Developer Manual

HeroChess Version 1.0

May 12, 2021

Paul John Lee, Irania Ruby Mazariegos,
Rachel Villamor, Keane Wong
EECS 22L, University of California Irvine, Irvine, CA 92697

Contents

Glossary

.c: file extension for a c programming source file
.h: file extension for a c header file
AI: Computer player which will play against humans in Human vs. AI mode
Array: a collection of data items
Bishop: A piece within the game that can only move diagonally across the board
Black: Classification for the player who will move second when beginning the game.
Board: A square 2D array with the size of 8x8 that consist of 64 pieces*
Capture: When a capturing chess piece is capable of moving into the space that is occupied by an opposing chess piece, the opposing chess piece can be removed from the game and the capturing piece moves into its previously held space. This is referred to as capture and is the primary means of advancing the game.
Castling: A special move in the game of chess where the king and rook are moved simultaneously. It is important to note that this move is only allowed if both pieces have not made a move and the king has two spaces available towards his left or right. This move may not be performed if the king is in check or if there is another piece blocking the areas where the king and rook would perform this move through. To perform the move, move the king two spaces in the desired direction(left or right) and swap the rook to the first square from the king's original position. If you start as white you could castle by moving the king from e1 to g1 and rook from h1 to f1.
Char: a character variable
Check: Designates that a chess piece is able to capture the king, forcing the targeted opponent to necessarily make a move to prevent the attack from happening. The resulting move must not leave the king open to any additional checks.
Checkmate: Designates that a check has been made, and the opponent is unable to make any moves that would protect the king from additional checks. This condition wins the game.
En passant: A special pawn capture that can only occur immediately after a pawn makes a move of two squares from its starting square, and it could have been captured by an enemy pawn had it advanced only one square. The opponent captures the just-moved pawn "as it passes" through the first square.
File: The 8 vertical columns on the board, labeled 'a' through 'h'
King: King can move in any direction in steps of one. It is important that the king stays out of check and uses its special moves with other pieces to win the game.
Knight: A knight can move in L shapes that are made up of two squares forward and one step to the side.
Main: the main function for the program
Move: A move history that indicates the source and destination xy coordinates of a piece(Chars)
Movelist: A Linked list which contains the replays of games
Pawn: Can only move forward on the board. Can choose to advance one or two spaces only on the first move. After the first move the pawn must move one space. The pawn can only capture diagonally.
Player: A player struct that contains player name and color

Promotion: When a pawn has moved to the opposite edge of the board and cannot advance any further, it may be turned into any other game piece excluding the king.
Queen: A piece within the game that can move in any direction on the board
Rank: The eight horizontal rows on the board, labeled 1 to 8.
Rook: A piece within the game that can only move vertically and horizontally across the board.
Rooking: A simultaneous move (the only one in chess) whereby a previously unmoved King moves 2 squares toward an unmoved Rook and the Rook is moved to the other side of the King
Struct: a collection of variables under a single name
White: A term used to classify the player that will make the first move

1 Software Architecture Overview

1.1 Main Data Types and Structures

Various data structures will be used to represent aspects of the game. Most prominently, a 2D struct array will be used to represent the game board, and because a chess board is of a fixed standard size, we can more easily hard code certain functions as needed. A 2d array is chosen because its fixed size is ideal to work with for a game with universally standard dimensions, and structs will be used to contain data.

Each piece itself will be represented as a struct containing character and integer types. Because each pieces in chess do not contain piece-specific internal data (i.e behave the same no matter their position and what has happened to the piece in the past turns), we will not need to modify the piece data as we go, allowing us to use simple character and integer codes to represent pieces, and we will also not need to differentiate between duplicate pieces either. Auxiliary data structures include a queue used to track captured pieces (for display or for strategic analysis), and a doubly linked list used to keep track of the elapsed moves. This allows us to keep data in order and to undo moves as necessary. Other important data forms include text file writing for use of saving information and game states.

1.2 Major Software components

1.2.1 Flow of control(Order of execution)

General Procedure:

1. Main calls game module
2. Game module reads data from the board
3. Game module checks if the move is valid
4. Game module executes or refuses move
5. Board is modified
6. Game checks for checks and checkmates
7. Next turn is executed in main

Turn execution:

1. Takes the user input for the piece they want to move(ex. E7)
2. Checks for the piece type
3. Create the list of legal moves of the selected piece(selected square)
4. Takes the user input for the destination
5. Checks for the validity
6. Execute. If the move is invalid, ask for a new input

1.2.2 Structs/Functions required

Main

Main function contains the methods that are required for the game execution related to standard user inputs and outputs. This will include a test function which contains multiple unit tests for each module and overall gameplay.

