EECS 22L: Chess Software

Specification v2.0

Prepared by: Team rand()

Affiliation: UCI Henry Samueli School of Engineering

Developers:

Yongye Li

Mary Campbell

Arjun Sivakumar

Joseph Principe

Cristian Pina Bravo

Zijie Huang

Date:

April 24, 2023

Contents

Glossary

- Check: A situation in which a player's king is under threat of capture on their opponent's next turn.
- Checkmate: A situation in which a player's king is in check and there is no legal move to remove the threat.
- Stalemate: A kind of draw that happens when one side has NO legal moves to make. If the king is NOT in check, but no piece can be moved without putting the king in check, then the game will end with a stalemate draw.
- Castling: A move to protect the king by having it move "behind" one of its rooks. The space between the king and the rook must be clear, and neither the rook nor the king can have moved in order for castling to be legal.
- Capturing: Any move that takes an opponent's piece.
- Promotion: A situation in which a player's pawn can be upgraded to another piece: except a king: when it has reached the other side of the board.
- Kingside Castling: Castling to the nearer rook. It is denoted by 0-0 in algebraic notation.

Figure 1: Before kingside castling

Figure 2: After kingside castling

- Queenside Castling: Castling to the further rook. It is denoted by 0-0-0 in algebraic notation.

Figure 3: Before queenside castling

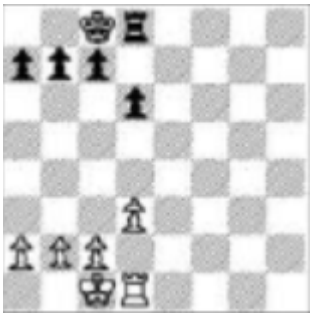

Figure 4: After queenside castling

- En Passant: A move occurs when a pawn advances two squares forward from its starting position and lands beside an opponent's pawn. In this

situation, the opponent has the option to capture the advancing pawn by moving diagonally behind that piece. This capture must be done on the very next move, otherwise the en passant is lost.

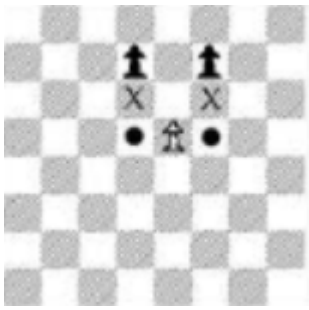

Figure 5: En Passant

- King: A piece that moves from its square to a neighboring square. Denoted as "K" in algebraic notation.

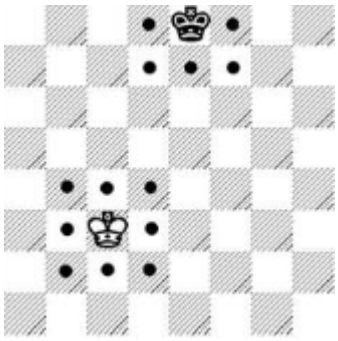

Figure 6: The King

- Queen: A piece that moves horizontally, vertically, and diagonally. Denoted as "Q" in algebraic notation.

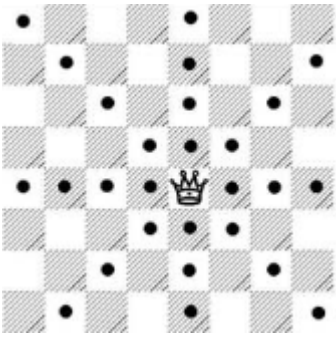

Figure 7: The Queen

- Bishop: A piece that moves diagonally. Denoted as "B" in algebraic notation.

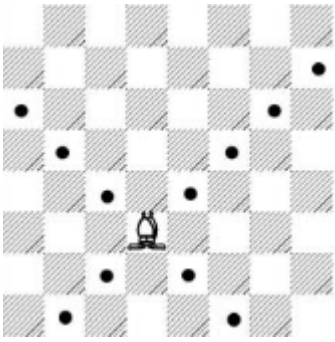

Figure 8: The Bishop

- Knight: A piece that moves in an "L-shape". Denoted as "N" in algebraic notation.

