Knights	0	4
Colorbound Clobberers	2	2
Locust chess	3	1
Shogi	0.5	3.5

Figure A.31: Results SimpleKing vs King

Game	Wins SME + SimpleKing	Wins SME + ImmM
FIDE chess	0	4
Nutty Knights	0	4
Colorbound Clobberers	1.5	2.5
Locust chess	1	3
Shogi	1.5	2.5

Figure A.32: Results SimpleKing vs ImmM

Game	Wins SME + SimpleKing	Wins SME + ImmC
FIDE chess	0.5	3.5
Nutty Knights	2	2
Colorbound Clobberers	1.5	2.5
Locust chess	3	1
Shogi	2	2

Figure A.33: Results SimpleKing vs ImmC

Game	Wins SME + King	Wins SME + ImmM
FIDE chess	1.5	2.5
Nutty Knights	1	3
Colorbound Clobberers	1	3
Locust chess	3	1
Shogi	1.5	2.5

Figure A.34: Results King vs ImmM

Game	Wins SME + King	Wins SME + ImmC
FIDE chess	3.5	0.5
Nutty Knights	2	2
Colorbound Clobberers	2	2
Locust chess	3	1
Shogi	3.5	0.5

Figure A.35: Results King vs ImmC

Game	Wins SME + ImmM	Wins SME + ImmC
FIDE chess	3.5	0.5
Nutty Knights	1.5	2.5
Colorbound Clobberers	2.5	1.5
Locust chess	4	0
Shogi	2.5	1.5

Figure A.36: Results ImmM vs ImmC

Game	Wins SME	Wins SMEPS
FIDE chess	0.5	3.5
Nutty Knights	0	4
Colorbound Clobberers	2.5	1.5
Locust chess	4	0
Shogi	2.5	1.5

Figure A.37: Results Base vs Position Soundness

Game	Wins 100:1	Wins SMEPS
FIDE chess	2	2
Nutty Knights	3.5	0.5
Colorbound Clobberers	2.5	1.5
Locust chess	2	2
Shogi	1.5	2.5

Figure A.38: Results Combined vs Position Soundness

Game	Wins ImmC	Wins SMEPS
FIDE chess	0.5	3.5
Nutty Knights	3.5	0.5
Colorbound Clobberers	1.5	2.5
Locust chess	0	4
Shogi	1	3

Figure A.39: Results ImmC vs Position Soundness

Game	Wins ImmM	Wins SMEPS
FIDE chess	2	2
Nutty Knights	2	2
Colorbound Clobberers	2	2
Locust chess	2	2
Shogi	2.5	1.5

Figure A.40: Results ImmM vs Position Soundness

Game	Wins SME + King	Wins SMEPS
FIDE chess	2.5	1.5
Nutty Knights	1	3
Colorbound Clobberers	3	1
Locust chess	4	0
Shogi	1.5	2.5

Figure A.41: Results King vs Position Soundness

Game	Wins SME + SimpleKing	Wins SMEPS
FIDE chess	0	4
Nutty Knights	1	1
Colorbound Clobberers	1.5	2.5
Locust chess	3	1
Shogi	0.5	3.5

Figure A.42: Results Simple King vs Position Soundness