Table 1: Main

Type	Name	Purpose
int	mainmenu()	Prints the starting menu for players.
int	convertColumn(char a)	Takes an alphabet character and returns the column value for 2d array.
int	convertRow(char a)	Takes an alphabet character and returns the column value for 2d array.
void	undo(PIECE **board, MLIST *mylist)	(HvAI)Takes in the board and move list and goes back two turns.
void	gameTest()	Contains the system test of the overall gameplay
void	unitTest()	Contains unit tests for module move/movelist, board, piece, board.

Piece

Piece indicates the chess pieces placed on the board. It will contain the information about the piece type and color that is required for the game to execute player moves, check the win condition each turn and display the updated board.

Table 2: Struct Piece

Type	Name	Purpose
char	type	To identify the piece type
char	color	To identify the piece color

Table 3: Piece

Type	Name	Purpose
Piece*	NewPiece(char name)	To Create a new piece element for the board
char	getType	To return identifier for the selected piece
char	getColor	To return the color identifier for the selected piece

Move

Move is where all the moves that players make will be stored in the form of a string. It tells which piece made what move on the board for tracking and takeback purposes. User will be able to save the moves in the form of .txt file after the game. Struct move will

Table 4: Struct move

Type	Name	Purpose
PIECE*	piece	To identify piece moved
PIECE*	removedPiece	To identify the piece that was captured (NULL if nothing was captured)
char*	source	Source square of the piece
char*	destination	Destination square of the piece

Table 5: move

Type	Name	Purpose
MOVE*	NewMove(PIECE *piece, PIECE *removedPiece, char *source, char *destination)	To create a new move element to the move list
heightvoid	DeleteMove(MOVE *m)	Delete a move and deallocate the memory used
PIECE	GetRemovedPiece(MOVE *m)	Returns removed piece
char *	GetSource(Move *m)	Return the source of the existing move
char *	GetDestination(MOVE *m)	Return the destination of the existing moved
PIECE	GetPiece(MOVE *m)	Return player number
void	PrintMove(Move *m)	To print the moves
int	getColS(MOVE* m)	Return column source
int	getRowS(MOVE* m)	Return row source
int	getColD(MOVE *m)	Return column destination
int	getRowD(MOVE *m)	Return row destination

Movelist

Movelist is a helper module for move.c that works as an iterator of the list of moves. A doubly linked list will be used. It allows the game to add/delete and print moves as long as the game goes. Inserting/deleting a move from the list will only be allowed statically to prevent any disruptions. Movelist has the pointers to the first and last element of the list including the length of the list, and MoveListEntry functions as an iterator which can move to the next/previous element at a time.

Table 6: Struct Movelist

Type	Name	Purpose
int	length	Length of the list
MENTRY*	first	Pointer to the first element of the linked list
MENTRY*	last	Pointer to the last element of the linked list

Table 7: Struct MovelistEntry

Type	Name	Purpose
MLIST*	list	Pointer to the list
MENTRY*	next	Pointer to the next move
MENTRY*	prev	Pointer to the previous move
MOVE*	move	Move element to be stored

Table 8: Movelist

Return Type	Name	Purpose
MLIST*	NewMoveList()	Construct a new move list in a form of linked list
void	DeleteMoveList(MLIST *l)	Destructor
void	PrintMoveList(MLIST *l)	Print list
void	AppendMove(MLIST *l, PIECE *piece, PIECE *removedpiece, int colS, int rowS, int colD, int rowD)	Add a move after the current move
MOVE*	RemoveLastMove(MLIST *l)	Remove the last move from the list
MOVE*	RemoveFirstMove(MLIST *l)	Return the length of the list
int	GetLength	Return the length of the list

Tree

The tree class is a set of functions that represent a tree of variable branches length. A given node has children and a set of siblings linked to it like a singly linked list.

