YOUR MOVE v1.0

Software Specification

EECS 22L
Spring 2023

King’s Sacrifice

Nishant Mehendre
Meghana Burugupalli
Priyanka Samavedam
Alex Dunn
Xinrui Xie

Table of Contents

Glossary

- **Pieces:**

1. Pawn:
    - Can only move forward
    - Can move two or one space initially
    - Can move only one space after
    - Can attack one space diagonally
2. Knight:
    - Can only move in an “L-shape” (either move two spaces vertically followed by one space horizontally or move two spaces horizontally followed by one space vertically)
    - Can move a total of three spaces
    - Can attack another piece according to the L-shape
3. Bishop:
    - Can move diagonally backward or diagonally forward
    - Can move any number of spaces in a single move
    - Can attack in any direction and move any number of spaces while attacking
4. Rook:
    - Can move vertically backward or vertically forward and horizontally (left and right)
    - Can move any number of spaces in a single move
    - Can attack in any direction and move any number of spaces while attacking

5. Queen:
    - Can move in any direction (forward, backward, diagonal, horizontal)
    - Can move any number of spaces
    - Can attack in any direction and move any number of spaces while attacking
6. King:
    - Can move in any direction
    - Can only move one space
    - If it can be attacked in the next move, player must move this piece to safety (CHECK)
    - If it can be attacked in the next move and the player has no spaces to ensure safety, player loses the game (CHECKMATE)

- **Endgame:**
    1. Checkmate:

        If a king is currently being attacked and the king has no safe spaces to move to, the player to whom the king belongs loses.
    2. Stalemate:

        If moving a king is the only legal move a player can make and the king is currently in a safe position, but all spaces where the king can move are being attacked, it is a stalemate and is considered a draw.

    3. Fifty Move Rule:

        The game is declared a draw if no piece has been captured or no pawn has been moved for 50 consecutive turns of a single player.

    4. Insufficient Material Draw:

        If both sides have any of the following scenarios:
        - only king surviving
        - only a king and a bishop surviving
        - only a king and a knight surviving
    5. Out of time:

        If a timer exists and a player is out of time, it is considered a defeat for that player and the other player wins.
- **Special Moves:**
    1. Castling:
        - The king may move two squares to either side, with the rook moving to the other side of the king. Not a legal move if either piece has moved or if the king is currently checked.
    2. En Passant:
        - a special pawn capture which can occur immediately after a player moves a pawn two squares forward from its starting position, and an enemy pawn could have captured it had the same pawn moved only one square forward. The opponent captures the just-moved pawn as if taking it as it passes through the first square. The resulting position is the same as if the pawn had moved only one square forward and the enemy pawn was captured normally. The En Passant capture must be done on the very next turn, or the right to do so is lost. Such a move is the only occasion in chess in which a piece captures but does not move to the square of the captured piece. If an En Passant capture is the only legal move available, it must be made.

- **Other:**
    1. Check:
        - When one of a player's pieces is attacking the opponent's king or vice versa. Unlike checkmate, here the king can be protected by moving it in another location or blocking the path of attack with another piece.
    2. Capture:
        - When one of a player's pieces takes the position that was previously held by an opponent's piece or vice versa through a legal move. The piece which initially held the position is removed from the game (no longer on the board), i.e the player cannot use it.
    3. Pawn Promotion:
        - A player can switch a pawn for a queen, a rook, a bishop, or a knight in the eighth rank for a White pawn and in the first rank for a Black pawn

Software Architecture Overview

1.1 Main Data Types and Structures

- **Pieces (Structure):**
    - **Position (Structure):** The location on the board where the piece is
    - **Player (enum):**
        Player 1 → White [This player makes the first move]
        Player 2 → Black
    - **Type (enum):** The type of piece decides how the piece moves (Its "legal moves")
        Pawn
        Knight
        Bishop
        Rook
        Queen
        King

      The movement of each piece is discussed in the glossary

- **Position (Structure):**
    Piece (pointer structure Pieces): The piece that is located at the particular position
    Rank (int): The game board row
    File (char): The game board column
    x1, y1, x2, y2 (float) → Coordinates

- **Game (Structure):**
    List (pointer structure GameList): Point to the list of games
    Board[8][8] (structure Position): A 2-D Array of positions (the game board)
    Prev (pointer structure Game): Points to the previous game (the board with one move prior)
    Next (pointer structure Game): Points to the next game (the board with the next move)

- **GameList (Structure):**
    Curr (pointer structure Game): Points to the latest game (one with the latest moved piece)

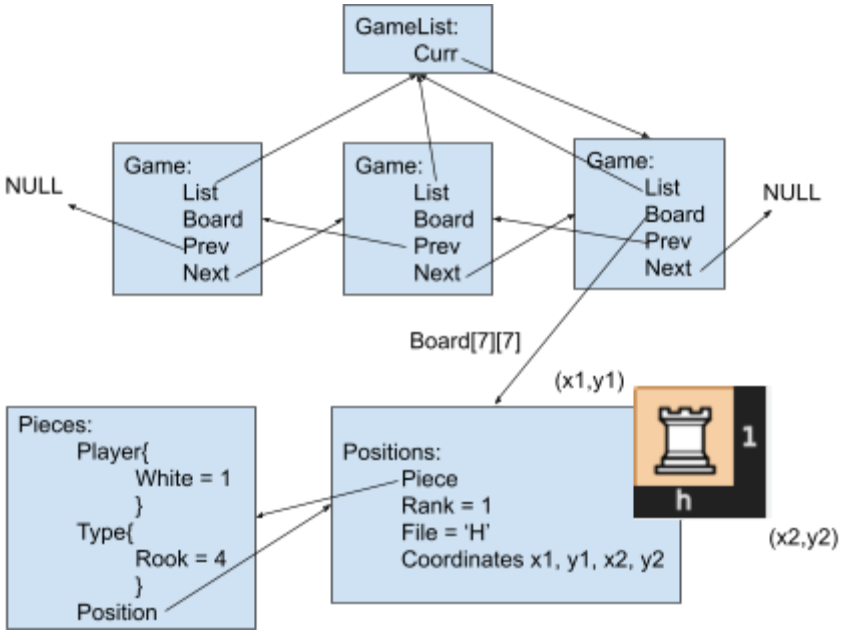

Figure 1 - Pointer structure of software program

1.2 Major Software components

Object Module: Contains all the data structures and memory allocation (Pieces, Positions, Game, Gamelist)

Move Module: Contains functions that control the player interactions and the defines the movement of the pieces

Main Module: Includes the user interface and Computer AI.

