Check Norris - a chess engine

Jakob Nordgren, Magnus Björck, Ngan Ha Pham, Aljaz Kovac

February 2020

Contents

1 Project Description

1.1 Project Summary

To sharpen our Haskell programming skills we have decided to make a chess engine that evaluates possible moves and chooses the best one in the given position. The original goal of the project was to make a playable chess board which would observe all of the rules of classical chess and enable two human players to play against each other. But soon our appetite grew and as we became more confident we decided to try and raise the bar and make a functioning chess engine: an engine that we would be able to plug into the Winboard interface protocol and allow it to play against another engine, such as perhaps Stockfish of different levels, or to let it play against itself or other human opponents. We understood from the beginning that not only would the engine itself be a challenge, but also making it communicate appropriately with the Winboard protocol could prove to be tricky. In the end, the whole bundle of challenges turned out to be quite a mountain to climb and it took many hours of dedicated work. We wanted our program to play against sophisticated software, so we needed to try and implement all the rules of the game, including the more tricky ones such as castling, "en passant" and promotion. Then there are other special chess positions and situations that one needs to take into account when constructing a chess engine, such as stalemate, draw by threefold repetition, and the 50 move draw rule. Unfortunately, due to time constraints, we were not able to implement all of these special moves and situations: "en passant", stalemate and draw by repetition remain to be implemented, but we have managed to incorporate castling and promotion. Despite those shortcomings, we have managed to build a functioning chess engine that can play against another chess engine, against itself, or a human opponent, but needs some finishing touches when it comes to the implementation of the aforementioned special chess rules and positions. So, how did we go about building this thing? We divided our workflow into three branches: the building of the data structures and types; the writing of all helper functions; the building of the engine's architecture. Our plan was to follow these basic steps for building a chess AI: move generation, board evaluation, and the minimax algorithm for finding the best possible move in a search tree. The following report details our build process and the inner workings of our product. After two weeks of blood, sweat and tears, we are happy to announce a new engine in the world of chess: The Check Norris!

1.2 A Quick Background Check

This part is intended for those readers that do not have much experience with the game of chess. Here follows a quick explanation of the basic and more complicated rules of chess, special moves, important chess terminology, etc. that should aid the reader of this report, and help them understand the principles of our chess engine, its performance as well as its constraints.

1.2.1 Chess rules, from simple to advanced

Chess is an ancient game played on a 8 x 8 board with two sets of pieces, black and white. Each set consists of 8 pawns, two knights, two bishops, two rooks, a queen and a king. All of these pieces can move in their own special ways. All of these pieces can be captured, except for the king, which is the most powerful but also the most vulnerable piece on the board. The ultimate goal of chess is to checkmate the opponent's king, which is what happens when the king has no valid square to move to and is put in check by another piece (threatening to be captured on the next move).

Here is an overview of a few special chess moves and positions:

1. CHECK: the king is under attack by an enemy piece, and will be captured on the next move if not moved out of check; should the king not move out of check on the next move, the king is lost and with it the game.

2. CASTLING: a special move involving the king and the rook, provided that neither had made a move yet. The king moves two squares towards the rook, and the rook moves to the other side of the king. Castling can be done both on the king side and the queen side.

3. PROMOTION: if a pawn manages to reach the end of the board, then it can be promoted into any other piece.

4. "EN PASSANT": pawns can move in the following ways; one or two squares on the first move, then one square on all the other moves. They capture diagonally, but if a pawn makes a two-square move on its first move and comes side-by-side with an enemy pawn, then that pawn can capture it sideways on the very next move (and only on that move).

5. DRAW BY REPETITION: if the same moves are repeated three times by both sides, then the game is automatically a draw.

6. 50 MOVE DRAW RULE: if the game does not produce a winner after 50 moves, it is a draw.

7. STALEMATE: if the active player has no legal moves, but is at the same time not put in check by an enemy piece, the result is a draw.

8. CHECKMATE: the king is in check and has no valid square to move to. Game over!

1.2.2 Chess terminology

1. RANKS AND FILES: a chess board is divided into ranks and files: ranks are rows that run from side to side and are marked with numbers from 1 to 8; files are columns that run vertically and are marked with letters from "a" to "h".

2. SQUARES: a square is where a rank and a file meet, for example "a8" or "c3".

3. CECP or Chess Engine Communication Protocol: CECP is an open communication protocol that enables chess engines to communicate with each other. Chess engines can be "plugged into" the CECP and play games against each other, or have human opponents play games against them. The protocol uses standard input/output to transmit moves, clock information for timed games and position evaluation.