Table 9: Struct TreeNode

Type	Name	Purpose
MOVE*	potMove	Stands for potential move
TREENODE*	child	
TREENODE*	nextSibling	

Table 10: Tree

Type	Name	Purpose
TreeNode*	NewNode(MOVE *newMove)	Construct a new move list as linkedlist
TreeNode*	EmptyNode()	Destructor
TreeNode*	GetChild(TREENODE *node)	Add a move after the current move
TreeNode*	GetNext(TREENODE *node)	Add a move before the current move
int	isEmptyNode()	Return a 1 if the node is empty
void	DeleteNode(TREENODE *nodeTBD)	Deletes the node, freeing it
void	DeleteNodeRecursive(TREENODE *nodeTBD) and its siblings recursively	Deletes all children of the node and its siblings recursively
void	SetChild(TREENODE *parent, TREENODE *newChild)	Sets child of the node to the input
void	SetNext	Sets sibling of the node to the input

Game

Table 11: Game

Type	Name	Purpose
int	isLegal(Piece** myBoard, int colSource, int rowSource, int colDestination int rowDestination, char curTurnColor)	If there is no piece at the source, it automatically fails If the selected piece is not the curTurnColor, it also fails If the Destination is out of bounds it automatically fails.
int	isLegalPawn(Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor)	Checks if it is at initial position (either row[1] for white or row [6] for black is in the (array) and if it is, the legal moves also includes a space 2 squares forward. This piece CANNOT capture whats in front of it. Also, checks for special moves that involve the pawn like EN PASSANT.
int	isLegalRook(Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor)	Moves consist of all spaces horizontal or vertical to it. Cannot move past pieces once it has captured a piece. Allows for castling if the king and itself both have yet to move.
int	isLegalBishop(Piece** myBoard, int colSource int rowSource, int colDestination, int rowDestination, char curTurnColor)	It's moves consist of diagonal movement across the board. can move as many squares as possible if there is no piece obstructing but must stop once it has captured.
int	isLegalQueen(Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor)	Can move in any direction but must stop once it has captured or if it is obstructed

Table 12: Game

int	isLegalKing(Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor)	Can move in any direction but only one square at a time. The exception is when it is performing castling.
int	isLegalKnight(Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor)	Checks only if the destination is empty or has an enemy 2 squares straight, then 1 square perpendicular
int	makeMove(Piece** myBoard, int colSource, int rowSource, colDestination, int rowDestination, char TurnColor)	Uses isLegal function from TABLE V to determain if the move is okay and then uses movePiece function to execute
int	isCheckmate(Piece** myBoard, char TurnColor)	Checks for checkmates for the curTurnColor. searches board for king of that color and then determines All possible checks. Returns 0 if there is no checkmates if there is at least one check but no checkmate it returns 1. If there is a checkmate it returns 2 and the game ends
int	isChecked(Piece** myBoard, char TurnColor)	Searches for king of color TurnCOlor. Checks for checks around that king. If there is none it returns 0. If there is at least one check then it runs (and immediately returns) isCheckmate. Run this function twice every turn; Once during 'islegal' function to determine if the C(which makes the move invalid) and once at the end of each turn to determine if the enemy is in check

Board

Board is the module that stores the 8x8 board where players will play the game of Chess. It is a 2D-array of struct Piece as they are the main elements of the game. Board module will enable the player moves to be an outcome on the board such as the initial setup, moving, taking pieces. The game will be able to obtain the information of a designated coordinate (whether it is occupied, type, color if occupied) that is required

by the program to properly run the game.

Table 13: Board

Type	Name	Purpose
Piece**	makeBoard()	initializes the game board in the form of a 2D array.
void	deleteBoard(Piece** myBoard)	Deletes the board, freeing the memory and deallocating the array.
Piece	getPiece(Piece** myBoard, int col, int row)	Returns the piece at position colrow
int	placePiece(Piece** myBoard, int col, int row, Piece q)	Puts the piece specified at colrow, returning 0 if successful else 1
Piece	removePiece(Piece** myBoard, int col, int row)	Removes piece from specified colrow, returning the removed piece and filling the spot with null. Returns null if there is nothing there.
int	movePiece(Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination)	Calls removePiece on the piece at Source and stores it. It also calls removePiece on the piece at Destination, then calls placePiece at destination. if the piece was removed then it returns the removed piece. The returned piece must be saved and then freed by the program during lifetime
void	PrintBoard(Piece** myBoard)	Prints board to the console.
void	initalizeBoard()	set up the initial condition to the board.