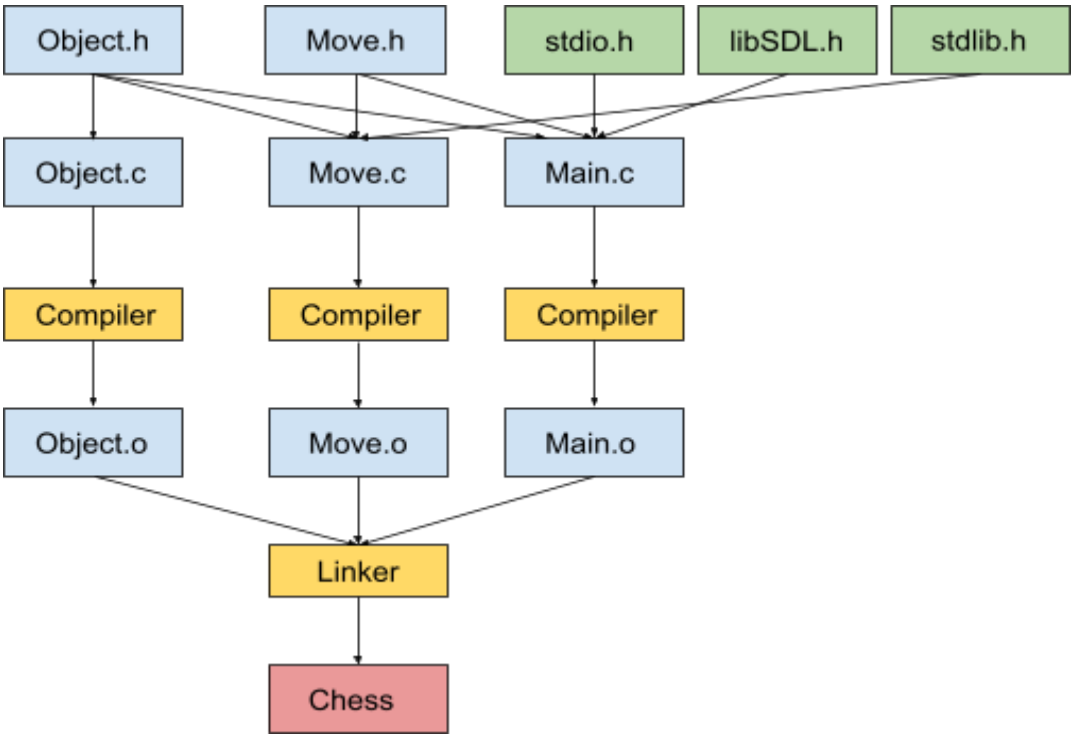

Figure 2 - Module hierarchy of software program diagram

1.3 Module Interfaces

The modules Object and Move determine the movement and functionality of the game
The Libraries are essential to receive user input and display output (stdio), assertions in the code to prevent errors and help with debugging (stdlib) and the GUI (libSDL)
The main module is the game program that defines the program control flow

Chess Main Module (Game Control Flow and Computer (AI)		
Object (Pieces, Positions, Game)	Move (Movement of the game)	GUI
Libraries (stdio, stdlib, libSDL)		
Linux OS		

Table 1 - API of module functions

1.4 Program control flow

The user first views the "Start Screen":

The user can then customize the game using the options this screen provides:
These options are in order of how they appear in the program

→ **Mode:** This decides the players that will be playing the game

1. Player vs Player:
   Both players are users
2. Player vs Computer:
   The user plays against the computer
3. Computer vs Computer:
   The user gets the computer to play against itself

→ **Difficulty:** The player can choose the difficulty of the computer
This option is only applicable if at least one of the players is the computer.

1. Rookie: Easiest difficulty, the program makes random legal moves.
2. Master: Medium difficulty, the program makes random legal moves but prioritizes protecting the pieces currently under attack.
3. Grandmaster: Hardest difficulty, the program protecting and attacking pieces based on the value of the piece.

→ **Player:** The player chooses who makes the first move

1. White: This player makes the first move
2. Black: The second player

→ **Timer:** The player can choose to have a time constraint to make moves

1. On: The player has 1 minute to make a move

2. Off: The player has no time constraint

→ Game Starts:

1. Players (Users or Computer) take turns playing a move depending on the turn which is based on which player moves White and which player moves Black pieces.

→ End game:

→ Checkmate: When one of the players makes it so that the king of the opponent is being currently attacked as well as has no new location to move to as all legal positions are also under attack.

In this case, the player who gave the checkmate wins and the other player loses.

→ Draw:

1. Stalemate: When one of the players has only the king piece remaining and it is in a safe position. However, the king cannot move to another location as all the legal positions are under attack by the opponent. This game ends in a draw.
2. Insufficient Materials: When both of the player have remaining pieces of the following combination:
    a. King and Bishop
    b. King and Knight
    c. King

→ Post Game:

1. The player can restart the game with current customization
2. The player can go back to the home page
3. The player can quit the application using the close application button on the top right corner