4. FORSYTH-EDWARDS NOTATION (FEN): A widely used notation for describing a chess position. A FEN string contains six fields, separated by spaces. For example the FEN string representing the starting position would be

   ```
   "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"
   ```

   where the first field lists all ranks of the chess board, separated by '/'. Lowercase characters represents black pieces, uppercase characters represent white pieces, and numbers represent sequences of empty squares. The second field indicates whose turn it is, 'w' means white to move and 'b' means black to move. The remaining fields are used to present information about castling availability, possible "en passant" moves, the number of moves in the game, and the number of moves since the last capture.

2 Program documentation

2.1 Overview

Our program consists of two major parts, the engine and the CECP interface. The engine provides functionality for keeping track of chess positions as well as generating and searching for possible moves to play. The interface is responsible for communicating with the host application and keeping track of the current game state, i.e. who plays which side, when a game is restarted etc. The program in divided into four modules, organized in a hierarchical structure (Fig. 1). Below is a broad overview of the vital parts of the program.

2.1.1 Board Representation

We chose to represent a chess board as a matrix using a list structure, where each index maps to a certain square. This would allow us to implement piece movement as simple arithmetic operations. The problem with this approach is how to detect when pieces move outside of the board, and preventing pieces from magically jumping from one edge of the board to the other. For this reason, we went with the well known method of putting our board inside a 12x10 matrix (Fig. 2). This way, during move validation, it is just a matter of checking if the

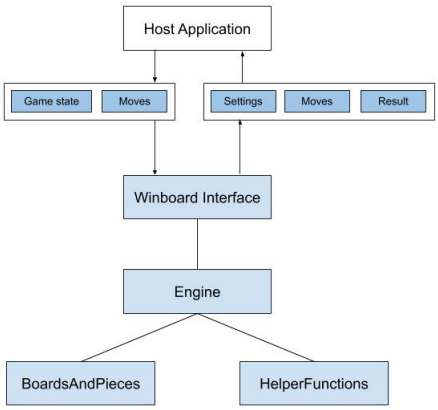

Figure 1: Overview of the module hierarchy, and how the Engine and Host application fit together

piece landed on a valid square. The data type representing the board matrix uses a regular list, containing other custom data types.

2.1.2 Move Generation and Validation

Moves are generated in three major steps. First we create a list of all "pseudo-legal" moves, i.e. moves that would obey the rules for piece motion, but that may put the own king in check. This is done by iterating through the squares of the board, ignoring squares that contain the active player's own pieces (Fig. 3). When a square is either empty or is occupied by an enemy piece, we then search through consecutive squares in all directions and the squares from which a knight could reach our square, until we either find one of the active player's pieces or an "Out" square. If we find one of the active player's pieces, we check if the direction we searched is a valid direction for that piece type. If so, we have found a possible legal move and add it to our list of pseudo-legal moves. If we encounter an "Out" square, we go back to the starting square and continue searching in the next direction. Since pawns and kings only can move a fixed number of squares in any direction, we generate moves for them separately, by simply searching the relevant squares around them.

Once we have our list of pseudo-legal moves, we filter out the moves that would put the active player's king in check. We detect a check by simply searching the square occupied by the king in the same manner described above.

Validation of input moves is simply a matter of generating the move list and checking if the input move is present in that list.

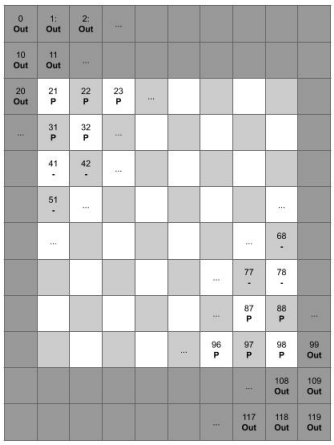

Figure 2: The representation of a chess board inside a 12x10 matrix with index [0...119] where each element has a value representing an off-board square ("Out"), an empty square on the board ("-") or a square on the board occupied by a piece ("P").

