Computer Chess

Software Specification Version 3

https://www.publicdomainpictures.net

Software Engineering In C
EECS 22L

Instructor
Quoc-Viet Dang

University of California Irvine
Team # 4 Deep Core Dump
Ameya Pandit
Matthew Dunn
Richard Duong
Yuming Wang
Yunhe Shao
Xingjian Qu

Table of Contents

Glossary

- **AI.c:** Module that will generate moves for the computer player
- **Board.c:** Module that will perform task related to the board
- **Moves.c:** Module that follows the rules of chess
- **ChessTree.c:** Module that performs task on TNODE
- **CopyBoard:** Function that copies the board
- **CreateBoard:** Function creates new board
- **Game:** Holds data and variables for all of game modules
- **GameBoard:** Holds variables related to the board
- **GameLog.c:** Module that allows for moves to be saved to a log, contains undo functionality
- **GameTree:** Data structure that holds all possible moves for a board
- **GetUserInput:** Function to get user input
- **Linux:** Operating system that the program runs on
- **LNODE:** Node of the MovesList tree, containing pointers to the next and previous node
- **Makefile:** File holding all of the compilations
- **Menu.c:** Module that prints ASCII menus of the user input
- **MobaXterm:** Application used to display the gui
- **MovesList.c:** Contains functions to perform task on the chess board tree

- **PrintBoard:** Prints the ASCII game board
- **Testing:** Function used for debugging and testing
- **TNODE:** A tree node containing a board and a double linked list to other nodes
- **Tree:** The structure of the data that the program is based on

1 Software Architecture Overview

1.1 Main Data Types and Structures

This chess game uses 4 main data structure components GameBoard, GameTree, MovesList and Game. These structures encapsulate all of the needed variables and data that is needed for game play. A visual representation for these structures is given in **Figure 1 and Figure 2** below.

1: GameBoard. This data structure contains all of the variables that are needed for various game play operations and related to the chess board. This struct contains the column, score and current player, as well as an array to store the player locations on the chess board. This struct is represented in **Figure 1** (log list) below as GBOARD.

4:GameTree. This data structure is designed to contain the the list of all possible moves that could be made from one game board configuration. The tree consists of tree nodes (TNODE). Each tree node contains a board and a double linked list that points to another tree node. This structure is show visually in **Figure 2** (tree) below.

3: MovesList. This data structure is designed using a double linked list. The list has a head (MLIST) and then nodes (LNODE) which contain the usual pointers to next, prev, head but also contain a pointer to a tree node in the tree data structure. The primary function of the list is to allow access to each tree node where the possible game boards are contained. It is possible to append an indefinite number of move configurations into the tree. This structure is also used for the log list. This structure is shown in **Figure 1** (log list) and **Figure 2** (tree) below.

4: Game. This data structure is designed to encapsulate the variables and data that are needed for all of the game play modules and their respective functions. This structure can be passed to the various functions that are called for during the game loop and contains the current game board, a list of all of the previous game boards and a variable that stores the current win condition. This struct is shown in **Figure 1** named GAME.