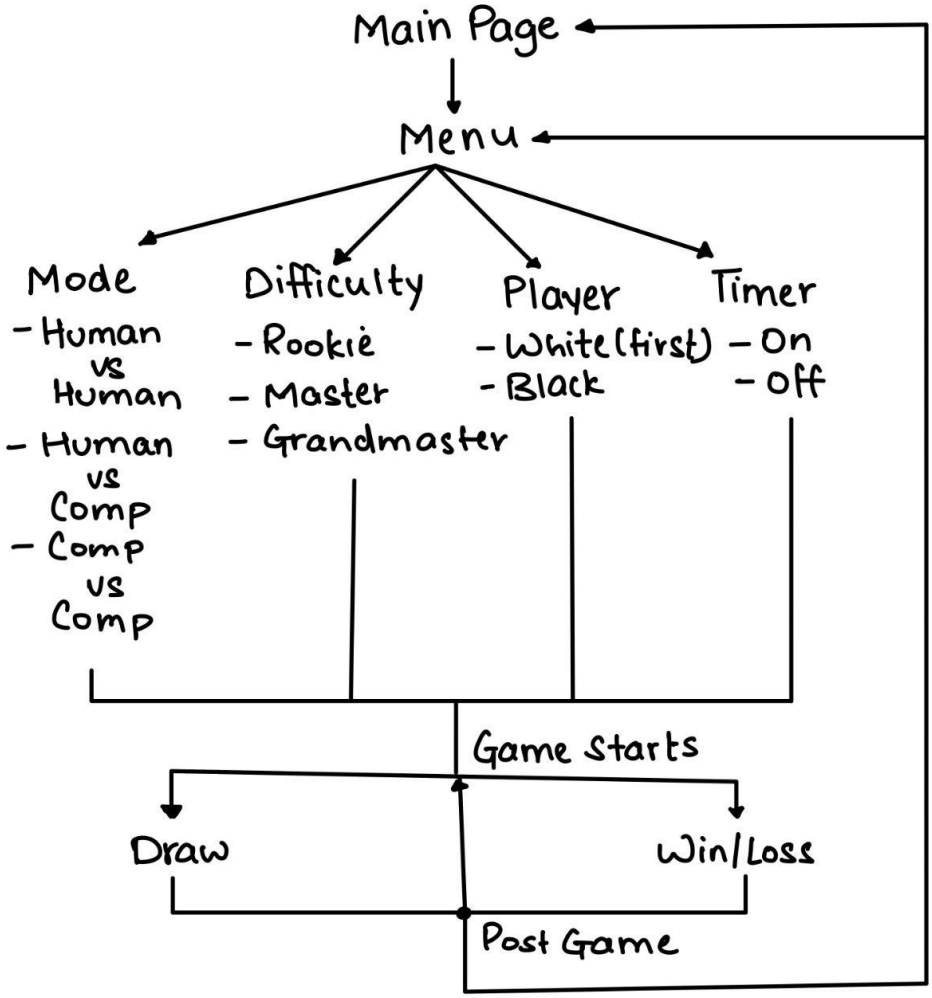

Figure 3 - Diagram of program control flow

Installation

2.1 System requirements, compatibility

	Minimum Requirements
CPU	x86_64
Operating System	Linux
Internet	Broadband Internet connection
Resolution	1024x768 Display Resolution

Table 1 - System requirements

2.2 Setup and configuration

If using Windows, open a terminal window via ssh (Putty is commonly used). Connect to one of the following UCI linux servers:

"Your user name"@bondi.eecs.uci.edu
"Your user name"@crystalcove.eecs.uci.edu
"Your user name"@laguna.eecs.uci.edu

If using Mac, open the terminal program and create a new remote connection to connect to one of the previously listed UCI linux servers.

2.3 Building, compilation , installation

Commands needed to build the source code package:

% gtar xvzf SourceArchive.tar.gz

% evince chess/doc/chess software.pdf

% cd chess

% make

% make test

% make clean

Documentation of packages, modules, interfaces

3.1 Detailed description of data structures

- **Piece (Structure):** The pieces on the board that the player moves to play the game.
    - **Position:** Pointer structure
        - The position on the board where the piece exists. The position is determined by its rank and its file.
    - **Player - enum**
        - Each piece has two different types, white and black. The colors indicate which team the user is on and who will make the first move (since white always moves first).
    - **Type - enum**
        - Each piece has six different types. Each type of piece moves differently from each other. To see how each piece can legally move, check the glossary. The six different pieces are:
            - Pawn = 1
            - Knight = 2
            - Bishop = 3
            - Rook = 4
            - Queen = 5
            - King = 6

```
/* Game Piece */
typedef struct Pieces{
    t_Position *Position;

    enum Player{
        White,
        Black;
    }
    enum Type{
        Pawn,
        Knight,
        Bishop,
        Rook,
        Queen,
        King;
    }
}t_Piece;
```

Fig 4: Pieces Data Structure

- **Position (Structure):**
  - **Piece: Pointer structure**
    - This points to the piece that exists on the location. If there is no piece that exists at the location it is set to NULL.
  - **Rank: Integer**
    - The row
  - **File: Character**
    - The column

```
/* Position */
typedef struct Positions{
    t_Piece *Piece;
    int Rank;
    char File;
    float x1,y1,x2,y2;
}t_Position;
```

Fig 5: Position Data Structure

- **GameList (Structure):**
  - Curr: Pointer to Game
    - Points to the game with the latest move

- **Game (Structure):**
  - **List:** Pointer to GameList
    - Points to the list of the games (boards with all moves)
    - Helps to implement undo, change the Curr of the list and delete the previous current.
  - **Board:** 2D Array of Positions
    - This is the game board where the players make moves.
  - **Prev:** Pointer to Structure Game
    - It points to the game with one less move (If undo is implemented Curr->Prev becomes the current)
  - **Next:** Pointer to Structure Game
    - It points to the game with next move (Curr is last element in the list)

```c
/* Game List */
typedef struct GameList{
    t_Game *Curr;
}t_GameList;

/* Game Board */
typedef struct Game{
    t_GameList *List;
    t_Position Board[8][8];
    t_Game *Prev;
    t_Game *Next;
}t_Game;
```

Fig 6: Game and GameList Data Structure

3.2 Detailed description of functions and parameters

Move.c, the module which contains majority of the functions includes the following ones:

- Is_Legal: Is_Legal is an integer function that takes in two parameters (position1 and position2) which are pointers to the struct t_Positions and returns a value of either 0,1, or 2. The Is_Legal function is used to identify if a move made by the player is legal or not according to the rules of chess. When the player makes a move that is illegal, the function returns a value of 0; when the move is legal without a piece of the opponent being captured, the function returns a value of 1; when the move is legal and there is a capture, the function returns a value of 2. The function asserts the existence of two different positions (an initial and a final) and makes use of various if statements to check for different types of pieces.

```c
int Is_Legal(t_Position *position1, t_Position *position2){

    /* Check if position1 and position2 exist */
    assert(position1);
    assert(position2);
    assert(position1 != position2);
```

Figure 7 - Is_Legal function with parameters and assert statements

If the piece is of type Pawn, the function first checks if the piece only moved one step forward and that there is no capture by checking if the rank of the initial position subtracted from the rank of the final position is equal to 1 and the piece in the final position is NULL. If the initial condition is not satisfied, it checks for capture by checking if the rank of the initial position subtracted from the rank of the final position is 1, the absolute value of the file of the initial position subtracted from the file of the final position is 1, and that the piece in the final position is of the opposite player. If neither of the conditions are satisfied, the move is illegal and the function returns 0.

```
/* Check for Pawn */
if (position1 -> Piece -> Type == Pawn){
        if (position1 -> Piece -> Player == White){
                if(position2 -> Rank - position1 -> Rank == 1 && position2 -> Piece == NULL){
                        return 1; /* no capture, legal move */
                }
                else if(position2 -> Rank - position1 -> Rank == 1 && abs(position2 -> File - position1 -> File) == 1 && position2 -> Piece != NULL && position2->Piece -> Player == Black){
                        return 2; /* capture, legal move */
                }
                else{
                        return 0; /* illegal move */
                }
        }
        else if (position1 -> Piece -> Player == Black){
                if(position1 -> Rank - position2 -> Rank == 1 && position2 -> Piece == NULL){
                        return 1; /* no capture, legal move */
                }
                else if(position1 -> Rank - position2 -> Rank == 1 && abs(position2 ->  File - position1 -> File) == 1 && position2 -> Piece != NULL && position2 -> Piece -> Player == White){
                        return 2; /* capture, legal move */
                }
                else{
                        return 0; /* illegal move */
                }
        }

}
```

Figure 8 - Is_Legal function checking for Pawn

If the piece is of type Bishop, the function first checks if the piece in the final position is of the opposite player and if the piece is moving diagonally (checks if the absolute value of the rank of the initial position subtracted from the rank of the final position is equal to the absolute value of the file of the initial position subtracted from the file of the final position). If either of these conditions are not satisfied, the move is illegal and the function returns 0. Otherwise, the function uses a two variable for loop, with increasing rank and file to check if there is any piece between the initial position and the final position (excluded) obstructing the Bishop's path. If there is, the function returns 0 as the Bishop can't jump over pieces like the Knight. If there is no piece in its path, the function returns 1 for legal move, no capture, and if there is a piece of the opponent player on the final position, the function returns 2 for legal move with capture.

```
/* Check for Bishop */
if (position1 -> Piece -> Type == Bishop){
        if (abs(position1 -> Rank - position2 -> Rank) != abs(position1 -> File - position2 -> File) || (position2 -> Piece -> Player == position2 -> Piece -> Player)){
              return 0; /*illegal move */
        }
        else{
            int checker_b = 0;
            /* white moves up and right, black moves down and left */
            if (position1 -> Rank < position2 -> Rank && Position1 -> File < Position2 -> File){
                    for (int i = position1 -> Rank, int j = position1 -> File; (i < position2 -> Rank) && (j < position2 -> File); i++, j++){
                        if (Board[i][j] -> Piece != NULL){
                            checker_b +=1;
                            return 0; /* illegal move, piece in path*/
                        }
                    }
                    if (checker_b == 0){
                        if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                            return 2; /* capture, legal move*/
                        }
                        else if (position2 -> Piece == Null){
                            return 1; /* no capture, legal move*/
                        }
                    }

            }
            /* white moves up and left, black moves down and right */
            else if (position1 -> Rank < position2 -> Rank && Position1 -> File > Position2 -> File){
                    for (int i = position1 -> Rank, int j = position1 -> File; (i < position2 -> Rank) && (j > position2 -> File); i++, j--){
                        if (Board[i][j] -> Piece != NULL){
                            checker_b +=1;
                            return 0; /* illegal move, piece in path*/
                        }
                    }
                    if (checker_b == 0){
                        if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                            return 2; /* capture, legal move*/
                        }
                        else if (position2 -> Piece == Null){
                            return 1; /* no capture, legal move*/
                        }
                    }

            }
```

Figure 9 - Is_Legal function checking for Bishop part1

```
            /* white moves down and left, black moves up and right */
            else if (position1 -> Rank > position2 -> Rank && Position1 -> File > Position2 -> File){
                    for (int i = position1 -> Rank, int j = position1 -> File; (i > position2 -> Rank) && (j > position2 -> File); i--, j--){
                        if (Board[i][j] -> Piece != NULL){
                            checker_b +=1;
                            return 0; /* illegal move, piece in path*/
                        }
                    }
                    if (checker_b == 0){
                        if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                            return 2; /* capture, legal move*/
                        }
                        else if (position2 -> Piece == Null){
                            return 1; /* no capture, legal move*/
                        }
                    }

            }

            /* white moves down and right, black moves up and left */
            else if (position1 -> Rank > position2 -> Rank && Position1 -> File < Position2 -> File){
                    for (int i = position1 -> Rank, int j = position1 -> File; (i > position2 -> Rank) && (j < position2 -> File); i--, j++){
                        if (Board[i][j] -> Piece != NULL){
                            checker_b +=1;
                            return 0; /* illegal move, piece in path*/
                        }
                    }
                    if (checker_b == 0){
                        if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                            return 2; /* capture, legal move*/
                        }
                        else if (position2 -> Piece == Null){
                            return 1; /* no capture, legal move*/
                        }
                    }
            }

            else return 0; /* illegal move */
        }
    }
```