2.1.3 Position Evaluation

In order to choose good moves to play, we must first be able to somehow evaluate the positions that arise from playing different moves. There are countless books and theories about what factors make a chess position better or worse for one side. In most cases though, the single most important consideration is that of material balance i.e. which side has the strongest force at hand. Not all pieces are equally valued, and typically pieces are assigned values in terms of pawns. The queen is generally considered to be worth the equivalence of 9 pawns, the rook is worth 5 pawns and the knight and bishop are worth 3 pawns respectively. In our program, the king is assigned a value of 100 pawns, to safe-guard against computing lines involving sacrificing your own king! We subtract the total of black's material from the total of white's material, so a positive score means white is better.

Only considering the material though, we found, leads to quite static and boring play, especially during the opening phase before any pieces have been captured. Therefore we also take into account the number of possible moves each side can make and compute the difference, where again, a positive value means white has more active pieces and is therefore better. This is a way to measure piece mobility, favouring moves that put the players pieces on squares where they cover more territory.

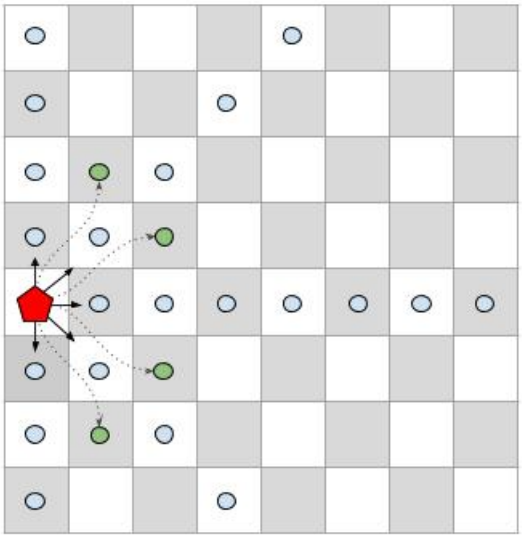

Figure 3: Search square - The red shape represents a "super piece" which can move like a queen and knight combined. Each direction is recursively searched (blue dots) until a piece or board edge is found, in which case a move can be generated if a piece of the right type and color is found. The green dots are searched for knights of the right color. Each square not occupied by the active player's own forces is searched in this manner and possible moves are added to the move list accordingly.

2.1.4 Building and Searching the Move Tree

When the engine is asked to make a move, we build up a game tree consisting of all legal moves in a given position down to a certain depth. We can then traverse the tree, evaluating the resulting positions of each leaf, and simply choose the path that leads to the leaf with the best evaluation. There is a problem with this approach though, which is that we must assume that our opponent makes good counter moves, and will probably not choose the path that leads to the highest evaluation. To solve this, we implemented a minmax algorithm, which is used to search the tree. Simplified, the minmax algorithm alternates between selecting the best continuation and selecting the worst continuation (Fig. 4). This way, we end up with the best position we can get, assuming our opponent makes the best counter moves.

We build the move tree by first generating all legal moves in a given position, putting each one in a new tree which we store in a list. We then iterate through all trees, generating new move trees, and adding them as child nodes. This way, we can build a complete game tree down to a given depth.

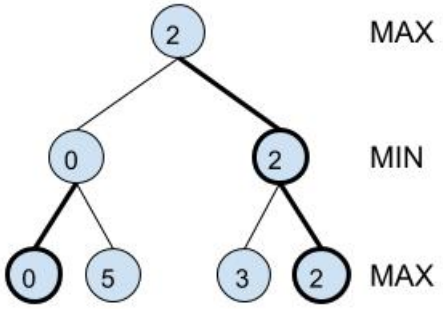

Figure 4: MinMax Algorithm, the maximising player chooses to go down the rightmost path, even though the leaf with the highest score is found to the left.

2.1.5 Communicating with the host application

This program is not meant to be executed as a standalone, but instead loaded into any application, graphical or otherwise, that supports the CECP. The interface used for this program is the minimum set of commands necessary to make the engine play inside for example winboard/xboard without interruption. CECP uses standard input and output to send text-based commands back and forth between the host and engines. The following is a brief description of the implemented commands sent from the host to the engine:

protover 2 Sent by the host when the engine is loaded, and means that the host is ready to accept "feature" commands from the engine