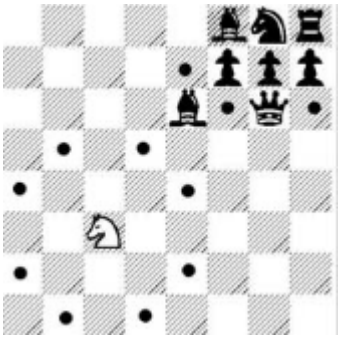

Figure 9: The Knight

- Rook: A piece that moves in its rank or file. Denoted as "R" in algebraic notation.

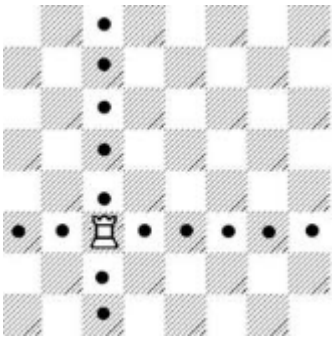

Figure 10: The Rook

- Pawn: A piece that moves one square straight ahead. Can move two squares as its first move. Can En Passant when it is three squares from where it started. Can promote when it has reached the other end of the board. Can be denoted as "P" in algebraic notation.

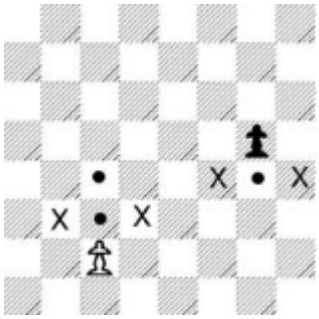

Figure 11: The Pawn

1 Software Architecture Overview

1.1 Main data types and structures

Below is a brief description of the core data types and structures:

- **Piece**: Represents a chess piece in international chess, including the type of piece (such as king, queen, bishop, knight, rook, and pawn) and color (white or black).
- **en_passant[2]**: contains an array of two integers, indicating the position of the last pawn that moved two squares forward, which is used to determine if a pawn can be captured by En Passant.

```
typedef struct {
char name;
char player;
int movement;
int x;
int y;
} Piece;

enum pieces {
rook='R',
pawn='P',
knight='N',
king='K',
queen='Q',
bishop='B',
empty=' '
};

int en_passant[2];
```

1.2 Major software components

This section describes the main software components and how they interact with each other:

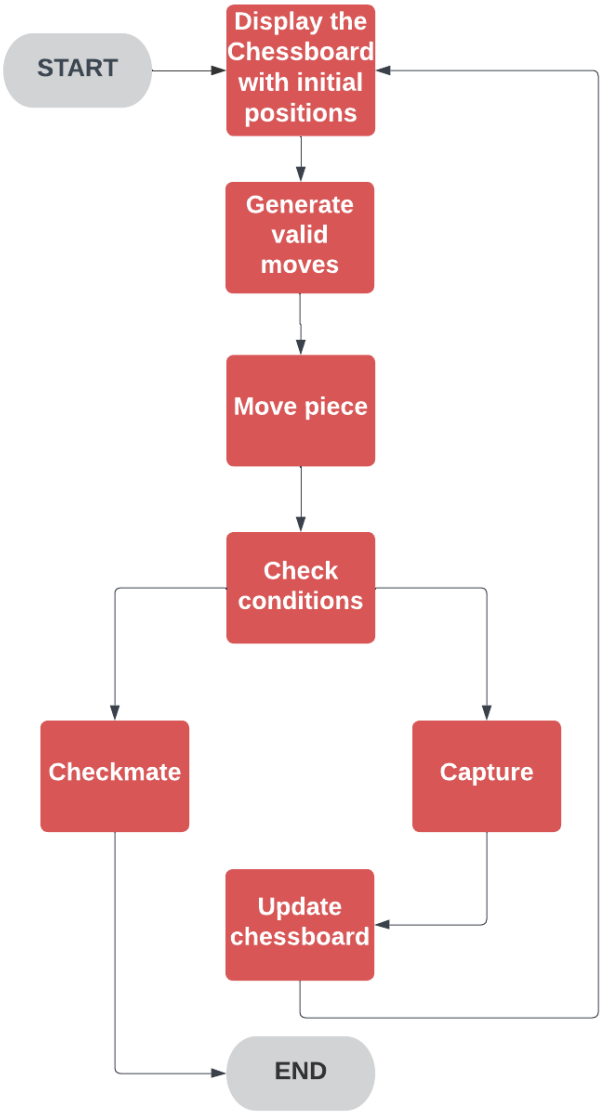

Figure 12: Module hierarchy

1.3 Module interfaces

Below is a brief description of the main functions in the software:

- **addpiece()**: Creates and places a new chess piece on the board.

```
void addpiece(Piece chessboard[BOARD\_SIZE][BOARD\_SIZE],
    char name, bool movement, int x, int y);
```

- **initiatechessboard()**: Responsible for initializing the ChessBoard structure, placing the pieces in their initial positions.

```
void initiatechessboard(
    Piece chessboard[BOARD\_SIZE][BOARD\_SIZE]);
```

- **displaychessboard()**: Responsible for displaying the current board state on the screen.

```
void displaychessboard(
    Piece chessboard[BOARD\_SIZE][BOARD\_SIZE]);
```

- **clean_board()**: Responsible for freeing the memory allocated for the ChessBoard structure.

```
void clean_board(
    Piece chessboard[BOARD\_SIZE][BOARD\_SIZE));
```

- **LoadFEN()**: Loads the board state from a FEN file.

```
int LoadFEN(
    Piece chessboard[BOARD\_SIZE][BOARD\_SIZE]);
```

- **GetFile()**: Gets the filename of the input fen file from the user.

```
FILE *GetFile(void);
```

- **ParseFEN()**: Parses the FEN string and updates the board and game state.

```
void ParseFEN(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE],
    char *fen_string);
```

- **SaveFEN()**: Saves the board and game state to a new FEN file.

```
void SaveFEN(
    const Piece chessboard[BOARD_SIZE][BOARD_SIZE]);
```

- **LoadLog()**: Loads the board state from a log file.

```
void LoadLog(void);
```

- **SaveLog()**: Saves the most recent move to a log file.

```
void SaveLog(void);
```

- **CreateNewLog()**: Creates a new log file.

```
void CreateNewLog(void);
```

- **is_valid_move()**: Checks if a given move is valid.

```
int is_valid_move(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE],
    int from_row, int from_col, int to_row,
    int to_col, char turn);
```

- **move_piece()**: Executes the move on the board based on input. returns 1 if the move is valid, 0 otherwise

```
int move_piece(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE],
```

```
3       char *move, char turn);
```

- **generate_all_moves()**: Generates all possible moves for a specific piece.

```
1 int generate_valid_moves(
2     Piece chessboard[BOARD_SIZE][BOARD_SIZE],
3     int row, int col,
4     int (*valid_moves)[2]);
```

- **generate_valid_moves()**: Generates all valid moves for a piece and considers check.

```
1 int generate_valid_moves(
2     Piece chessboard[BOARD_SIZE][BOARD_SIZE],
3     int row, int col,
4     int (*valid_moves)[2]);
```

- **is_valid_square()**: Checks if a given square is valid. returns 1 if the square is valid, 0 otherwise

```
1 int is_valid_square(int row, int col);
```

- **add_single_move()**: Adds a single move to the valid moves array.

```
1 int add_single_move(
2     Piece chessboard[BOARD_SIZE][BOARD_SIZE],
3     int from_row, int from_col,
4     int to_row, int to_col,
5     int (*valid_moves)[2], int move_count);
```

- **add_line_move()**: Adds a line of moves to the valid moves array.

```
1 int add_line_moves(
2     Piece chessboard[BOARD_SIZE][BOARD_SIZE],
3     int from_row, int from_col,
```

```
    int dr, int dc, int (*valid_moves)[2],
    int move_count);
```

- **is_castling_move()**: Checks if a move satisfies the conditions for castling.

```
int is_castling_move(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE],
    int from_row, int from_col,
    int to_row, int to_col, char player);
```

- **is_check()**: Checks if a given color is in check.

```
int is_check(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE],
    char player);
```