Figure 10 - Is_Legal function checking for Bishop part2

If the piece is of type Rook, the function first checks if the piece in the final position is of the opposite player. If not, the move is illegal and the function returns 0. Otherwise, the function uses a for loop with increasing rank (White Rook moves up and Black Rook moves down), or decreasing rank (White Rook moves down and Black Rook moves up), or increasing file (White Rook moves right and Black Rook moves left), or decreasing

file (White Rook moves left and Black Rook moves right) to check if there is any piece between the initial position and the final position (excluded) obstructing the Rook's path. If there is, the function returns 0 as the Rook can't jump over pieces like the Knight. If there is no piece in its path, the function returns 1 for legal move, no capture, and if there is a piece of the opponent player on the final position, the function returns 2 for legal move with capture.

```
    /* Check for Rook */
    if (position1 -> Piece -> Type == Rook) {
        int checker_r = 0;

        if (position1 -> Piece -> Player == position2 -> Piece -> Player){
            return 0; /* illegal move */
        }

        else {

            /* white moves up, black moves down */
            if (position1 -> Rank < position2 -> Rank && position1 -> File == position2 -> File){
                for (int r = position1 -> Rank; r < position2 -> Rank; r++){
                    if (Board[r][position1 -> File] -> Piece != NULL){
                        checker_r += 1;
                        return 0;  /* illegal move, piece in path */
                    }
                }
                if (checker_r == 0){
                    if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                        return 2; /* capture, legal move*/
                    }
                    else if (position2 -> Piece == NULL){
                        return 1; /* no capture, legal move */
                    }
                }
            }
            /* white moves down, black moves up */
            else if (position1 -> Rank > position2 -> Rank && position1 -> File == position2 -> File){
                for (int r = position1 -> Rank; r > position2 -> Rank; r--){
                    if (Board[r][position1 -> File] -> Piece != NULL){
                        checker_r += 1;
                        return 0; /* illegal move, piece in path */
                    }
                }
                if (checker_r == 0){
                    if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                        return 2; /* capture, legal move */
                    }
                    else if (position2 -> Piece == NULL){
                        return 1; /* no capture, legal move */
                    }
                }
            }
            /* white moves right, black moves left */
            else if (position1 -> Rank == position2 -> Rank && position1 -> File < position2 -> File){
                for (int r = position1 -> File; r < position2 -> File; r++){
                    if (Board[position1 -> Rank][r] -> Piece != NULL){
                        checker_r += 1;
                        return 0; /* illegal move, piece in path */
                    }
                }
                if (checker_r == 0){
                    if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                        return 2; /* capture, legal move */
                    }
                    else if (position2 -> Piece == NULL){
                        return 1; /* no capture, legal move */
                    }
                }
            }
```

Figure 11 - Is_Legal function checking for Rook part1

```
            /* white moves left, black moves right */
            else if (position1 -> Rank == position2 -> Rank && position1 -> File > position2 -> File){
                for (int r = position1 -> File; r > position2 -> File; r--){
                    if (Board[position1 -> Rank][r] -> Piece != NULL){
                        checker_r += 1;
                        return 0; /* illegal move, piece in path */
                    }
                }
                if (checker_r == 0){
                    if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                        return 2; /* capture, legal move */
                    }
                    else if (position2 -> Piece == NULL){
                        return 1; /* no capture, legal move */
                    }
                }
            }
            else return 0; /* illegal move */

        }
    }
```

Figure 12 - Is_Legal function checking for Rook part2

If the piece is of type Knight, the function first checks if the piece in the final position is of the opposite player. If not, the move is illegal and the function returns 0. Otherwise, the function checks if the Knight is moving in an L-shape by checking if the absolute value of the rank of the initial position subtracted from the rank of the final position is 2 and the file of the initial position subtracted from the file of the final position is 1,or vice versa. If this condition is true, it returns 1 for no capture, or 2 for capture. Else, it returns 0 as the move is illegal.

```
    /* check for knight */
    if (position1 -> Piece -> Type == Knight){
        if (position1 -> Piece -> Player == position2 -> Piece -> Player){
            return 0; /* illegal move, same player capture */
        }
        else if ((abs(position1 -> Rank - position2 -> Rank) == 2 && abs(position1 -> File - position2 -> File) == 1) || (abs(position1 -> Rank - position2 -> Rank) == 1 && abs(position1 -> File - positio
n2 -> File) == 2)){
            if (position2 -> Piece == NULL){
                return 1; /* no capture, legal move*/
            }
            else if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                return 2; /* capture, legal move */
            }
        }
        else return 0; /* illegal move */
    }
```

Figure 13 - Is_Legal function checking for Knight

If the piece is of type Queen, the function first checks if the piece in the final position is of the opposite player. If not, the move is illegal and the function returns 0. Otherwise, it checks for diagonal movement in the same manner it did for the Bishop, and for vertical and horizontal movement in the same manner it did for the Rook. The Queen function is a combination of the part of the function for Bishop and Rook. If none of the conditions are satisfied, the function returns 0 (illegal move).

```
/* Check for Queen */
if (position1 -> Piece -> Type == Queen){
    int checker_q = 0;
    if (position1 -> Piece -> Player == position2 -> Piece -> Player){
        return 0; /* illegal move, same player capture */
    }

    /* white moves diagonally up and right, black moves diagonally down and left */
    else if (position1 -> Rank < position2 -> Rank && Position1 -> File < Position2 -> File){
        for (int i = position1 -> Rank, int j = position1 -> File; (i < position2 -> Rank) && (j < position2 -> File); i++, j++){
            if (Board[i][j] -> Piece != NULL){
                checker_q +=1;
                return 0; /* illegal move, piece in path*/
            }
        }
        if (checker_q == 0){
            if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                return 2; /* capture, legal move*/
            }
            else if (position2 -> Piece == Null){
                return 1; /* no capture, legal move*/
            }
        }
    }

    /* white moves diagonally up and left, black moves diagonally down and right */
    else if (position1 -> Rank < position2 -> Rank && Position1 -> File > Position2 -> File){
        for (int i = position1 -> Rank, int j = position1 -> File; (i < position2 -> Rank) && (j > position2 -> File); i++, j--){
            if (Board[i][j] -> Piece != NULL){
                checker_q +=1;
                return 0; /* illegal move, piece in path*/
            }
        }
        if (checker_q == 0){
            if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                return 2; /* capture, legal move*/
            }
            else if (position2 -> Piece == Null){
                return 1; /* no capture, legal move*/
            }
        }

    }

    /* white moves diagonally down and left, black moves diagonally up and right */
    else if (position1 -> Rank > position2 -> Rank && Position1 -> File > Position2 -> File){
            for (int i = position1 -> Rank, int j = position1 -> File; (i > position2 -> Rank) && (j > position2 -> File); i--, j--){
                if (Board[i][j] -> Piece != NULL){
                    checker_q +=1;
                    return 0; /* illegal move, piece in path*/
                }
            }
            if (checker_q == 0){
                if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                    return 2; /* capture, legal move*/
                }
                else if (position2 -> Piece == Null){
                    return 1; /* no capture, legal move*/
                }
            }
```