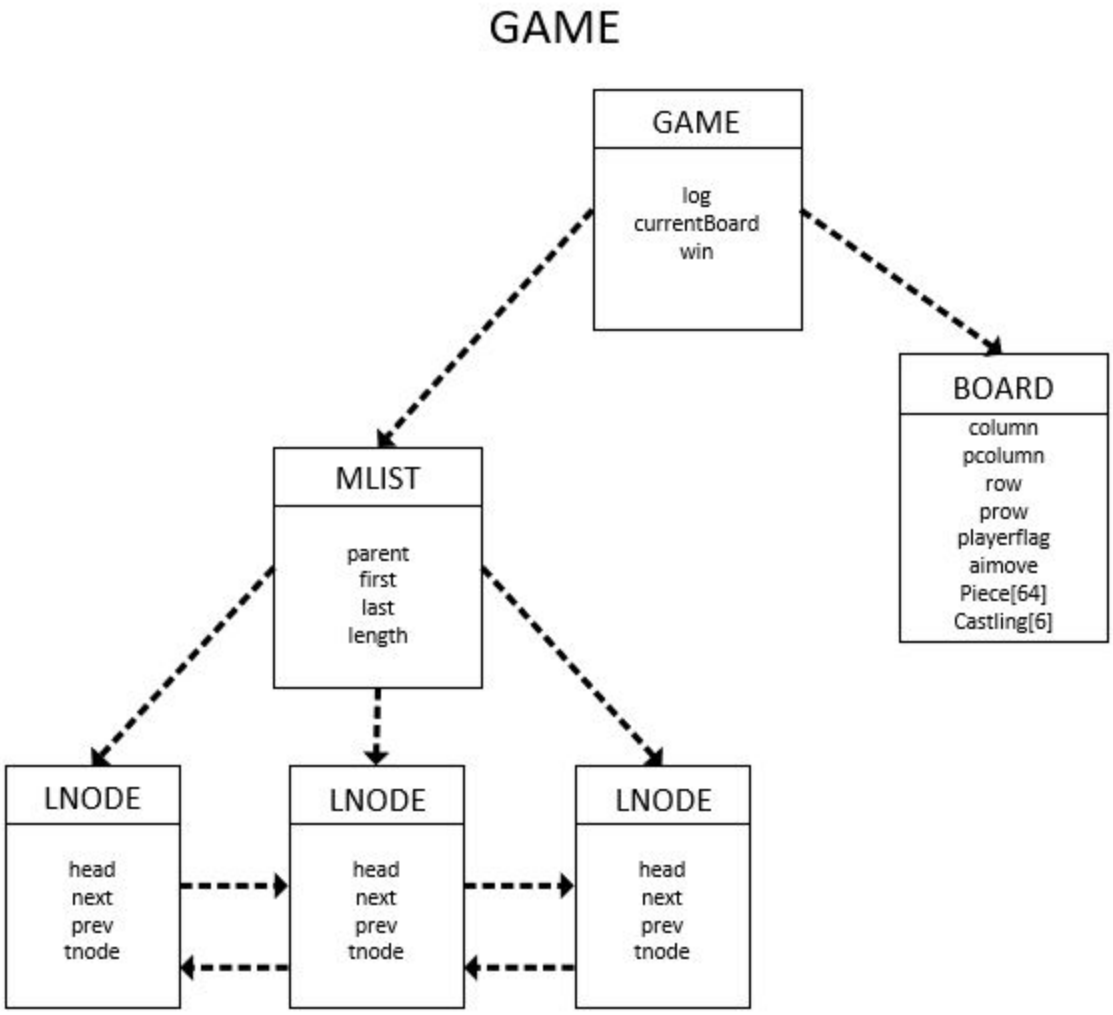

Figure 1: Game Data Structure

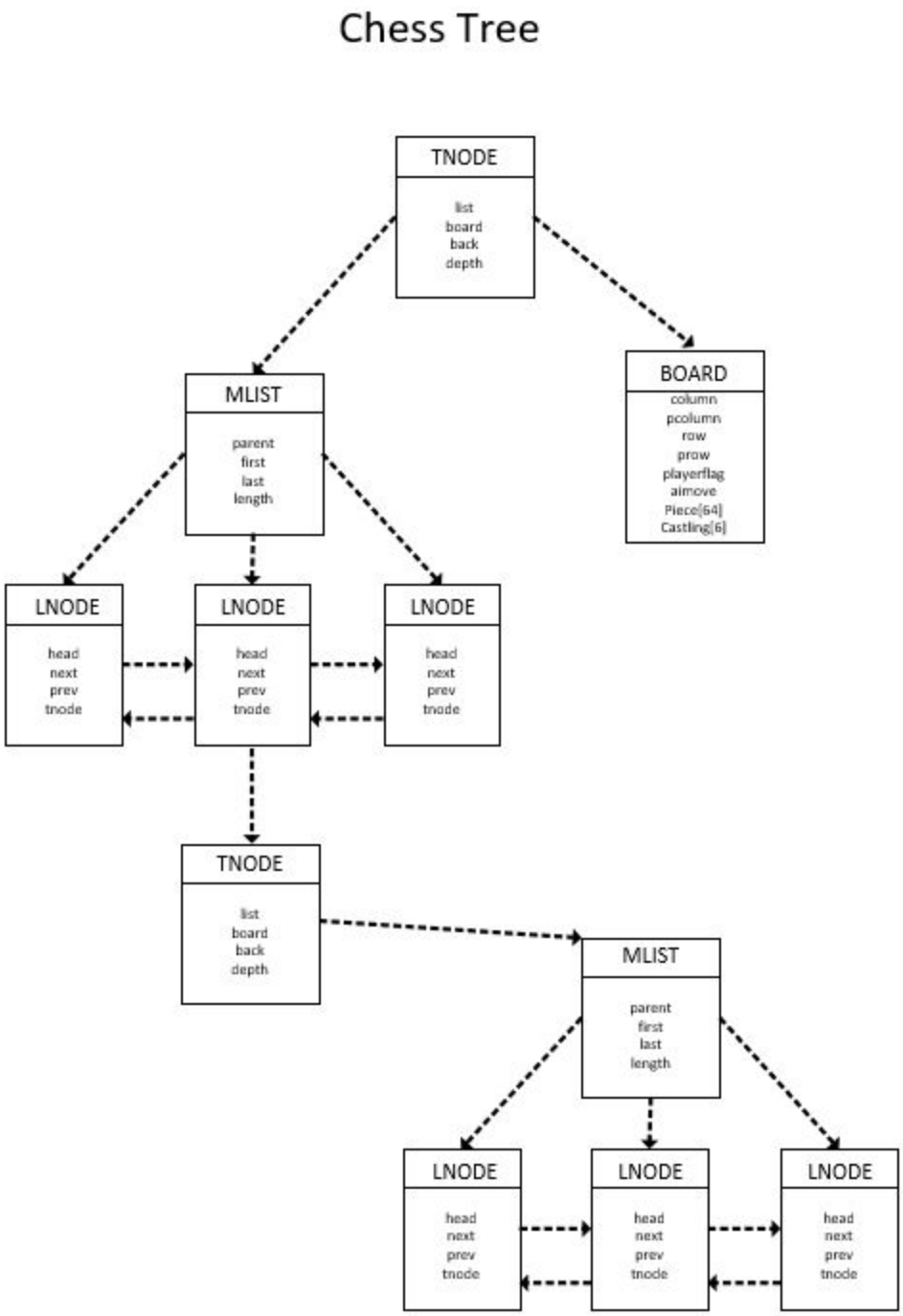

Figure 2: Tree of possible moves

1.2 Major software components

The chess game is decomposed into several modules which are compiled as object files. These files and the C library are then linked when the executable file is compiled. Each module has a header file and a source code file. The source code file ending in .c is the file that contains all of the functions that relate to the module. The header file ending in .h is the file that contains the function headers for all of the functions. The respective header file is included in each module were these functions are to be called. The hierarchical composition of the chess game is shown in **Figure 3**.