USERMOVE Relays a move made by the user, for example "e2e4" or "a7a8Q"

new Tells the engine to reset the game to the starting position, start accepting moves and playing as black

go Tells the engine to make a move on the current board, as whatever side's move it is, and then continue playing that side

force Tells the engine to stop making moves, but keep the current position and continue accepting moves

quit When this command is received, the engine should halt immediately

The commands sent from the engine to the host:

feature done=0 Tells the host to wait for more feature commands

feature myname=NAME Tells the host the name of our engine

feature sigint=0 sigterm=0 Disable the usage of UNIX signals, which can cause unexpected interruptions in UNIX systems

feature done=1 Tells the host that the engine is done sending feature commands

move MOVE Relays a move from the engine, for example "move e2e4"

RESULT message When checkmate is detected a result string is sent, for example "1-0 White wins"

For the sake of completeness, here is an abbreviated example of communication between winboard and the engine.

```
Host                Engine

> xboard
> protover 2
                    > feature done=0 myname="engine"
> accepted done
> accepted myname
                    > feature done=1
> accepted done
> new
> force
> e2e4
> go
                    > move e7e5
> g1f3
                    > move b8c6
```

2.2 BoardAndPieces.hs

The module BoardAndPieces is where the board and pieces are handled. It consists of data types that represent a chess board and the pieces on the board. It also consists of functions to create a board (createStartBoard), a function to communicate with the Engine module and a function to move a piece on the board.

2.2.1 Data types

To represent the colors of the pieces and whose turn it is, the data type Color was implemented. Color has two constructors, Black that represents the color black and White that represents the color white.

```
data Color = White | Black
```

To represent the six different pieces the data type PieceType was implemented. PieceType consists of six constructors: Pawn to represent pawns, Rook to represent rooks, Knight to represent knights, Bishop to represent bishops, Queen to represent queens and King to represent kings.

```
data PieceType = Pawn | Rook | Knight | Bishop | Queen | King
```

A chess piece consists of a color and what type it is, and therefore the data type Piece, which represents a chess piece, and consists of the data types Color and PieceType.

```
data Piece = Piece {c::Color t::PieceType}
```

To represent the squares on the board the data type Square was implemented. It consists of four constructors: Empty which represents an empty square on the board, Out which represents a square outside of the board, Ps Piece which represents a piece on the board and SquareError which is used to show that an error has occurred.

```
data Square = Empty | Out | Ps {p::Piece} | SquareError
```

To be able to keep track of what square is being handled the type Index was implemented. Index consist of an Int.

```
type Index = Int
```

The type Board represents the chess board and consists of a tuple which contains a list of tuples of Index and Square, and Color. The list of tuples of Index and Square is the actual board, while Color represents whose turn it is. Index shows what position an element in the list has and Square represents a square on the board.

```
type Board = ([(Index,Square)],Color)
```

To make it easier to keep track of what was handled in a function the type BoardSquares was implemented. It consists of a list of tuples that contain Index and Square.

```
type BoardSquares = [(Index,Square)]
```

The fromFEN function in Engine handled a certain list of strings and the functions in BoardAndPieces handled Board, therefore the type EngineString was implemented to make it easier to read the code.

```
type EngineString = [String]
```

To make it easier to read the code the type Move was implemented. Move represents a move on the board and consists of a tuple of Index.

```
type Move = (Index, Index)
```

ListOfMoves was implemented to store Move. ListOfMoves consists of a list of Move.

```
type ListOfMoves = [Move]
```

2.2.2 Functions

The functions in BoardAndPieces revolve around handling the Board and the Pieces on it. There are five functions in BoardAndPieces where two are internal functions. The most important function is the move function.

```
move :: Board -> Move -> Maybe Board
```

The `move` function takes a Board and a Move (in the form of (i,i') where i is the piece's current square, and i' is the square it will be moved to) and then through the use of auxiliary functions moves a piece from one "square" to another. `move` returns a Maybe Board with changed turn color and moved piece. The function first checks what kind of move it should make. The auxiliary functions `isCastling`, `isPassant` and `isPromotion` are used to see if there should be a special move. `isCastling` takes a square and a move, and returns a Bool. There are only four possible ways to castle, and `isCastling` checks if it is one of these. If `isCastling` returns `True` then the function `moveCastling` is used, if it returns `False` it is not.

```
isCastling :: Square -> Move -> Bool
```