- **is_checkmate()**: Checks if a given color is in checkmate.

```
int is_checkmate(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE],
    char player);
```

- **is_stalemate()**: Checks if a given color is in stalemate.

```
int is_stalemate(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE],
    char player);
```

- **promote_pawn()**: Displays the pawn promotion menu

```
char promote_pawn();
```

- **ai_move()**: Returns the best possible move for the computer's side

```
int ai_move(Piece chessboard[BOARD_SIZE][BOARD_SIZE],
        struct UserList *userlist,
```

```
3         struct AiList *ailist);
```

1.4 Overall program control flow

Below is the overall control flowchart of the program:

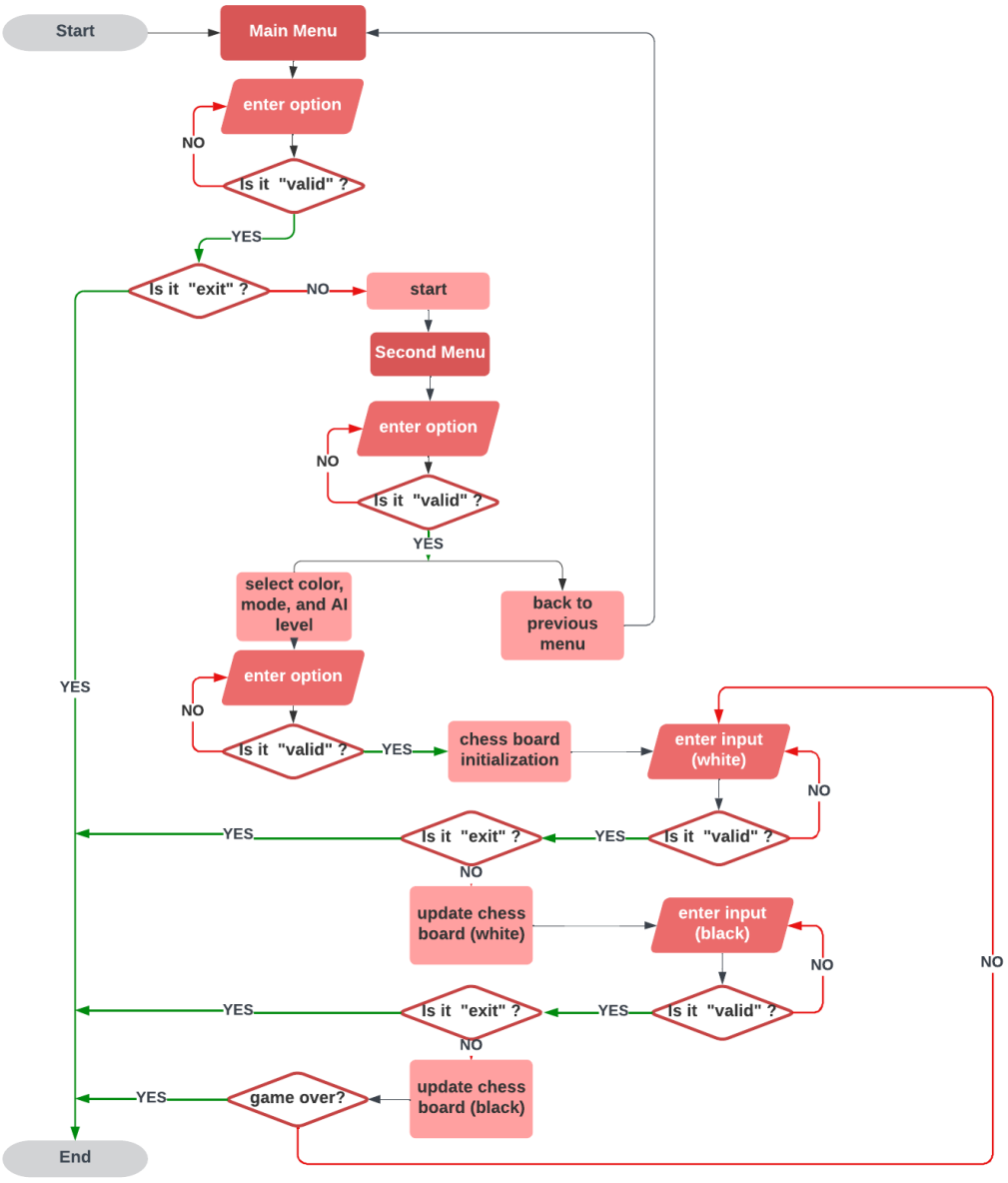

Figure 13: Overall control flowchart

2 Installation

2.1 System requirements

- RAM: 2 GB
- Disk Space: 200 MB
- Operating System: Linux OS (CentOS-7-x86_64)

2.2 Setup and configuration

To download the program, use the scp command to copy the files from the UCI EECS Linux servers to your local machine. Enter the program's directory. To build the program, type "make". To test the program, type "make test". To run the program, type "make run".

2.3 Uninstalling

To uninstall the program, simply run a "make clean" command in the terminal.

3 Documentation of packages, modules, interfaces

3.1 Detailed description of data structures

- **Piece**: The structure includes the type of piece (e.g., king, queen, bishop, knight, rook, and pawn) and color (white or black). This structure contains the following fields:
  - char name: The type of piece ('K', 'Q', 'B', 'N', 'R', 'P').
  - char player: The color of the piece ('W' or 'B').
  - int movement: If the piece has moved or not.
  - int x: The x-coordinate of the piece.
  - int y: The y-coordinate of the piece.

- **ChessBoard**: The structure contains an 8x8 ChessPiece array, representing the possible pieces at each position on the board. This structure contains the following fields:
  - Piece board[BOARD_SIZE][BOARD_SIZE]: Represents the pieces at each position on the board.