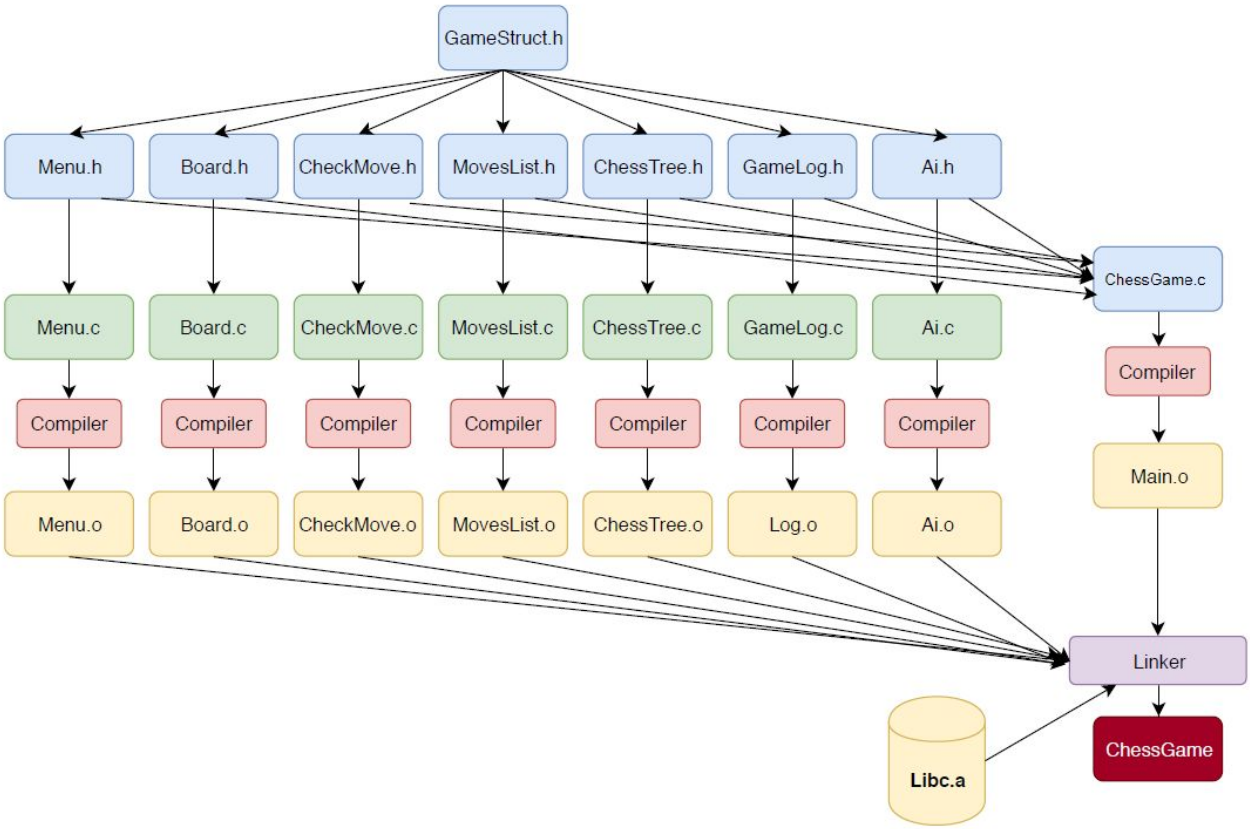

Figure 3: Module Hierarchy

1.3 Module Interfaces

Each of the 8 program modules contains the following API.

1: ChessIO.c

This module contains the functions to print the ASCII menus of the user input. The major function definitions are as follows:

Gets game type from the user

```
int GetGameType(void);
```

Output: returns integer between 0-1

Gets game color from the user

```
int GetGameColor(void);
```

Output: returns integer between 1-2

Gets board type from user

```
int GetBoardType(void);
```

Output: returns integer between 2-1

2: Board.c

This module contains the functions that will perform all of the tasks related to the chess board. The major function definitions are as follows:

Initializes and prints the ASCII game board

```
void PrintBoard(GBOARD *board);
```

Input : Game struct

Output: prints the board to the screen.

Creates a new game board

```
int CreateBoard(int CosFlag, GBOARD *board);
```

Input :

Output: returns a empty board

Copies an existing game board.

```
int CopyBoard(GBOARD *board);
```

Input: board

Output:

Gets the custom piece location for a special board setup

```
int GetUserInput();
```

Input : Numbers of rows and columns.

Output: int that is corresponding to the actual location.

3: Moves.c

This module contains the functions that pertain to the rules of chess. The function here can be called to check for valid moves and generate moves for the current board. The major function definitions are as follows:

Checks if the player made a valid move returns a game pointer

```
void checkmove(TNODE* allmoves);
```

Input: TNODE pointer that points to the current board

Output: TNODE pointer holding all possible moves

Description: Iterates through the board to determine what pieces move to check

Generates all the possible moves for pawn returns a list pointer

```
void *pawnmoves(GBOARD *board, int column, int row, TNODE
*tree);
```

Input:GBOARD pointer, two ints for position, and TNODE pointer

Output: Adds all possible pawn moves to the tree pointer

Description: Checks where the pawn is and what possible moves it has

Generates all the possible moves for rook returns a list pointer

```
void *rookmoves(GBOARD *board, int column, int row, TNODE
*tree);
```

Input:GBOARD pointer, two ints for position, and TNODE pointer

Output: Adds all possible rook moves to the tree pointer

Description: Checks where the rook is and what possible moves it has

Generates all the possible moves for queen

```
void *queenmoves(GBOARD *board, int column, int row, TNODE
*tree);
```

Input:GBOARD pointer, two ints for position, and TNODE pointer

Output: Adds all possible queen moves to the tree pointer

Description: Checks where the queen is and what possible moves it has

Generates all the possible moves for king returns a list pointer

```
void *kingmoves(GBOARD *board, int column, int row, TNODE
*tree);
```

Input:GBOARD pointer, two ints for position, and TNODE pointer
Output: Adds all possible king moves to the tree pointer
Description: Checks where the king is and what possible moves it has

Generates all of the possible moves for bishop returns a list pointer

```
void *bishopmoves(GBOARD *board, int column, int row, TNODE
*tree);
```

Input:GBOARD pointer, two ints for position, and TNODE pointer
Output: Adds all possible bishop moves to the tree pointer
Description: Checks where the bishop is and what possible moves it has

Generates all of the possible moves for knight returns a list pointer

```
void *knightmoves(GBOARD *board, int column, int row, TNODE
*tree);
```

Input:GBOARD pointer, two ints for position, and TNODE pointer
Output: Adds all possible knight moves to the tree pointer
Description: Checks where the knight is and what possible moves it has

Checks for check condition returns a bool value

```
bool CheckCheck(GBOARD *board, int flag);
```

Input: GBOARD pointer of the current board, an int flag
Output: A boolean variable determining whether a player is in check
Description: Finds the appropriate king and checks if it is in check

Checks for checkmate or stalemate condition returns an int value 1 for true

```
int checkmate(GBOARD *board);
```

Input: GBOARD pointer of the current board
Output: An int determining the winner of the game, or if no winner is determined
Description: Calls check first, if true determines if there are any moves

Checks if one of the players have won

```
void checkwin(GAME *game);
```

Input: GAME pointer to the current game
Output: Changes int win in the game struct if a player has won

Description: Checks if any of the players have won, if so ends the program

Checks if the en passant is available for the pawn

```
void Enpassant(TNODE *tree, GBOARD *board, int column, int row);
```