The functions `isPassant` and `isPromotion` both work in the same way. They take a Move and return a Bool. If there is an "en passant" move or a promotion then there is a special "code" in i'. If the first three digits of the i' are 110 or 111 then we have an "en passant" move and the function returns `True`, and the `movePassant` function is used. 110 means it is a white piece making the move and 111 means that it is a black piece making the move. If the digits are 120, 121, 122, or 123 then there is a promotion move and the function returns `True`, and the `makePromotion` function is used. 120, 121, 122 and 123 correspond to the wanted promotion. 120 is a rook, 121 is a knight, 122 is a bishop and 123 is a queen.

```
isPassant :: Move -> Bool
isPromotion :: Move -> Bool
```

If all of the "is" functions return `False` there is no special move and the piece is moved with the use of the auxiliary functions `move'` and `move''`.

```
move' :: Board -> Move -> Square -> BoardSquares -> Board
move'' :: Board -> Move -> BoardSquares -> Board
```

`move'` sets the square at index i' to be the same as square at i and then calls on `move''`. `move''`sets the square on index i to be empty and then returns a changed Board.

```
moveCastling :: Board -> Move -> Board
movePassant :: Board -> Move -> Board
makePromotion :: Board -> Move -> Board
```

`moveCastling`, `movePassant`, and `makeePromotion` also use `move'` and `move''`, although in a different way than a standard move. `moveCastling` uses `move'` twice, once for the king and once for the rook. `movePassant` uses `move'` to move the pawn and a extra `move''` to set the pawn it passes square to Empty. `makePromotion` uses move in a standard way but changes the square depending on the wanted promotion.

```
createStartBoard :: Board
```

`createStarBoard` creates a board with the pieces in starting position.

```
fromBoardToString :: Board -> EngineString
```

`fromBoardToString` transforms a Board into an EngineString.

```
takeSquare :: Board -> Index -> Square
```

`takeSquare` returns a certain square at index i in a board.

```
changeTurnColor :: Board -> Board
```

`changeTurnColor` is used to change the Color in a Board. If the input has White as the color then the function returns the Board with the color Black, and the other way if the input is a Board with the color Black.

2.3 HelperFunctions.hs

The module `HelperFunctions` consists of utility functions used by various parts of the program. It consists of the following functions:

`squareToIndex :: String -> Int`

Converts a square coordinate to its corresponding index value in the 12x10 board matrix. Eg "e2" is converted to 85.

`indexToSquare :: Int -> String`

Converts an integer to the coordinate represented by that index in the 12x10 board matrix. For example, 85 is converted to "e2"

`countMaterial :: Board -> Color -> Int`

Counts the total value of one side's forces, and is called from `evaluatePosition` function inside `Engine.hs` . A pawn is valued at 100 points, bishops and knights are worth 300, rooks 500, and queens 900 points. Finally, the king is assigned a value of 10 000 points to prevent the engine from trying to sacrifice its own king.

```
boardLookupSquare :: Board -> Int -> Square
```

Returns the square of a certain index in a given `Board` .

```
isMove :: String -> Bool
```

Checks if a string is in the form of a move according to the CECP standard. `isMove` is from `WinboardInterface.hs` whenever a command is received from the host in order to decide what to do next.

2.4 Engine.hs

2.4.1 Data types

When searching for the next move, moves are generated and stored in a general tree data structure, which is then traversed by the minmax and maxmin functions to search for the best candidate move.

```
data MTree a =  MTree a [MTree a]
```

2.4.2 Move Generation

The backbone of the engine is the move generating function

```
generateMoves' :: Board -> [Move] -> [Move]
```

It generates a list of pseudo-legal moves for a given position. Note that the function takes a `Board` as well as a list of moves `[Moves]` . The list of moves is needed to check if castling or "en passant" moves are available. Pseudo-legal move generation simply means that this function will generate moves that may put the own king in check, which is of course illegal. The main function for generating moves simply calls `generateMoves'` and filters out moves that would leave the player's king in check, and looks like this:

```
generateMoves :: Board -> [Move] -> [Move]
generateMoves b moves = filter legal $ generateMoves' b moves
  where
    legal = not . isCheck . changeTurnColor . fromJust . (move b)
```

Moves are generated in three steps: moves for pieces and the king are generated first by calling the function `searchSquare` , which is the main algorithm for finding pseudo-legal moves, explained in more detail below. Then, pawn moves are generated, which include diagonal captures and "en passant". Finally, possible castling moves are searched for. This is the definition for `searchSquare` :

```
searchSquare :: (Index, Square) -> Board -> [Index]
```