Figure 14 - Is_Legal function checking for Queen part1

```
        }

        /* white moves diagonally down and right, black moves diagonally up and left */
        else if (position1 -> Rank > position2 -> Rank && Position1 -> File < Position2 -> File){
                for (int i = position1 -> Rank, int j = position1 -> File; (i > position2 -> Rank) && (j < position2 -> File); i--, j++){
                    if (Board[i][j] -> Piece != NULL){
                        checker_q +=1;
                        return 0; /* illegal move, piece in path*/
                    }
                }
                if (checker_q == 0){
                    if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                        return 2; /* capture, legal move*/
                    }
                    else if (position2 -> Piece == Null){
                        return 1; /* no capture, legal move*/
                    }
                }
        }

        /* white moves up, black moves down */
        else if (position1 -> Rank < position2 -> Rank && position1 -> File == position2 -> File){
            for (int r = position1 -> Rank; r < position2 -> Rank; r++){
                if (Board[r][position1 -> File] -> Piece != NULL){
                    checker_q += 1;
                    return 0;  /* illegal move, piece in path */
                }
            }
            if (checker_q == 0){
                if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                    return 2; /* capture, legal move*/
                }
                else if (position2 -> Piece == NULL){
                    return 1; /* no capture, legal move */
                }
            }
        }
        /* white moves down, black moves up */
        else if (position1 -> Rank > position2 -> Rank && position1 -> File == position2 -> File){
            for (int r = position1 -> Rank; r > position2 -> Rank; r--){
                if (Board[r][position1 -> File] -> Piece != NULL){
                    checker_q += 1;
                    return 0; /* illegal move, piece in path */
                }
            }
            if (checker_q == 0){
                if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                    return 2; /* capture, legal move */
                }
                else if (position2 -> Piece == NULL){
                    return 1; /* no capture, legal move */
                }
            }
        }
```

Figure 15 - Is_Legal function checking for Queen part2

```
        /* white moves right, black moves left */
        else if (position1 -> Rank == position2 -> Rank && position1 -> File < position2 -> File){
            for (int r = position1 -> File; r < position2 -> File; r++){
                if (Board[position1 -> Rank][r] -> Piece != NULL){
                    checker_q += 1;
                    return 0; /* illegal move, piece in path */
                }
            }
            if (checker_q == 0){
                if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                    return 2; /* capture, legal move */
                }
                else if (position2 -> Piece == NULL){
                    return 1; /* no capture, legal move */
                }
            }
        }
        /* white moves left, black moves right */
        else if (position1 -> Rank == position2 -> Rank && position1 -> File > position2 -> File){
            for (int r = position1 -> File; r > position2 -> File; r--){
                if (Board[position1 -> Rank][r] -> Piece != NULL){
                    checker_q += 1;
                    return 0; /* illegal move, piece in path */
                }
            }
            if (checker_q == 0){
                if (position2 -> Piece -> Player != position1 -> Piece -> Player){
                    return 2; /* capture, legal move */
                }
                else if (position2 -> Piece == NULL){
                    return 1; /* no capture, legal move */
                }
            }
        }

        else return 0; /* illegal move */
    }
}
```

Figure 16 - Is_Legal function checking for Queen part3

The part of the function that checks for piece type King is still being developed.

- **movePiece:** movePiece is a void function that takes in two parameters (position1 and position2) which are pointers to the struct t_Positions. This function asserts for two different positions and then calls the Is_Legal function. If the Is_Legal function returns a value of 0, the move function displays an error message to the player. When the move function receives a return value of 1 or 2 from the Is_Legal function, it moves the piece from position1(initial) to position 2(final). Additionally, if the return value is 2, it replaces the piece in position2 with the piece in position1 indicating that the piece has been captured.

```
/* Move Piece */
void movePiece(t_Position *position1, t_Position *position2){
        /* Check if position1 and position2 exist */
        assert(position1);
        assert(position2);

        if(Is_Legal(position1, position2) == 0){
                /* Pop-up window */
                /* Error: Illegal move */
        }

        else if(Is_Legal(position1, position2) == 1){

                /* Change the properties of new positoin */
                position2->Piece = position1->Piece;
                position2->Piece->Position = position2;

                /* Change the properties of initial position */
                position1->Piece = NULL;
        }

        else{

                /* Change properties of position2 */
                // TODO: Add piece of position2 to graveyard
                position2->Piece = position1->Piece;
                position2->Piece->Position = position2;

                /* Change properties of position1 */
                position1->Piece = NULL;
        }
}
```