Input: TNODE pointer, GBOARD pointer, two ints for position
Output: Adds the possible moves to the tree pointer
Description: Checks if the conditions for an en passant are met

Checks if the king is able to perform either castle

```
void Castling(TNODE *tree, GBOARD *board, int column, int row);
```

Input: TNODE pointer, GBOARD pointer, two ints for position
Output: Adds the possible moves to the tree pointer
Description: Checks if the conditions for castling are met

Checks if there is a stalemate

```
int stalemate(GBOARD *board);
```

Input: GBOARD pointer of the current board
Output: an int determining whether or not there is a stalemate
Description: Checks if the current board is in stalemate

4: ChessTree.c

This module contains the functions to perform the needed tasks on the double linked list MLIST. The major function definitions are as follows:

Deletes the whole tree

```
void DeleteTree(TNODE *node);
```

Input: TNODE struct
Output: none
Description: deletes a tree and all nodes

Append a list to the tree node

```
void AppendList(TNODE *node, MLIST *list);
```

Input: TNODE struct, MLIST struct
Output: none
Description: appends a list to the tree node and handles all the pointers.

Creates a tree for the AI

```
TNODE *CreateTree(void);
```

Input: none
Output: returns a TNODE struct
Description: creates a tree for the game moves

5: MovesList.c

This module contains the functions to perform the needed tasks on the chess board tree TNODE. The major function definitions are as follows:

Create an list and initialize for appending nodes returns a pointer to the list

```
MLIST *CreateMoveList(void);
```

Input: none
Output: MLIST pointer
Description: returns a list pointer to the list created

Deletes a complete list and all of its nodes

```
void DeleteMoveList(MLIST *list);
```

Appends a node to the end of the list

```
void AppendListNode(TNODE *tnode, GBOARD *gboard);
```

6:GameLog.c

This module contains the functions that will allow for a moves log to be saved at the end of the game. The major function definitions are as follows:

Saves a log of moves to a file.

```
void logOfMoves(TNODE *tree);
```

Input: pointer to a tree; follows struct set-up to retrieve struct GameBoard (tree->list->first->tnode->board)
Output: log file containing all the moves in the game
Description: creates text file containing all the moves in the game

"Undos" a previous move if competing against player client

```
void undoPlayer(GAME *game);
```

Input: pointer to a game; follows struct set-up to retrieve previous GameBoard (game->log->tree->list->last->tnode->board)
Output:
Description: copies previous board set-up (preceding current board) and sets current board to previous board set-up in event of an undo call.

"Undos" a previous move if competing against computer client

```
void undoComputer(GAME *game);
```

Input: pointer to a game; follows struct set-up to retrieve previous GameBoard (game->log->tree->list->last->tnode->board)
Output:
Description: copies board set-up preceding computer's turn (two preceding current board) and sets current board to previous board set-up in event of an undo call.

Adds the current board set up to the list for undo and log functionality

```
void AddMoveToLog(GAME *game);
```

Input: pointer to a game; follows struct set-up to retrieve previous GameBoard (game->log->tree->list->last->tnode->board)
Output:
Description: Adds game board set up as a node to the linked list data structure.

7: Ai.c

This module contains the functions that will generate the moves for the computer turn in the game. The major function definitions are as follows:

AI function for generating next move which returns a pointer to a board

```
GBOARD *GetBestMove(GAME *game);
```

Input: GAME struct for the game being played
Output: GBOARD struct for the best board setup detected
Description: detects the best move possible

8: Main.c

This module contains the main function and the game play loops for each of the game play modes. There are no other function definitions in Main.c

10. Playersmove.c

This module contains the functions that print the text and get input from user to select which player and what coordinate to move. For now it also contains the promote function for the pawn. The major function definitions are as follows:

Get input from user for piece type and coordinate

```
int playersmove(GAME *game);
```

Input: pointer to the game struct
Output: Returns a 0 if the game is over, otherwise returns a 1
Description: Asks user for input and validates whether their move is possible

Checks if there are any pawns that need to be promoted

```
void checkpromote(TNODE *treenode);
```

Input: pointer to treenode inorder to access the current board
Output: Changes the pawn to the desired piece that the player wants
Description: Checks the top row for white pawns and the bottom row for black pawns and promotes them to the piece of their respective colors

1.4 Overall Program Control Flow

The control flow for the chess game is as shown in **Figure 4**. In the user setup section the user will be prompted to enter the parameters that they are allowed to change such as the color, type of game (PVP, PVC, CVC), and also the game board configuration either standard or custom.

Once the game loop is started the player will be asked for their move if they are first, or depending on the type of game the other player or computer will make their move. Once the move is decided then the move will be checked for validity and the end conditions such as check, checkmate or stalemate will be checked.

If there is no end condition the game loop will repeat until the condition occurs. Once an end condition occurs the game loop will be ended, the winner will be printed moves logged and saved, memory cleaned up and the game will go to the end condition.