Given a square at a certain index and a board, `searchSquare` iterates through all possible directions, recursively searching along a certain file, rank or diagonal. Since the board is just a 12x10 matrix represented as a linear list, stepping through each square in a direction is simply a matter of adding or subtracting a certain number from the current index. For example, here is a snippet from the function implementation:

```
search' (i, (Ps piece)) dir restart =
if c piece == sideToMove && validDir dir piece
    then [i]
    else if restart
        then search' (i, Empty) dir False
        else []

search' (i, Out) _ _                          = []

search' (i, Empty) 8 _ =      --Forward movement
    let nextSquare = boardLookupSquare searchBoard (i+10) in
        search' ((i+10), nextSquare) 8 False
```

The code demonstrates how all squares in the "upward" direction are recursively searched by searching the current index i+10, which corresponds to the square right above the current square, until either a piece or an "Out" square is found. The second argument of `search'` is an integer representing the direction to be searched.

Pawn moves are generated by iterating through all the active player's pawns and checking if the square in front of the pawn (or two squares, if the pawn is on the starting rank) is empty, and checking the two diagonal squares in front of the pawn for enemy pieces to generate captures. There is also the special move "en passant" which is generated if present.

Lastly, the position is checked for possible castling moves, which consists of checking the move list if neither the king nor the castling rook has moved before, and checking that no enemy piece is threatening one of the squares the king will travel through.

2.4.3 Search

The engine's algorithm for finding a move to play involves building a game tree by recursively generating moves down to a certain depth and then traversing the tree using the MinMax algorithm and a simple evaluation function for finding the best candidate move. The tree is built by first creating a forest, i.e. a list of single node MTrees. This is done in the function

```
createForest :: Board -> [Move] -> [Move] ->
    [MTree ((Board, [Move]), Int)]
```

The function simply takes a `Board` , a list of the played moves in the game, and a list of generated moves and puts each of the generated moves into a new `MTree` . The trees in the list can then be grown by passing the list into

```
growMTree :: [MTree ((Board, [Move]), Int)] ->
    [MTree ((Board, [Move]), Int)]
```

Which recursively generated a new forest for each of the moves in the list we pass in. By combining these functions we can create a game tree of arbitrary depth by calling

```
generateMTree :: Board -> [Move] -> Int ->
    [MTree ((Board, [Move]), Int)]
generateMTree b mvs 0 = createForest b mvs (generateMoves b mvs)
generateMTree b mvs n = growMTree (generateMTree b mvs (n-1))
```

Where we simply create a forest, and then recursively grow each tree n times. Because Haskell evaluation is lazy, even very deep trees can be generated this way. However, our program does not implement any type of pruning algorithm meaning that the whole tree must be searched each time. Given the rapid growth rate of the number of possibilities in a game of chess, it is only feasible for the program to generate a tree of height 2, which corresponds to one move by the engine, followed by one counter move from the opponent.

Once the tree is generated we traverse it using

```
minMax :: [MTree ((Board, [Move]), Int)] -> ((Board, [Move]), Int)
```

and

```
maxMin :: [MTree ((Board, [Move]), Int)] -> ((Board, [Move]), Int)
```

`minMax` and `maxMin` are mutually recursive, meaning they call each other to compute the correct result. For example, given a list of trees of depth n, minMax returns the move of highest evaluation if n=1, otherwise it returns the move of highest evaluation out of whatever moves maxMin returns when passed each of the subtrees with height of (n-1). Similarly, `maxMin` returns the move of lowest evaluation if n=1, otherwise it returns the move of lowest evaluation out of whatever `minMax` returns when passed each of the subtrees in the list.

The interfacing function for generating a move from a given position is

```
searchAndPlay :: Board -> [Move] -> (Board, [Move])
```

Which takes a board and the game moves, builds a search tree and uses the calls either the minMax or the maxMin function, depending on whether it is white (the maximising player) or black (the minimizing player) to move. Subsequently the found move is performed and the resulting board and list of game moves is returned. `searchAndPlay` generates a list of trees with height 2, meaning it only searches among its possible moves in the given position, and the opponent's responses to those moves. Greater search depths were just not feasible for this project, since the number of possible moves grows exponentially, and the Engine would not compute a move in an acceptable amount of time.