Figure 17 - movePiece function

- **Castling:** Castling is a void function that takes in two parameters (position1, position2) which are pointers to the struct t_Positions. It checks if castling is a legal move.
- **En_Passant:** En Passant is a void function that takes in two parameters (position1, position2) which are pointers to the struct t_Positions. It checks if the pawn can be captured when it is horizontally next to another pawn of the opponent player.
- **Check:** Check is an int function that takes in two parameters (piece, game) which are pointers to the struct t_Piece and t_Game respectively. It returns a value of 2 for checkmate, 1 for check, and 0 for neither.

```
/* Check */
int Check(t_Piece *piece, t_Game *game){
        /* Check that the piece is a king */
        assert(piece->Type == King);

        if(piece->Player == White){
                for(int i = 0; i < 8; i++){
                        for(int j = 0; j < 8; j++){
                                if(game->Board[i][j]->Piece->Player == Black){
                                        if(Is_Legal(game->Board[i][j], piece->Position) == 2){
                                                if(Game_Over()){
                                                        return 2;
                                                        /* Pop up for game over */
                                                        /* Black wins */
                                                }
                                                else{
                                                        return 1;
                                                }
                                        }
                                }
                        }
                }
        }

        else if(piece->Player == Black){
                for(int i = 0; i < 8; i++){
                        for(int j = 0; j < 8; j++){
                                if(game->Board[i][j]->Piece->Player == White){
                                        if(Is_Legal(game->Board[i][j], piece->Position) == 2){
                                                if(Game_Over(1, game)){
                                                        return 2;
                                                        /* Pop up for game over */
                                                        /* White wins */
                                                }
                                                else{
                                                        return 1;
                                                }
                                        }
                                }
                        }
                }
        }

        return 0;
}
```

Figure 18 - Check function

- **Computer strategy:** Computer strategy is a set of functions that generate the decision of the computer. First, the computer gets all of the possible moves of the piece based on the current state of the game. This allows the computer to find the best sequence of movements. Then, the computer uses the Minimax algorithm to generate the decision tree, and picks the best solution from the tree. The computer also takes the Alpha–beta pruning method to optimize the Minimax algorithm.
    - findAllMoves(t_Game *current_board): findAllMove is a function that takes the current board state (t_Game *current_board) as a parameter and return the list of all available next steps. It calls the function findMove continuously until there is no more possible movement, and then puts all the results in a list.

```
t_Game *findAllMoves(t_Game *current_board)
{
    t_Position *current_pos = NULL;// the current position to check for possible movement
    current_pos = (t_Position *)malloc(sizeof(t_Position));

    t_Game *available_moves = NULL;
    // Assume there are at most 300 possible movements for each state, it may change according to debug result
    available_moves = (t_Game *)malloc(sizeof(t_Game) * 300);

    for (int i = 0; i < 8; i++)// rows (Rank 1~8)
    {
        current_pos->Rank = i;
        for (int j = 0; j < 8; j++)// columns (File a~h)
        {
            current_pos->File = (char)('a' + j);
            if (current_board->board[i][j] != NULL)
            {
                // find all the available movement in this position
                findMoves(current_board, current_pos, available_moves);
            }
        }
    }

    return available_moves;
}
```

Figure 19 - findAllMoves function

- findMoves(t_Game *current_board, t_Position *current_pos, t_Game *available_moves): findMoves takes the current game state, the position to check and the array of t_Game, which is used to save all of the available movements and to be returned as result, as parameters. It tries all of the movements of the piece in the current_pos position and adds the available ones into the array available_moves. This function returns nothing because the available_moves is passed by reference, and every edit will directly change the content of this array.

```
void findMoves(t_Game *current_board, t_Position *current_pos, t_Game *available_moves, int *len)
{
    /*
        According to the type of the piece, try each movement that it can do.
        For example, if it is a pawn in the current_pos, we should check whether it can
        go forward, front-left, front-right, or not.
    */
    // Pawn
    if (current_pos->Piece->Type == Pawn)
    {
        t_Position *next_pos;
        next_pos->Piece = current_pos->Piece;
        int direction = 1;// default the piece is White
        if (current_pos->Player == Black)// change to Black
        {
            direction = -1;
        }

        /* Check its front */
        next_pos->Rank += direction;
        next_pos->File = current_pos->File;
        // what is the use of floats?
```

Figure 20 - findMoves function part 1

```
if (Is_Legal(current_pos, next_pos) != 0)
{
    /* Change the state of the game, record it to the available_move, and then undo it.
       There should be a proper way to change the whole game state, but not just change
       the piece on these two position.
       Thus, we should complete the movePiece() function and add the undo() funcition.
     */
    movePiece(current_pos, next_pos);
    available_moves[len] = newGamestate;
    //undo();
}

/* Check its front-left */
next_pos->Rank += direction;
next_pos->File -= 1;
if (Is_Legal(current_pos, next_pos) != 0)
{
    movePiece(current_pos, next_pos);
    available_moves[len] = newGamestate;
    //undo();
}

/* Check its front-right */
next_pos->Rank += direction;
next_pos->File += 1;
if (Is_Legal(current_pos, next_pos) != 0)
{
    movePiece(current_pos, next_pos);
    available_moves[len] = newGamestate;
    //undo();
```

Figure 21 - findMoves function part 2

- minimaxDecTree(int depth, t_Game *game, int alpha, int beta, int is_bot_turn): minimaxDecTree is a recursive function that takes the depth of the decision tree, the movement arrays (t_Game *game), integer alpha & beta used for Alpha–beta pruning, and integer is_my_turn, which is used to distinguish between computer's turn and player's turn. The initial value of alpha & beta should be set as negative infinite and positive infinite.

  First, the computer recursively constructs a tree of all possible moves of the chess piece based on the given depth, and calculates the weights of all the child nodes using the method evaluateBoard.