AI
Module supports the Human vs. AI gamemode. See **Table 14**.

Replay
This module relates to printing the game replay text file. See **Table 15**.

1.3 Module Interfaces

The primary functions of each module file should not require interaction with any private functions and perform their task concisely and without additional undocumented changes.

The piece files should have methods to change and get their respective parameters, and should only be changed as necessary or accessed as needed for settings and game manipulation. Similarly the board should have strict accessor and mutator functions, and should not require knowledge of the workings of the board as a data structure. These will be primarily accessed and modified by the game function, which only checks for move legality and returns appropriate messages based on errors, successes, and win conditions. The game will otherwise rely on the main function for inputs.

Otherwise, the settings files will be separated from the other files and primarily only contain mutator functions for user side settings and similarly the movelist function will not access the game nor board functions, but instead records data as the user inputs it, and is driven by the main function. This removes the modules from the rest of the program but retains its usual functionality.

Table 14: AI

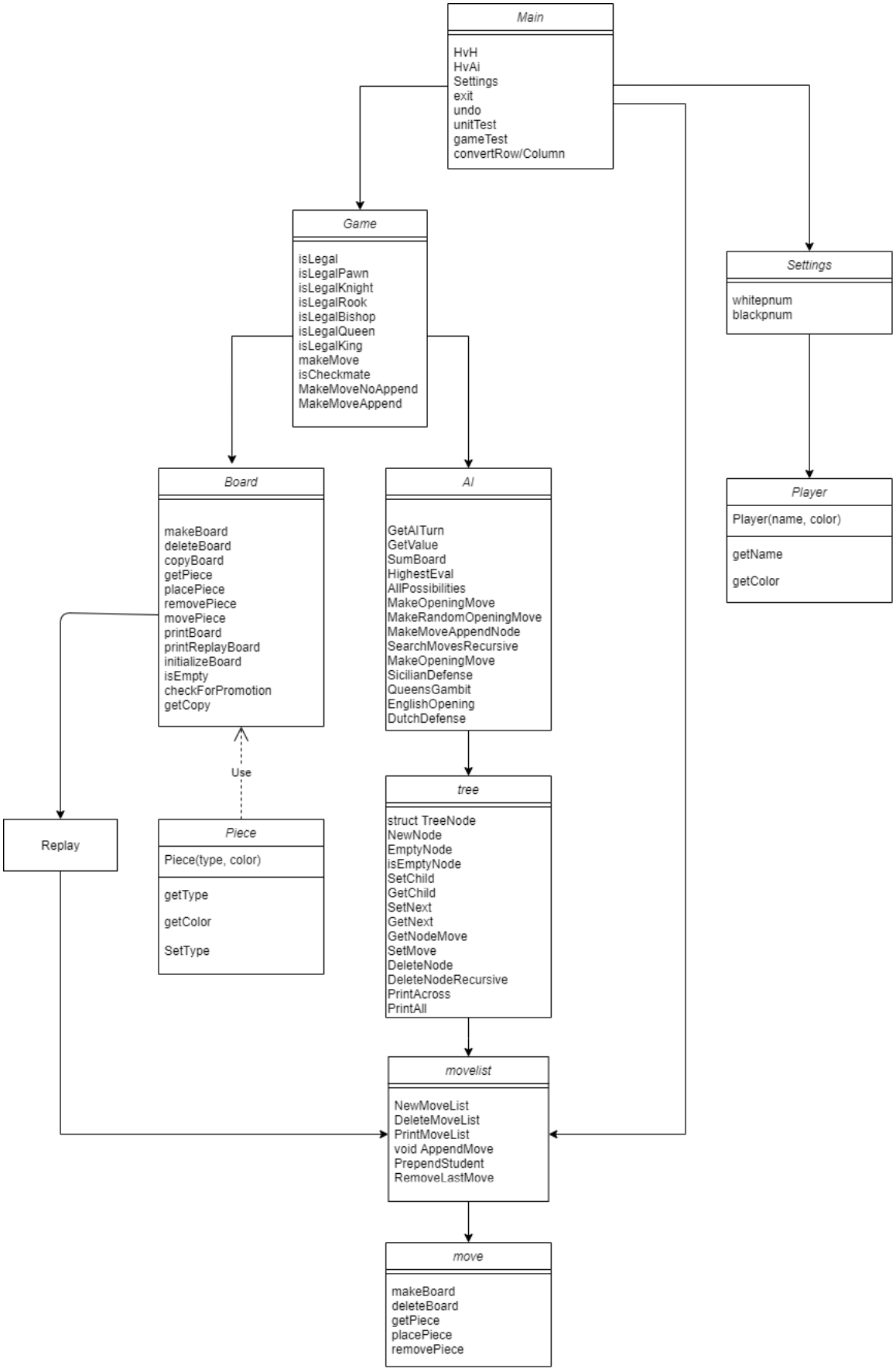
Type	Name	Purpose
int	GetAITurn(PIECE **myBoard, char computerColor, MLIST *myList,int Depth)	make move on the AI's behalf
int	GetValue(PIECE *myPiece, char curTurnColor)	returns value based on type of piece
int	SumBoard(PIECE **myBoard, char curTurnColor)	adds pieces' values to find worth of board
MOVE*	HighestEval(PIECE **myBoard, char curTurnColor, MLIST *myList)	evaluates board and determines best move
void	AllPossibilities(PIECE **myBoard, char curTurnColor, MLIST *myList, TREENODE *root)	stores every possible move as children to root
int	MakeOpeningMove(PIECE **myBoard, char curTurnColor, MLIST *myList)	makes a random hardcoded opening move
int	MakeRandomOpeningMove(PIECE **myBoard, char curTurnColor, MLIST *myList)	makes a random hardcoded opening move
int	MakeMoveAppendNode(PIECE **myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor,MLIST *myList, TREENODE *blankNode)	make a move without appending it to myList
int	SearchMovesRecursive(PIECE **myBoard, char computerColor, MLIST *myList,TREENODE *myNode, MOVE *bestMove, int Depth, int originalDepth)	recursively look through levels of moves
int	SicilianDefense(PIECE **myBoard, char curTurnColor, MLIST *myList)	sicilian defense opening move
int	QueensGambit(PIECE **myBoard, char curTurnColor, MLIST *myList)	queen's gambit opening move
int	EnglishOpening(PIECE **myBoard, char curTurnColor, MLIST *myList)	nglish opening, opening move
int	DutchDefense(PIECE **myBoard, char curTurnColor, MLIST *myList)	dutch defense opening move