  - en_passant[2]: contains an array of two integers, indicating the position of the last pawn that moved two squares forward, which is used to determine if a pawn can be captured by En Passant.

3.2 Detailed description of functions and parameters

Below is a detailed description of the functions and their parameters:

- **addpiece()**: Creates and places a new chess piece on the board.

```
1 void addpiece(Piece chessboard[BOARD_SIZE][BOARD_SIZE],
2     char name, bool movement, int x, int y);
```

  - chessboard: The board to which the piece is added.
  - name: The type of piece ('K', 'Q', 'B', 'N', 'R', 'P').
  - movement: If the piece has moved or not.
  - x: The x-coordinate of the piece.
  - y: The y-coordinate of the piece.

- **initiatechessboard()**: Responsible for initializing the ChessBoard structure, placing the pieces in their initial positions.

```
1 void initiatechessboard(
2     Piece chessboard[BOARD_SIZE][BOARD_SIZE]);
```

  - chessboard: The board to be initialized

- **displaychessboard()**: Responsible for displaying the current board state on the screen.

```
void displaychessboard(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE]);
```

  - chessboard: The board to be displayed

- **clean_board()**: Responsible for freeing the memory allocated for the ChessBoard structure.

```
void clean_board(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE));
```

  - chessboard: The board to be cleaned

- **LoadFEN()**: Loads the board state from a FEN file.

```
int LoadFEN(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE]);
```

  - Opens a .fen file and parses the FEN string inside, updating the board state and game state.
  - Returns an integer indicating the status of the operation. 0 if successful, 1 if chessboard passed in was NULL, and 2 if the FEN file was empty.
  - chessboard: The Piece array that will be loaded to from the FEN file

- **GetFile()**: Gets the filename of the input fen file from the user.

```
FILE *GetFile(void);
```

  - Prompts the user for an input file name (under 64 characters). Opens the file if it exists or asks again if it does not.
  - Returns a pointer to the fen file.