2.4.4 Evaluation

In order for the search algorithm to work, a function for evaluating resulting positions is needed. This program uses two metrics to score a position, material value and piece mobility. The evaluation function looks like this

```
evaluatePosition :: Board -> Int
evaluatePosition board =
    (countMaterial board White - countMaterial board Black)
    + (numberMovesSide board White - numberMovesSide board Black)
```

The function takes a board and returns a score, where positive values mean white is better. `counMaterial` is defined in the module `HelperFunctions.hs` and piece mobility is simply measured by computing the number of possible moves for each side:

```
numberMovesSide :: Board -> Color -> Int
numberMovesSide (pos, _) col = length $ generateMoves' (pos, col) []
```

It should be clear from looking at the definition for `countMaterial` that the lowest possible difference in material is 100 points (equivalent to a pawn) and, although perhaps theoretically possible, in practical play there will never be a difference of 100 possible moves between the players. Hence, `numberMovesSide` will only help to distinguish between moves of equal material value, promoting active piece play.

2.4.5 Utility functions

The remaining parts of the module are utility functions, used by various parts of the program, or for debugging purposes, and are listed below.

```
fromFEN :: String -> Board
```

Used to convert a string FEN string to a Board type. This function uses an abbreviated version of the FEN notation, only reading the first two fields of the FEN string. This function is actually not called in the program, but has been heavily utilized in testing and debugging. Also the program may be extended in the future to set up custom positions for playing or analyzing, a feature supported by CECP.

```
isCheck :: Board -> Bool
```

Used to detect if the active player's king is in check.

```
isCheckmate :: Board -> [Move] -> Bool
```

Used to detect if the active player has been mated

```
squareAttacked :: Board -> (Index, Square) -> Bool
```

Checks if a given square is within the reach of an enemy piece. Used primarily to look for check and when computing if castling is available.

```
validMove :: Board -> [Move] -> Move -> Bool
```

Checks if a given move would be legal. This is simply done by generating moves for the given position and checking if the input move is present in the generated move list.

```
fromJust :: Maybe a -> a
```

Used to extract a pure value a from a Maybe type.

2.5 WinboardInterface.hs

This is the main module meant to be compiled and loaded into a host application. It's job is to listen for commands from the standard input and send commands to the standard output. The commands are described in section 5.1.2, and the flowchart in Fig. 5 gives an overview of the flow of the program. The functions of the interface are:

```
main :: IO ()
```

The entry point of the program which calls the following function third parameter set to True:

```
communicate :: Board -> [Move] -> Bool -> IO ()
```

The main loop of the program. It reads a line from standard input and checks if it is a valid command, if not it just calls itself with the same parameters as passed in. If the command is recognised, the function acts accordingly to the specification in section 5.1.2. When a move command is received , it is first passed to `updateBoard` , then the resulting position is checked for checkmate. If the position is checkmate, the function `endGame` is called and the program enters an idle state, until triggered by the "new" command to setup a new game. If the command `"quit"`is recieved `communicate` returns `(),`terminating the program.

```
endGame :: Board -> IO ()
```

Prints a result string, telling the host that the game has ended. For example "1-0 White Wins". As of now, the engine does not handle stalemate or, draw by repetition.

```
updateBoard :: Board -> [Move] -> String -> (Board, [Move])
```

Updates the board with the move passed in as a string, by calling the function `move`inside `BoardAndPieces.hs`.

`makeMove :: Board -> [Move] -> IO (Board, [Move])`

This function is called when the host prompts the engine to make a move. It calls the function `searchAndPlay`from `Engine.hs`and, prints the resulting move to standard output and finally updates and returns the position with the new move performed.