Table 15: Replay

Type	Name	Purpose
void	replay(MLIST *l)	generate game replay file
void	printFormat(FILE *fptr, char *fname)	works on file directly

1.3.1 Hierarchical Diagram

FIG. 2: V1.0 Hierarchical Diagram



1.4 Overall Program Control Flow

The HeroChess program will operate primarily from a command line driven by a main function to take user input and set initial parameters. This main function will control the primary operation of the game on the user end and also manage functions such as saving and replay. This is the primary function that handles turn to turn interactions, and makes use of the function outputs.

The board will be represented by a 2d array of pieces, which by themselves do not interact strongly with one another and contain only data relevant to the game rules. The board files do not check for move legality or errors except for out of bound errors. The software contained in the game.c and game.h files will primarily drive the game logic and do most of the runtime error handling and handle move legality. The game files and functions are written to handle most checks and rules automatically, and check win conditions as accurately as possible.

More separated from this hierarchy, but regularly interacting with it, are the movelist and settings files. These functions modify or read in data from the game or board without being directly tied to it or forming any of its structure. The setting files will change mostly cosmetic aspects without affecting the game itself very much.

The movelist data, represented as a linked list of structs, will be regularly called by the program to read in data and save it temporarily as the game progresses. These functions should NOT modify any game data. At the end, they can be accessed by the main function and stored into a readable plaintext file for replay later, and can alternatively be used to start the game and board from a set position.

2 Installation

2.1 System Requirements, Compatibility

In using any version of HeroChess users will want to access the EECS Linux Servers. Users are recommended to have the Linux version CentOS 6.10 and perhaps even Xming if the graphical user interface is included.

2.2 Setup and configuration

1. Setup

Download a SSH client with terminal support.

1. Create a directory where the game will run as needed
    - From the command line, run the command mkdir <directory path>
    - Linux example: /EECS22L
2. Get the latest version
    - From the command line, run the command git pull

3. Extract the file
    - From the command line, run the command

Use "UCI student ID"@crystalcove.eecs.uci.edu or team""@crystalcove.eecs.uci.edu

Other servers: bondi.eecs.uci.edu, laguna.eecs.uci.edu, zuma.eecs.uci.edu

All 4 servers will work fine to run the game.