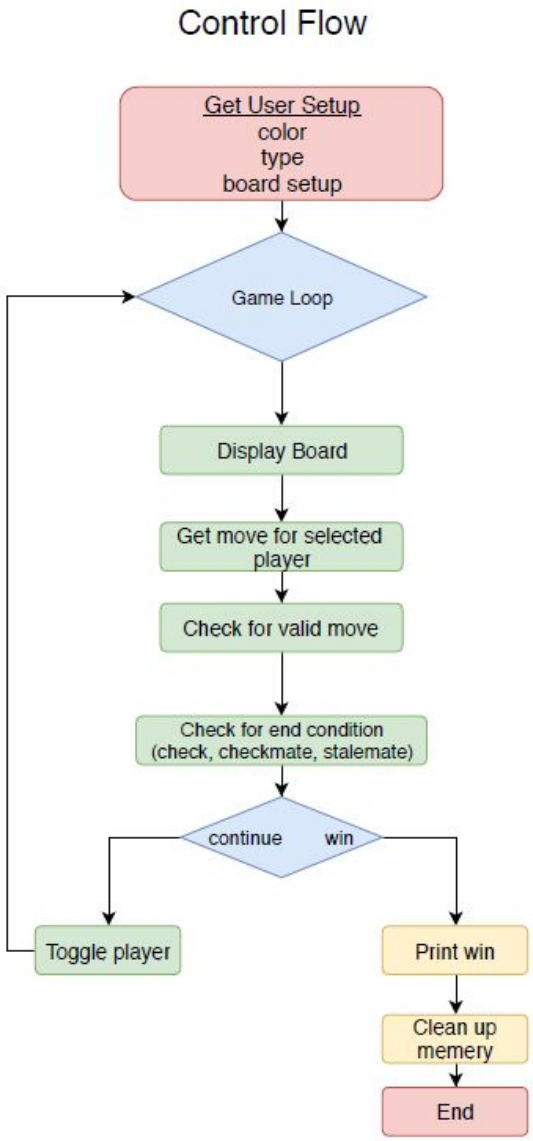

Figure 4: Control flow for typical game

2 Installation

2.1 System Requirements, Compatibility

Hardware: Linux Server
Software Linux: CentOS release 6.10
More, the Linux application: MobaXterm

2.2 Setup and Configuration

The first step to setting up the program is to download the zip file.

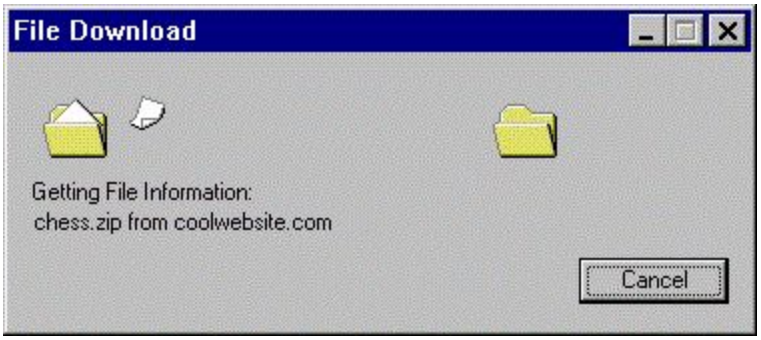

Figure 5: Downloading the zip file

Once the file has finished downloading, locate the zip file, for most web browsers the zip file will be placed in the download folder. Next, using an application to view zip files, open and extract all of the contents of zip file. Locate where the zip file was extracted. Once the zip file is extracted, there will be a Chess Program file, we copy and paste the file into our terminal, which is Linux, using the following command: "`gtar xvzf chessprogram.tar.gz`". Then switching to the directory /chess/bin/chess/. Then type the "make" command to compile the code file to execute.

In-game settings like audio or video can be adjusted once the game has launched. Click on the settings and go to the audio or video tab to adjust. The configuration is supposed to be the normal chess game, player interfaces are mainly based on the brown and black color. The images we used are all like mid-century style.

2.3 Building, Compilation, Installation

The zip file consist of : Makefile, Menu.c, Board.c, Board.h, ChessGame.c, ChessStruct.h, ChessTree.c, ChessTree.h, GameLog.c, GameLog.h, MovesList.c, MovesList.h, test.h, AI.c, AI.h, CheckMove.c, and CheckMove.h. All the compilation is done within the makefile that is included in the zip file. When the command "`make`" is entered into the console, the object file for each c file is made and the gcc command as seen in **Figure 6** will link all the object files and make the executable file ChessGame.

```
gcc -ansi -std=c99 -Wall -g -c ChessIO.c -o ChessIO.o
gcc -ansi -std=c99 -Wall -g -c ChessTree.c -o ChessTree.o
gcc -ansi -std=c99 -Wall -g -c MovesList.c -o MovesList.o
gcc -ansi -std=c99 -Wall -g -c Ai.c -o Ai.o
gcc -ansi -std=c99 -Wall -g -c Board.c -o Board.o
gcc -ansi -std=c99 -Wall -g -c Main.c -o Main.o
gcc -ansi -std=c99 -Wall -g -c GameLog.c -o GameLog.o
gcc -ansi -std=c99 -Wall -g -c Moves.c -o Moves.o
gcc -ansi -std=c99 -Wall -g -c playersmove2.c -o PlayersMove.o
gcc -ansi -std=c99 -Wall -lm Main.o ChessTree.o Moves.o MovesList.o
```

Figure 6: The gcc command used to link object files and make the executable

Before launching the game, ensure that the MobaXterm program is running. If not, the gui may not launch and cause other problems when trying to launch the game. Running the executable file can be done with the command " . `/ChessGame`". At this point the game will launch and the main menu will be shown. To exit the game, simply close the window or click on "exit game" in the menu.

To remove the unnecessary files from the directory where the zip file was copied to, type the command "`make clean`". All the object files and the executable file will be removed, leaving only the files extracted from the zip file.

3 Documentation of Packages, Modules, Interfaces

3.1 Detailed Description of Data Structures

```
typedef struct GameBoard{
    char colom;
    char pcolum;
    int row;
    int prow;
    int aimoves;
    int *playerflag;
    char piece[64]
    int castling[6]
};
```

Figure 7: Game board struct.

The GameBoard struct who's source code is displayed in **Figure 7** is for the Chess Board for both player and AI. The structure will take care of each player's total score on the board (as different pieces are assigned with a score value)and then compare them together to get the best move. The column and row section will be used to store each player's move. The player flag is used as identifier for turn indication. The array Piece[64] is where the game pieces are stored. The castling[6] keeps track of the movement of the rooks and king for castling condition. Aimove is to record if it is the Ai making a move or a player. Capital letter is used for white player, player one and lower case letter is used for black player, player two.

```
typedef struct TreeNode{
    MLIST *list;
    GBOARD *board;
    TNODE *back;
    int depth;
}TNODE;
```

Figure 8: Tree node struct

The TreeNode struct who's source code is displayed in **Figure 8** is used to create a node in the tree of possible moves. This node has a double linked list pointer that points to a list of other tree nodes. It also has a pointer to a game board struct so that a possible game board can be stored here. The pointer back will point back to the parent tree node. Depth is an integer value and contains the depth at which this node exists in the tree.

```
typedef struct MovesList{
    TNODE *parent;
    LNODE *first;
    LNODE *last;
    int length;
}MLIST;
```