- **ParseFEN()**: Parses the FEN string and updates the board and game state.

```
void ParseFEN(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE],
    char *fen_string);
```

  - Reads through each character of the fen string and makes the appropriate updates.
  - chessboard: The Piece array that will be loaded to from the FEN file
  - fen_string: The fen string to be parsed

- **SaveFEN()**: Saves the board and game state to a new FEN file.

```
void SaveFEN(
    const Piece chessboard[BOARD_SIZE][BOARD_SIZE]);
```

  - Gets the board and game state. Converts these states to a string then saves the string into a fen file.
  - chessboard: The Piece array that will be saved into the FEN file

- **LoadLog()**: Loads the board state from a log file.

```
void LoadLog(void);
```

- **SaveLog()**: Saves the most recent move to a log file.

```
void SaveLog(void);
```

- **CreateNewLog()**: Creates a new log file.

```
void CreateNewLog(void);
```

- **is_valid_move()**: Checks if a given move is valid.

```
int is_valid_move(
    Piece chessboard[BOARD\_SIZE][BOARD\_SIZE],
    int from_row, int from_col, int to_row,
    int to_col, char turn);
```

  - Returns 1 if the move is valid, 0 otherwise
  - chessboard: The reference board
  - from_row: The row of the source
  - from_col: The column of the source
  - to_row: The row of the destination
  - to_col: The column of the destination
  - turn: The color of the player making the move

- **move_piece()**: Executes the move on the board based on input.

```
int move_piece(
    Piece chessboard[BOARD\_SIZE][BOARD\_SIZE],
    char *move, char turn);
```

  - Returns 1 if the move is valid, 0 otherwise
  - chessboard: The board containing all of the pieces
  - move: The move being made
  - turn: The player who is making the move

- **generate_all_moves()**: Generates all possible moves for a specific piece.

```
int generate_valid_moves(
    Piece chessboard[BOARD\_SIZE][BOARD\_SIZE],
    int row, int col,
    int (*valid_moves)[2]);
```

  - Returns the number of moves

- chessboard: The reference board
- row: The row of the piece
- col: The column of the piece
- valid_moves: The array of valid moves

- **generate_valid_moves()**: Generates all valid moves for a piece and considers check.

```
int generate_valid_moves(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE],
    int row, int col,
    int (*valid_moves)[2]);
```

  - Returns the number of valid moves
  - chessboard: The reference board
  - row: The row of the piece
  - col: The column of the piece
  - valid_moves: The array of valid moves

- **is_valid_square()**: Checks if a given square is valid.

```
int is_valid_square(int row, int col);
```

  - Returns 1 if the square is valid, 0 otherwise
  - row: The row of the piece
  - col: The column of the piece

- **add_single_move()**: Adds a single move to the valid moves array.

```
int add_single_move(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE],
    int from_row, int from_col,
    int to_row, int to_col,
    int (*valid_moves)[2], int move_count);
```

- Returns the number of valid moves
- chessboard: The reference board
- from_row: The row of the source
- from_col: The column of the source
- to_row: The row of the destination
- to_col: The column of the destination
- valid_moves: The array of valid moves
- move_count: The number of moves

- **add_line_move()**: Adds a line of moves to the valid moves array.

```
int add_line_moves(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE],
    int from_row, int from_col,
    int dr, int dc, int (*valid_moves)[2],
    int move_count);
```

  - Returns the number of valid moves
  - chessboard: The reference board
  - from_row: The row of the source
  - from_col: The column of the source
  - to_row: The row of the destination
  - to_col: The column of the destination
  - valid_moves: The array of valid moves
  - move_count: The number of moves

- **is_castling_move()**: Checks if a move satisfies the conditions for castling.

```
int is_castling_move(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE],
    int from_row, int from_col,
    int to_row, int to_col, char player);
```

- Returns 1 if the move is a castling move, 0 otherwise
- chessboard: The reference board
- from_row: The row of the source
- from_col: The column of the source
- to_row: The row of the destination
- to_col: The column of the destination
- player: The color of the player making the move

- **is_check()**: Checks if a given color is in check.

```
int is_check(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE],
    char player);
```

  - Returns 1 if the color is in check, 0 otherwise
  - chessboard: The reference board
  - player: The color of the player making the move

- **is_checkmate()**: Checks if a given color is in checkmate.

```
int is_checkmate(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE],
    char player);
```

  - Returns 1 if the color is in checkmate, 0 otherwise
  - chessboard: The reference board
  - player: The color of the player making the move