Set the port to 22 and connection type to SSH.

2. Configuration

Configuration files and their purpose:
ai.c - Contains functions for the automated player gameplay
ai.h - Contains declarations of functions in ai.c
board.c - Contains definitions of structs and functions pertaining to the board
board.h - Contains declarations of structs and functions in board.c
game.c - Contains functions for game algorithms and win conditions
game.h - Contains declarations of functions in game.c
main.c - Contains main game function and print menu function
move.c - Contains definitions of structs and functions relating to moves
move.h - Contains declarations of structs and functions in move.c
movelist.c - Contains definitions of functions related to the list of moves
movelist.h -Contains declarations of structs and functions in movelist.c
piece.c - Contains definitions of structs and functions related to the chess pieces
piece.h - Contains declarations of structs and functions in piece.c
replay.c - Contains definitions of functions related to printing the game replay text file
replay.h - Contains declarations of functions in replay.c
settings.c - Contains definitions of functions related to player and game handling
settings.h - Contains declarations of functions in settings.c
tree.c - Contains definitions of structs and functions pertaining to tree implementation
tree.h - Contains declarations of struct and functions in tree.c
makefile - Required. Contains the compilation command along with shared libraries.

2.3 Building, Compilation, Installation

On the terminal type tar -xvzf `Chess_V1.0_src`.tar.gz and press enter. Then, type cd `Chess_V1.0_src` and press enter. Finally, type make and press enter. Installation is now complete.
To run the game, type ./bin/HeroChess, press enter, and enjoy the game.
Users will be able to uninstall the Chess game executable files by being in the directory called `Chess_V1.0_src` and using the "make clean" command.

3 Documentation of Packages, Module, Interfaces

3.1 Detailed Description of Data Structures

The majority of the important parts of the program are represented as basic structs containing basic data that pertains to the game rules.

The base piece structs are abstracted to represent only a piece on the board, and are initialized to contain the piece name and color. The rules for movement that pertain to each piece are hardcoded in the game files, and handled there independently to the pieces themselves. This data is meant to be initialized then accessed only, except in special cases.

These structs are then contained in a dynamic 8x8 array of pieces at a fixed size to represent a board. The board is regularly modified and read to progress the game and contains purely positional data, and does not contain internal data to things such as turns elapsed, replays, or save states. This is the primary data structure to present the game and does not change, except in contents and location of contents.

Auxiliary data structures include a double linked list with a head and tail end, and contains data for elapsed moves that will then be applied to a set initial board, and saved to an external text file as appropriate. Each move in the movelist is contained in a move entry and can be accessed through the functions detailed in 1.2, and moves can be appended and deleted as necessary without the relative positioning of other moves being changed.

AI determines its move using a minimax algorithm, where it calculates the next move based on the net board weight. Upon calculating every possible board weight, tree is used to store all the possibilities. Initial move for each piece is stored in the pointer called root, and all other subsequent moves are stored as its child, and the children nodes will have their own children. This recursive computation stops when AI finds the best move possible.

3.2 Detailed Description of Functions and Parameters

Main

mainmenu: Prints the starting menu for players. This will take user input out of 4 different integer options and return the option number players choose.
convertColumn: Takes an alphabet character and returns the column value for 2d array.
convertRow: Takes an alphabet character within 1 to 8, and return the row value for the 2d array. The range is 0 to 7
undo: Takes in the board and move list and set back the board by two turns. Only used for Human vs AI mode. This only activates whenever movelist have more than 2 elements.
gameTest: Contains the system test of the overall gameplay.
unitTest: Contains unit tests for module move/movelist, board, piece, board.

Piece

NewPiece: This function creates a new piece element for the board. The parameter is char name.
getType: This function returns the identifier for the selected piece.
getColor: This function returns the color identifier for the selected piece.

Move

NewMove: This function creates a new move element to the move list. The parameters are char *move and PIECE *piece.
DeleteMove: Delete a move and deallocate the memory used.
GetRemovedPiece: Returns removed piece.
GetSource: Return the source of the existing move.
GetDestination: Return the destination of the existing move.
GetPiece: Return player number.
PrintMove: To print the moves.
getColS: Return column source.
getRowS: Return row source.
getColD: Return column destination.
getRowD: Return row destination.