  Then, according to different chess colors, the parent node takes the maximum or minimum value of the child node. If the white child, the maximum value of the child node is returned to the parent node, and vice versa.

```
int minimaxDecTree(int depth, t_Game *game, int alpha, int beta, int is_bot_turn)
{
    if (depth == 0)
    {
        return -evaluateBoard(game);
    }

    t_Game *newGameMoves = findAllMoves(game);

    if (is_bot_turn == 1)// it is my turn
    {
        int bestMove = -9999;
        for (int i = 0; i < SIZE; i++)
        {
            bestMove = MAX(bestMove, minimaxDecTree(depth-1, game[i], alpha, beta, is_bot_turn^1));
            alpha = MAX(alpha, bestMove);
            if (beta <= alpha)
            {
                return bestMove;
            }
        }
    }
```

Figure 22 - minimaxDecTree function part 1

```
    else
    {
        int bestMove = 9999;
        for (int i = 0; i < SIZE; i++)
        {
            bestMove = MIN(bestMove, minimaxDecTree(depth-1, game[i], alpha, beta, is_bot_turn^1));
            beta = MIN(beta, bestMove);
            if (beta <= alpha)
            {
                return bestMove;
            }
        }
    }
}
```

Figure 23 - minimaxDecTree function part 2

- evaluateBoard(t_Game *game): evaluateBoard is a function that takes one of the possible movement cases as a parameter and calculates the total weight of the pieces on the board. It calls the getPieceValue function to get the weight of the piece on each position.

```
int evaluateBoard(t_Game *game)
{
    int total = 0;
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            total += getPieceValue(game, i, j);
        }
    }
    return total;
}
```

Figure 24 - evaluateBoard function

- getPieceValue(t_Game *game, int row, int column): getPieceValue is a function that takes the current game state, the row & column of the position to be evaluated as parameters and return the corresponding weight of the piece on that position. If there is no piece, the result will be zero. The weight of each kind of piece is different and got from lots of chess players' experience.

```
int getPieceValue(t_Game *game, int row, int column)
{
    if (game->Board[row][column] == NULL)
    {
        return 0;
    }

    t_Piece *current_piece = game->Board[row][column]->Piece;
    int isWhite = 1;// default for white
    if (current_piece->Player == Black)
    {
        isWhite = -1;
    }

    if (current_piece->type == Pawn)
    {
        return 10 * isWhite;
    }
    else if (current_piece->type == Knight)
    {
        return 30 * isWhite;
    }
    else if (current_piece->type == Bishop)
    {
        return 30 * isWhite;
    }
```

Figure 25 - getPieceValue function part 1

```
    else if (current_piece->type == Rook)
    {
        return 50 * isWhite;
    }
    else if (current_piece->type == Queen)
    {
        return 90 * isWhite;
    }
    else if (current_piece->type == King)
    {
        return 900 * isWhite;
    }
}
```

Figure 26 - getPieceValue function part 2

- **insufficientMaterials:** insufficientMaterials is an int function that takes in a single parameter (game) which is a pointer to the struct t_Game and returns 1 if there is a draw due to insufficient materials and 0 otherwise. The function first checks if there are sufficient materials (pawns, queens, or rooks exist) and returns 0 if there are. Otherwise, it counts the number of Knights and Bishops each player has. If both players have only a

King, or only a King and a Bishop, or only a King and a Knight, the function returns 1 indicating the game is drawn due to insufficient materials.

```
int insufficientMaterials(t_Game *game){

        /* Check if insufficient materials */
    int wBishop, wKnight, bBishop, bKnight;
    for(int i = 0; i < 8; i++){
        for(int j = 0; j < 8; j++){

            /* If Sufficient Materials exist */
            if(Board[i][j]->Piece->Type == Pawn || Board[i][j]->Piece->Type == Rook || Board[i][j]->Piece->Type == Queen){
                return 0;
            }

            /* Count the number of Bishops each player has */
            else if(Board[i][j]->Piece->Type == Bishop){
                if(Board[i][j]->Piece->Player == White){
                    wBishop += 1;
                }
                else{
                    bBishop += 1;
                }
            }

            /* Count the number of Horses each player has */
            else if(Board[i][j]->Piece->Type == Knight){
                if(Board[i][j]->Piece->Player == White){
                    wKnight += 1;
                    }
                else{
                    bKnight += 1;
                }
            }
        }
    }
    if((wBishop == 1 || wKnight == 1) && (bBishop == 1 || bKnight == 1)){
        return 1;
    }

    /* Game is not over */
    return 0;
}
```

Figure 27 - insufficientMaterials function

3.3 Detailed description of input and output formats

- **Syntax/format of a move input by the user**
    - The user input for moving pieces will be through mouse clicks. The user will use their mouse to click on the GUI to select their chess piece and then they will click where they want it to move on the chess board.
- **Syntax/format of a move recorded in log file**
    - Chess moves will be recorded and saved in a text file for each game. This log will have a format similar to what is shown in the following figure:

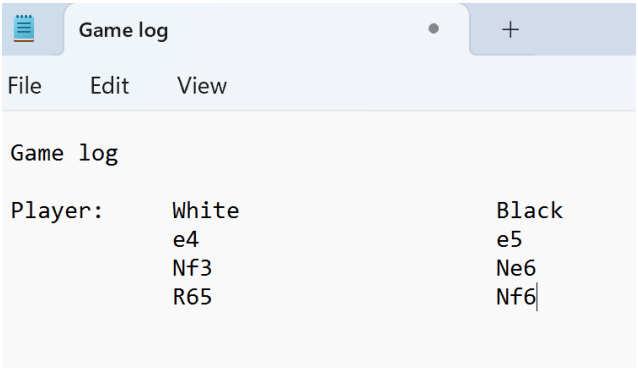

Figure 28 - Example text file for chess game log

Development plan and timeline

4.1 Partitioning of tasks

4/11/23

Setup version control with GIT to facilitate team software development

4/12 - 4/23

- Chess Board
- Computer/opponent
- Chess pieces and their movement
    - Pawn
    - Knight
    - Bishop
    - Rook
    - Queen
    - King
    - Special moves (Castling, en passant)
- GUI

4/24/23

Software alpha version completed

5/1/23

Software release

4.2 Team member responsibilities

Nishant Mehendre - Data Structure, Moving pieces, Check, Endgame

Meghana Burugupalli - Chess pieces and their movement

Priyanka Samavedam - Setup version control with GIT/ Chess board

Alex Dunn - GUI

Xinrui Xie - Computer/opponent

Copyright

References

https://www.chess.com/terms/chess

-Resource was used to help gain a better all round knowledge of the game of chess

Index