Figure 9: Move list struct

The MoveList struct who's source code is displayed in **Figure 9** is used to create a list for possible moves from a certain board configuration. This struct contains the pointers needed to access the first and last item in the list respectively first and last. It also contains a pointer that points to the parent tree node to which the list has been appended. Length is the variable for how many items are appended in the list.

```
typedef struct ListNode{
    MLIST *head;
    LNODE *next;
    LNODE *prev;
    TNODE *tnode;
}LNODE;
```

Figure 10: List node struct

The MoveList struct who's source code is displayed in **Figure 10** is used to define a node in the double linked list. This node has pointers head, next and prev. Head points to the top of the list, next is the pointer to the next item in the list and prev is a pointer to the previous item.

```
typedef struct Game{
    MLIST *log;
    GBOARD *currentBoard;
    int win;
    int difficultyLevel;
}GAME;
```

Figure 11: Game struct

The Game struct who's source code is displayed in **Figure 11** is used to encapsulate the important game variables and history. This struct contains a variable for win who's value can be changed depending on if a win condition is detected or the game will continue. The log pointer is a pointer to a list of moves. This is for the history log and the undo function. Each time a move is made, a copy of the board can be stored in this list for retrieval at a later time. The currentBoard pointer is a pointer to the current game board at any point in game play. Difficulty level stores the user input for difficulty level in the game.

Image 3: 3.2 Detailed Description of Functions and Parameters

```
void PrintBoard(GBOARD *board);
```

This function will print out the Board struct that print the information that store as a 1-D array out as a complete chess board. Nothing will be returned by the function, only the board will be printed out.

```
int CreateBoard(int CosFlag, GBOARD *board);
```

This function will initialize a chess board. CosFlag is the variable that determine if the user choose to custom the board or not. If the user choose not to custom the starting board, the board will be initialized as the standard way. Else, the board will be initialized as the user want.

```
int Testing(GBOARD *board);
```
This is a temporary function that will be used for debug and testing.

```
int GetUserInput();
```
GetUserInput function will be used to get what the user input during the game such as position coordination and commands such as undo, hint etc,. The function will return an int that either be the converted coordination or the code that represent the command that the user has entered.

```
GBOARD *CopyBoard(GBOARD *board);
```
CopyBoard function will be used to create another separated board struct to be stored in the log function.

```
void PrintMainMenu(void);
```
PrintMainMenu function will be used to print out the text for the main menu options that allows the user to choose the type of game they would like to play. The options that are printed are 1:player vs player, 1: player vs computer and 3:computer vs computer.

```
void PrintColorMenu(void);
```
PrintColorMenu function will be used to print out the text for the menu that allows the user to choose the color of player they would like. Options are either 1:b (black) or 2:w (white).

```
void PrintMoveMenu(void);
```
PrintMoveMenu function will be used to print out the menu allowing the user to select the move they would like to make. The user input that is asked for is in the format of letter.number. For example a3.

```
void *AppendList(TNODE *node,MLIST *list);
```
This function will append a list of moves to a node in a tree. The input parameters are a pointer to a list to be appended and a pointer to a node to which the list will be appended into. The function handles all of the pointers and returns void.

```
void *DeleteTree(TNODE *node);
```

This function will delete a board tree. The tree is deleted completely including all the lists, boards and nodes in the tree. Recursion is used to complete this task. The input parameter is a pointer to a tree. The function returns a void type.

```
TNODE *CreateTree(void);
```
This function is used to initially create a board list tree The function handles the memory allocation for the tree and returns a pointer to the tree.

```
MLIST *CreateMoveList(void);
```
This function is used to create a list for the possible moves. A empty list is initialized and the pointers are set to null. The pointer to the list is returned.

```
void *DeleteMoveList(MLIST *list);
```
This function is used to delete a board list completely. All of the nodes, tree nodes and boards will be deleted. The input parameters are a pointer to a list.

```
void *AppendListNode(TNODE *tnode,GBOARD *board);
```
This function will append a node to a move list. The input parameters are a pointer to a tree node and a pointer to a game board.

```
void logOfMoves(TNODE *tree);
```
This function will save the log of game moves to a file. The input parameters are a pointer to a game. The function will loop through the list of game moves and save them in a format of color, piece, current column, current row, next column and next row. This format is repeated for all of the moves in the game. The output is saved in a text file.

```
GBOARD *GetBestMove(GAME *game,int Level);
```
This function is the AI for game play. The function input parameters are a pointer to the game struct and a level for the depth the function will evaluated to. The algorithm that is used for this function calls the generate game move functions to generate a tree that is one level deep. Minimax is then performed on the tree to determine the best move.

```
int *checkmove(GAME *game, int position);
```
This function will check if the move that the player has choosen is a valid move. The input parameters are the game struct where the current board is stored and an integer