Movelist

NewMoveList: This function constructs a new moves list in the form of a linked list.
DeleteMoveList: This function deconstructs the moves list that was created.
PrintMoveList: Prints list.
AppendMove: This function adds a move to the list after the current move in the moves list.
RemoveFirstMove: This function removes the last move from the list and returns it.
RemoveLastMove: This function removes the first move from the list and returns it.
GetLength: This function returns the length of the moves list. The parameter is MoveList*.

Tree

NewNode: Construct a new move list in a form of linkedlist
EmptyNode: Destructor
GetChild: Add a move after the current move
GetNext: Add a move before the current move
Add a move before the current move: Return a 1 if the node is empty, i.e doesn't have a move
DeleteNode: Deletes the node, freeing it
DeleteNodeRecursive: Deletes all children of the node and its siblings recursively until it is done
SetChild: Sets child of the node to the input
SetNext: Sets sibling of the node to the input

Game

isLegal: This function checks the color of the piece at the source and the color of the piece at the destination. The parameters are Piece** myBoard, int colSouce, int rowSource, int colDestination, int rowDestination, and char curTurnColor. If there is no piece at the source, the selected piece is not the current turn color, or the destination is out of bounds, the function fails. It will also fail if the selected piece and the piece at the destination are the same color. The function will proceed if the selected piece and the destination are the same color if the selected piece is a king or a rook and the destination is a rook or a king. If the destination is empty or has an opposing piece, the function checks what type of piece is at the source and calls the private isLegal function that corresponds to it. If the destination is legal, it then checks that the move does not place the friendly king in check.
isLegalPawn: This function checks if the pawn is at its initial position (row [1] for white and row [6] for black) and, if it is, the legal moves also include a space 2 squares forward. The pawn cannot capture anything that is directly in front of it. This function also checks for en passant, If the pawn moves forward and ends its turn on the opposite side of the board, a message will be printed to the console, taking user input. The user will be asked if they want to promote their pawn to bishop, knight, rook, or queen. The parameters are Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor.
isLegalKnight: This function only checks if the destination is empty or has an opposing piece 2 squares forward then 1 square perpendicular. The parameters are Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor.
isLegalRook: This function checks if the destination of the piece is horizontal or vertical to its current position. The piece cannot move past pieces once it has captured a piece. The function also allows for castling if there is a king at the destination and neither the king or rook have moved. The parameters are Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor.
isLegalBishop: This function checks if the destination of the piece is diagonal (right and left) to its current position. The piece cannot change directions once it has moved in a certain direction. It also cannot move past pieces once it has captured a piece. The parameters are Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor.

isLegalQueen: This function is a combination of isLegalRook and isLegalBishop. It will check if the destination of the piece is horizontal, vertical, diagonal (right and left) to its current position. Similarly, it cannot move past pieces it has captured. The parameters are Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor.
isLegalKing: This function checks if the destination of the piece is within one space, in any direction, of its current position. It allows for castling only if the rook and king have not moved. The parameters are Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor.
makeMove: This function uses the isLegal functions described above to determine if the move entered is legal. It then uses the movePiece function to execute the move. The parameters are Piece** myBoard, int colSource, int rowSource, int colDestination, int rowDestination, char curTurnColor.
isChecked: This function searches for the king of the current color's turn and checks for checks around it. If there are no checks, it returns 0. If there is at least one check, the function runs and immediately returns the isCheckmate function. This function will run twice every turn - once during an isLegal function to determine if the move entered accidentally put the friendly king in check. The second time at the end of each turn to determine if the opposing king is in check. The parameters are Piece **myBoard and char TurnColor.
isCheckmate: This function checks for checkmates for the current turn color. It will return 0 if there is no checkmate and 1 if there is a checkmate. If the return is 1, a message will also be displayed to the console alerting the user of the checkmate. The parameters are Piece **myBoard and char curTurnColor.