- **is_stalemate()**: Checks if a given color is in stalemate.

```
int is_stalemate(
    Piece chessboard[BOARD_SIZE][BOARD_SIZE],
    char player);
```

- Returns 1 if the color is in stalemate, 0 otherwise
- chessboard: The reference board
- player: The color of the player making the move

- **promote_pawn()**: Displays the pawn promotion menu

```
char promote_pawn();
```

  - Returns the piece that the pawn is promoted to

- **ai_move()**: Returns the best possible move for the computer's side

```
int ai_move(Piece chessboard[BOARD\_SIZE][BOARD\_SIZE],
        struct UserList *userlist,
        struct AiList *ailist);
```

  - Returns 1 if the move is valid, 0 otherwise
  - chessboard: The reference board
  - userlist: The list of users
  - ailist: The list of AI players

3.3 Detailed description of input and output formats

- Syntax/format of a move input by the user: Users need to enter the position of the piece they want to move and a position they want to move the piece to, for example, A2A4 or B7B8Q. The first two characters represent the row and column of the piece to be moved, and the last two characters represent the row and column of the destination.

- Syntax/format of a move recorded in the log file: The log file will record the piece that was moved, the position to which it was moved, and the number of the full move when it occurred. For example, on the first turn, if white moves pawn at e2 to e4 and black moves pawn at d7 to d5, this would be recorded in the log file as "1. E4 D5". The Forsyth-Edwards Notation is a common way of storing the state of the board and the game in a single string or text file. A fen file

will record the locations of pieces on the board; which player's turn it is; which castlings are available, if any; the en passant square if it exists; the number of half moves made; and the number of full moves made. As an example, the FEN string for the initial board is "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq: 0 1". The lowercase letters represent black pieces, the upper case letters represent white pieces, and the numbers represent empty spaces.

4 Development plan and timeline

4.1 Partitioning of tasks

- Week 1: User Manual: We will be writing up a document with all the rules of the game and providing detailed instructions on how to operate the program. Will also contain descriptions of functions being used during the program execution.

- Week 2: Software Specifications: We will create a document outlining the format of the program in a code-intensive manner, delving deep into the way the program is written. We will outline the created data structures, the user-defined functions and the program flow.

- Week 3: Alpha Release: We will release the first iteration of the Chess Game, with a barebones AI and a functional chess game. It will meet several of the requirements and allow for a playable experience. We will then set up testing and begin to streamline the game to make it tournament-ready.

- Week 4: Final Release: Our team will present our final version of the Chess Game, with a streamlined, intelligent AI and a sleeker look of the board and the game overall. Our priority will be to increase the intelligence of the AI and make it capable of playing organized chess at an intermediate level.

- Week 5: Tournament Prep: This week will be spent preparing the AI for the tournament to be played against fellow teams. It will involve a deeper look into the game of chess and the strategies involved. We expect to have a fully functional chess game at this point, and our priority will be to increase the features of the program.

4.2 Team member responsibilities

- Yongye Li: Main program, User interface, Chess objects, Chess rules.
- Zijie Huang: Main program, User interface, Chess objects, Chess rules, AI Module.
- Cristian Pina Bravo: Chess Rules.
- Arjun Sivakumar: AI Module, Chess Rules, Testing
- Mary Campbell: Move Trees, AI Module
- Joseph Principe: Log File, Documentation

Back matter

Copyright

References

- Hartikka, L. (2017, March 15). A step-by-step guide to building a simple chess AI. freeCodeCamp.org. Retrieved April 10, 2022, from https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/
- US Chess Rule Book Online. (n.d.). Retrieved April 24, 2023, from https://new.uschess.org/sites/default/files/media/documents/us-chess-rule-book-online-only-edition-chapters-1-2-10-11-9-1-20.pdf

Error messages

- Invalid move error: This error occurs when the user tries to make an illegal move. A message will be displayed, prompting the user to make a legal move.
- Invalid Command Error: This error occurs when the user tries to input some illegal command. A message will be displayed, prompting the user to make a legal command.
- Out of memory error: This error occurs when the system does not have enough memory to run the program. It is recommended to close other applications and try again.

Index

- T
  - Timer
- U
  - Usage scenario