`strToMove :: String -> Move`

Converts a move string to a `Move`type

`moveToStr :: Move -> String`

Converts a `Move`type to a move string

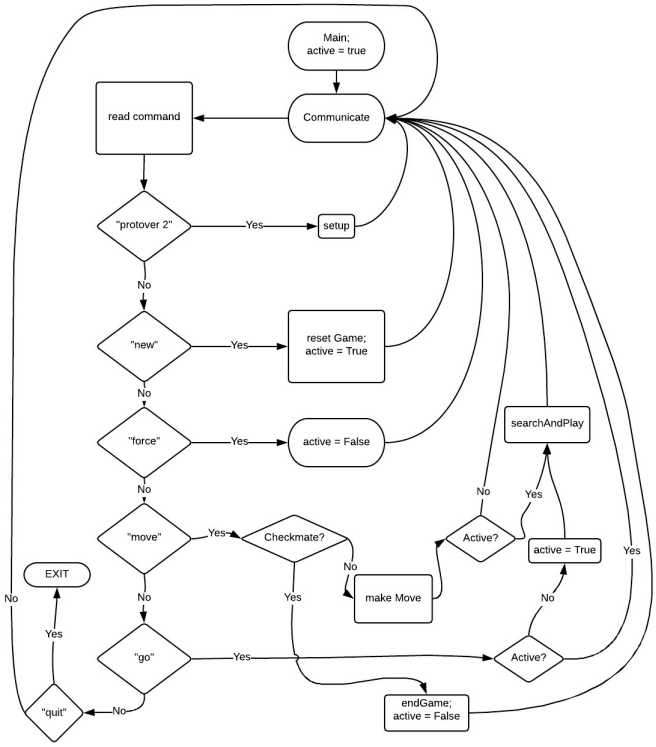

Figure 5: Overview of the flow of the program while running.

3 Usage

This program is meant to be loaded into some application that supports the Chess Engine Communication Protocol (CECP), also known as the Winboard/Xboard

protocol. This will allow the user to test the engine either by playing against it themselves, or by matching it against another engine of their choosing. However, it is also possible to manually type in CECP commands in the terminal to test the functionality, although this is not how the program is intended to be used and some functionality may be lacking.

3.1 Usage with CEPC compatible application

We suggest using xboard for Unix systems or it's MS-Windows port, Winboard. Xboard/Winboard comes with some pre-built engines that can easily be loaded into the GUI. The following are instructions for the windows operating system using Winboard, however any CECP compatible application should work. To load our engine into Winboard on Windows, please follow these steps:

1. Download and install Winboard from http://hgm.nubati.net/
2. Download the project and build using
   `ghc --make WinboardInterface.hs`
3. Open up the Winboard Game Viewer
4. Go to *Engine >Load First Engine...*
5. In the *Load first Engine* dialog, enter the full path to the WinboardInterface.exe file created in step 2, for example:
   `C:\Users\username\group14\WinboardInterface.exe`
   Or, alternatively, click the button to the left of the text box to select the file using the system dialog. Then press *OK*
6. To start the game, just make a move on the board, the Engine should respond with a counter move after a couple of seconds.
7. If you wish to test our engine against another engine, or even against itself, go to *Engine >Load Second Engine...* and select one of the engines in the list to the right, for example "Fairy-Max", or supply the path manually as in step 5. With both engines loaded, go to *Mode >Two Machines* or press `Ctrl+T`.

4 Conclusion

Although this chess engine has many of the core features necessary to further improve it's playing strength, most notably perhaps the move generating algorithm, it also has some design flaws, which would require attention. Firstly, in order to increase the search depth, some kind of pruning technique would probably be necessary, such as the alpha-beta algorithm. However, for pruning to be efficient, it is important that the program implements ordering of the generated moves, making it more likely that we search good candidate moves first. For example, captures and moves giving check may be considered before

"quiet" moves. And as of now, our datatype for representing moves cannot hold any extra information about the moves made, so that would probably be a good next step. Also, although it was not the main goal of this project to prove the correctness of our search algorithm, given more time, we would have liked to dive a bit deeper in testing the tree building and minMax algorithm. While we do think that this engine comes up with surprisingly realistic moves, given its shallow search depth and it's naive evaluation function, it occasionally produces unexpected moves, which we have had a hard time figuring out how they were calculated.

Furthermore, the move format in the CECP does not clearly state when a pawn captures "en passant", which is something we realized much too late. For example, a white pawn on the "e5" square capturing a pawn on the "d6" square would be sent as "e5e6", but the same string would be sent for the same pawn capturing a pawn on "d5" "en passant". This means that we would need to check each incoming move to detect EP captures. Unfortunately we had no quick way of doing this and decided to leave that feature aside for now. What this means is that our engine will generate and sometimes execute EP moves, which is fine, but if the opponent captures EP, this would not be recognized as a valid move by our engine, and the game would need to be reset.

Finally, we have not implemented any functionality for detecting drawn positions, i.e. stalemate, 3-move repetition or the 50-move rule.

Despite the shortcomings of our implementation, we feel very pleased with the fact that we have managed to build a functioning chess engine in just three weeks. With more time on our hands we could have improved both the interface and the search algorithm, and that is something that we aim to do in our free time in the near future. But the eager reader does not have to wait that long to play a chess game against Check Norris: feel free to download the files, load the engine into Winboard and play a game of chess. Good luck!