for the location where the player would like to move to in the linear game board array. The function returns a 1 if the move was valid and a 0 if the move was not valid.

```
MLIST *pawnmoves(GBOARD *currentBoard, int position);
```
This function generates a list of all the possible moves that a pawn can make from its current location on the gameboard. The input parameters are a game board pointer and a integer for position of the piece on the linear game array. A pointer to a list is returned that contains the possible legal moves that the pawn can make.

```
MLIST *queenmoves(GBOARD *currentBoard, int position);
```
This function generates a list of all the possible moves that a queen can make from its current location on the gameboard. The input parameters are a game board pointer and a integer for position of the piece on the linear game array. A pointer to a list is returned that contains the possible legal moves that the queen can make.

```
MLIST *kingmoves(GBOARD *currentBoard, int position);
```
This function generates a list of all the possible moves that a king can make from its current location on the gameboard. The input parameters are a game board pointer and a integer for position of the piece on the linear game array. A pointer to a list is returned that contains the possible legal moves that the king can make.

```
MLIST *rookmoves(GBOARD *currentBoard, int position);
```
This function generates a list of all the possible moves that a rook can make from its current location on the gameboard. The input parameters are a game board pointer and a integer for position of the piece on the linear game array. A pointer to a list is returned that contains the possible legal moves that the rook can make.

```
MLIST *knightmoves(GBOARD *currentBoard, int position);
```
This function generates a list of all the possible moves that a knight can make from its current location on the gameboard. The input parameters are a game board pointer and a integer for position of the piece on the linear game array. A pointer to a list is returned that contains the possible legal moves that the knight can make.

```
MLIST *bishopmoves(GBOARD *currentBoard, int position);
```
This function generates a list of all the possible moves that a bishop can make from its current location on the gameboard. The input parameters are a game board pointer and a integer for position of the piece on the linear game array. A pointer to a list is returned that contains the possible legal moves that the bishop can make.

```
int checkmate(GBOARD *currentBoard);
```
This function checks to see if the king for a certain player is in checkmate in its current location. The input parameter for this function is a pointer to a game board. The function returns a bool value.

```
int check(GBOARD *currentBoard);
```
This function checks to see if the king for a certain player is in check in its current location. The input parameter for this function is a pointer to a game board. The function returns a bool value.

```
void undoPlayer(GAME *game);
```
This function allows for the undo feature in the game when competing against another human user. The input parameter is a pointer to the game struct. The function will delete the current game play situation and revert back to the previous game board. This function can be called until there are no moves left to undo.

```
void undoComputer(GAME *game);
```
This function allows for the undo feature in the game when competing against the computer client. The input parameter is a pointer to the game struct. The function will delete the current game play situation and revert back to the game board preceding the previous game board (last user move made before the computer made its move). This function can be called until there are no moves left to undo.

```
void AddMoveToLog(GAME *game);
```
This function adds the game board set up as a node to the linked list data structure. This is vital as this is the list that is to be referenced by the logOfMoves function, along with both the undo functions.

3.3 Detailed Description of Input and Output Formats

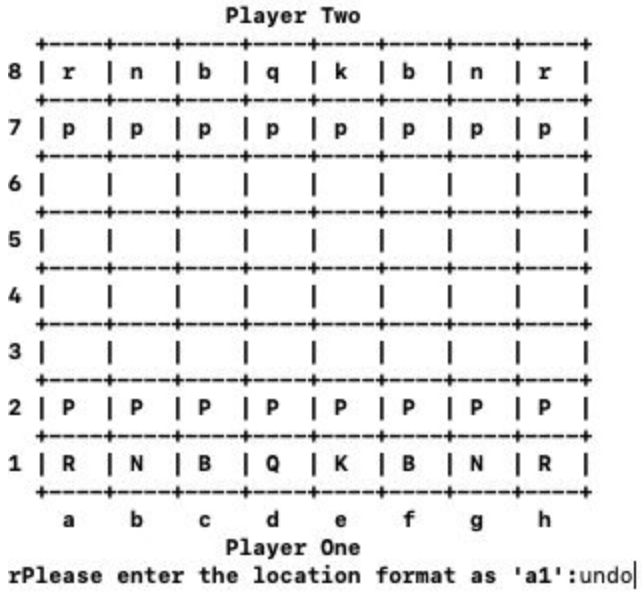

Figure 12: Board example

For the game play input format the player will be asked for 2 inputs. The first is what piece they would like to move. The message will print: "`Which piece would you like to move`". Then the player should type the right coordinate according to the board. And example of this would be column.row format as in (a3). The player that is selected with this input is the player in column 'a', and row '3'. For the output, the player will see the possible valid moves printed to the screen. Next the player will be asked to choose the next step of "`Which piece would you like to move`". The input format for choosing a location is the same as player selection of column.row format (a3). Once the player selects a location this move will be checked for validity. If the move player choose is wrong an error message will print and the player will be asked to select another move location.

The logOfMoves function, as aforementioned, will have an input parameter which has a pointer to the game that was being played. The pointer will have access to a node in a list of game moves, containing individual piece identity, current location, and the location from where the pieces moved from. The function will iterate through all the nodes in the list, thus scanning and having access to every piece movement. The function will output how each move unfolded, in the format of "[move number] : [chess piece] [previous

column] [previous row] to [current column] [current row].” The chess piece will be upper-case for a white move, and lower-case for a black move. Users will find the chess moves file in a text file that is produced. **Figure 13** is an example of how the text file might look like:

```
1: P e2 to e4
2: p e7 to e5
3: N b1 to c3
4: b f8 to c5
```

Figure 13: Text file output

In this example, the first move made is by the white pawn from coordinate e2 to coordinate e4. The next move is by a black pawn from coordinate e7 to e5. The third move is from a white knight from b1 to c3, and the final move is by the black bishop from f8 to c5.

4 Development Plan and Timeline

4.1 Partitioning of Tasks & Team Member Responsibilities

Table 1:

	Ameya Pandit	Matthew Dunn	Yunhe Shao	Richard Duong	Yuming Wang	Xingjian Qu
Software Development	GameLog Extra Functions	AI ChessTree ChessList	Board GUI AI	CheckMoves Extra Functions	CheckMoves Extra Functions	Main GUI
Team Roles	Presenter	Manager	Reflector	Reflector	Reflector	Recorder

Ameya is responsible for the development of the GameLog file, which creates a text file containing every single move committed by the human or computer client. The file also features undo functionality, an extra function that Ameya developed. The objective of this file is that any game played can be recreated, as there is a historical log

of all the moves. The moves are listed in order starting from the first move of the game till the final move of the game. The text file produced at the end of the game contains the details of each move: the color of the piece, the letter associated with the piece, the current location of the piece, and the location of where the piece moved to. Furthermore, Ameya is the presenter of the team; he is the liason for communication between the team he is part of and the instructor and other teams, ensures every member of the team has an input before seeking input from outside sources, and will manage crafting the presentation to showcase the game. Ameya also manages the team Github account, and offers assistance if teammates have issues with the service.

Matthew is responsible for developing ChessTree, ChessList, and the AI client. The first two contain functions that allow for list and tree manipulations. The tree and the list can grow or be appended, respectively, and nodes can be deleted. Being able to append and delete is imperative with the CheckMoves and the AI client; the CheckMoves client finds all possible moves on the chess board and adds them as nodes onto the list - these are valid moves, and if the user picks a valid move it should be present on this list. The AI client utilizes the ChessTree client to generate a tree from a position on the chess board with potential future moves. Once the AI client makes a decision, the entire tree is deleted and a new one forms with the next position in question. Matthew is also the manager of the team; he makes sure the team is prompt and focused, is responsible for time management, and values input from every single team member.

Yunhe is responsible for developing Board, GUI, and the AI client. The Board and GUI client are intertwined; the Board client updates the chess board after every move and keeps track of all the pieces on the board, while the GUI client expands upon the Board client and is the user interface on which the game is played. Yunhe is also working with Matthew on the AI client; together, Yunhe and Matthew will create an AI which creates a tree as deep as possible with possible moves from a certain position, and acts according to an arbitrary score that the client maintains to determine which move is the most likely to result in a win. Yunhe is also a group reflector; he makes sure the team is unified, consistent, and thorough with their solutions, and observes and comments on general team dynamic so that the team can achieve as much as possible.

Richard and Yuming are responsible for the aforementioned CheckMoves client. The CheckMoves client finds all possible moves on the chess board and appends each possible move onto a list of possible moves. If the next move is a valid move, the node for this valid move should be present in the list. This is to ensure that the moves that are being made are valid, and that the game can continue, i.e if there are no more valid

moves there is either a checkmate or a stalemate. This CheckMoves client also helps the AI client; the AI client similarly makes a tree of all possible future moves and act accordingly to make the best possible move. Richard and Yuming work with Yunhe as the reflectors of the team. By making sure the team is consistent with their solutions and monitoring the team dynamic, the trio helps keep every team member on the same page so that there is less confusion and no animosity, making the team efficient and the working environment healthy.

Xingjian is responsible for writing the Main client. The main client assures that the game experience runs smoothly, running all the functions appropriately as to simulate a proper chess game. Xingjian is also working with Yunhe on the GUI client, which is the user interface on which the game is to be played. As aforementioned, the GUI is an extension of the Board client. Together, they will develop the complete GUI client which includes the board, the pieces, movements of pieces, and other functionality of the user interface. Furthermore, Xingjian will be the team recorder; he will log all the discussion comments when the team is sharing ideas and will take note of the important concepts every member of the team has learned in the development of this game.

The task of any potential additional features has been given to Ameya, Richard, and Yuming. This trio has ideas for the development of a timer mechanism, which will set a time limit for each turn. This is to further authenticate the playing experience, and to simulate a real chess game. Further additional features, such as hints, has also been discussed - the hints feature will work similarly as the AI client; scouting all potential moves and recommending the best possible one.

Although the team has elected to split tasks, it is imperative to note that every single team member will be working together and assisting one another when need be. For instance, although Ameya is responsible for building the presentation, members of the team will assist him with their respective parts. Every team member will assist each other with development and debugging their code. Evidently, although this section partitions the responsibilities of the project between team members, all team members will work together and help each other in order to develop the project and any documentation.

4.2 Timeline of Development, Testing, Releases

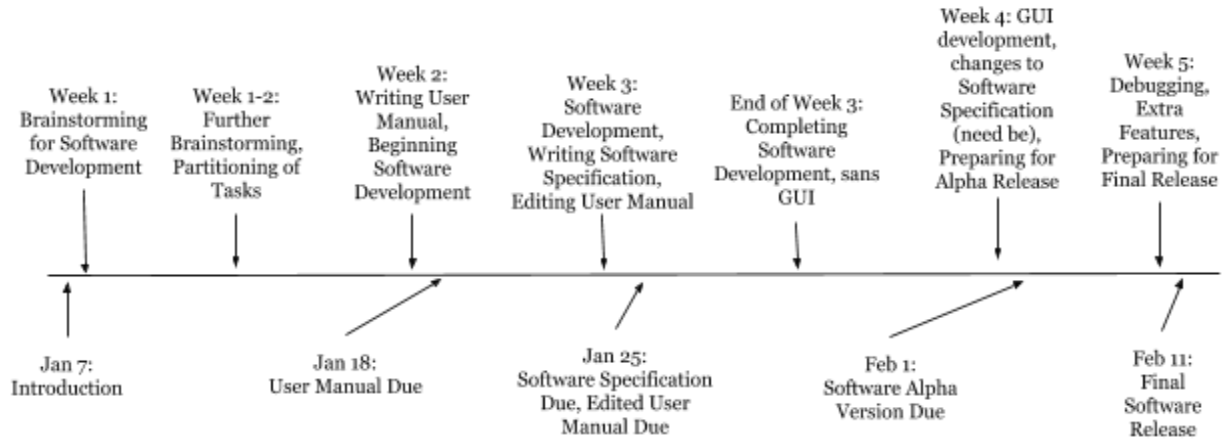

Figure 14: Timeline of our development

January 7: The team was introduced to the challenge, as well as introduced to the other team members. The team name was decided upon.
Week 1: The team began brainstorming for the development of the project. Game mechanics and gameflow were discussed.
Week 1-2: The team had further brainstorming for the development of the project. Ideas such as the utilization of certain data structures, how to build the AI, challenges with the GUI interface were discussed. After these ideas were noted, tasks according to these ideas were partitioned to members of the team.
Week 2: The team began to write the User Manual. The document covered usage scenario, features, installation, etc. Some software development had begun; team members began working on their respective bits of code. Team members gathered to discuss team roles.
January 18: The User Manual was due.
Week 3: Software Development continued; teammates worked on their respective clients and the project started to take shape. The User Manual needed editing, which was a task for the team. Another document, the Software Specification, was written. This document covered gameflow, the objective of functions in the software, development plan and timeline, amongst other things.
January 25: The Software Specification and the edited User Manual were due.
End of Week 3: Software Development was nearing the end; teammates were on the verge or had completed their individual clients, sans GUI.

Week 4: GUI development had begun, game mechanics were almost completed. Extra features were discussed. Objective was to have a completed product; few errors and details could be addressed later.
February 1: The first release, the Alpha release, was due.
Week 5: Any inconsistencies and issues that were in the Alpha release were addressed. This week was dedicated to debugging any issues that were present. Time permitting, extra features were developed. Objective was to have a fully functional product.
February 11: Final Software release was due.

Copyright

Copyright © 2019 DeepCoreDumped Team. All rights reserved.

References

(rules)
https://www.flyordie.com/games/help/chess/en/games_rules_chess.html
(minimax algorithm)
http://stanford.edu/~cpiech/cs221/apps/deepBlue.html

Index