Board

makeBoard: This function initializes the board in the form of a 2d array and fills the spaces with null.
printBoard: This function prints the board to the console with each piece arranged in their starting positions. It will also print the rank on the left-hand side of the board and the file at the bottom of the board. The parameter is Piece** myBoard which will print the initialized board and update it as the user makes moves.
deleteBoard: This function deletes the board, frees the memory, and deallocates the 2d array. The parameter is Piece** myBoard.
initializeBoard: This function will set up the initial conditions of the board.
getPiece: This function returns the piece with its position on the board. The parameters are Piece** myBoard, int col, and int row. This allows a unique return type for each piece. The return type will then be used in functions such as placePiece which needs the previous position of the piece in order to move it.
placePiece: This function moves the selected piece to the new destination that the user has entered. The parameters are Piece** myBoard, int col, int row, and Piece q. It will return 0 if successful or 1 if it is not successful. If not successful, an error message will be displayed to the console.
removePiece: This function removes a piece from a specific column and row, returning the removed piece and filling the space with null. If there is no piece to remove, the function will return null. The parameters are Piece** myBoard, int col, and int row.

movePiece: This function calls the removePiece function on the piece at Source and stores it. It also calls the removePiece function on the piece at destination, then calls the placePiece function at destination. The parameters are Piece** myBoard, int colSource, int rowSource, int colDestination, and int rowDestination.

AI

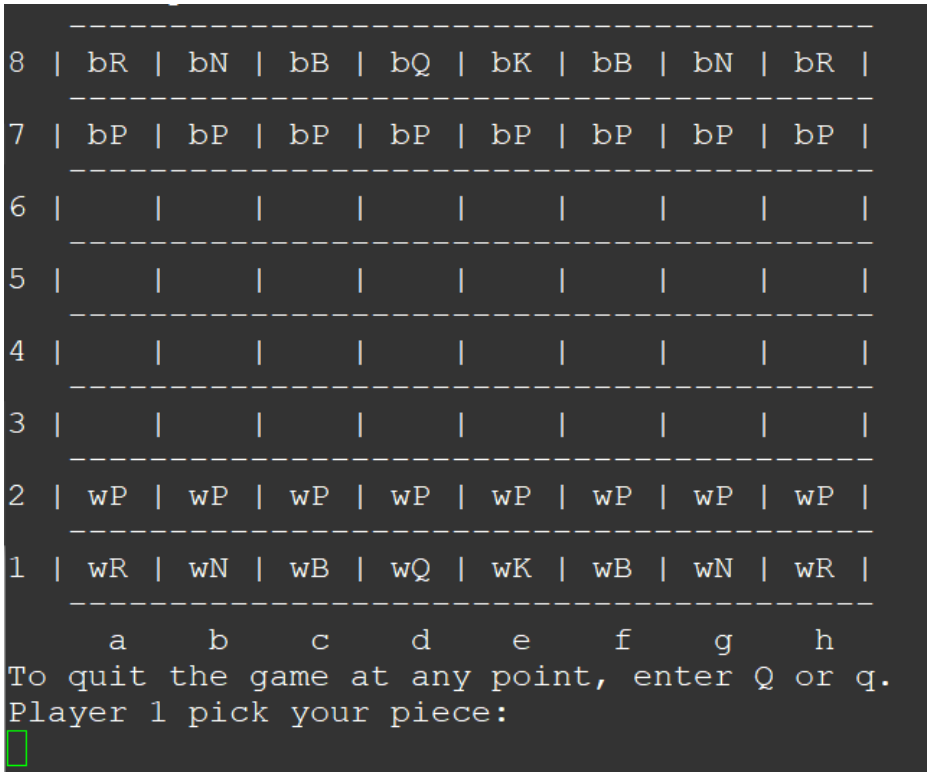
GetAITurn: A function to makemove on the AI's behalf based on what its turn is. Recursively searches to a depth of moves equal to Depth.
GetValue: returns value based on type of inputted piece; positive values for friendly pieces and negative values for enemy pieces
SumBoard: determines worth of board by adding all the pieces' values; analyzes risks and benefits
HighestEval: evaluates board and determines what move would leave it in the best position
AllPossibilities: gets every single possible move on the board and stores them as children to root, all connected to one another by pointer links
MakeOpeningMove: makes a random hardcoded opening move from a small library
MakeRandomOpeningMove: makes a random hardcoded opening move from a small library
MakeMoveAppendNode: make a move without appending it to myList, instead adds the move to an empty node or blankNode; similar to MakeMoveNoAppend
SearchMovesRecursive: a helper function to recursively look through the various levels of moves
SicilianDefense: sicilian defense opening move
QueensGambit: queen's gambit opening move
EnglishOpening: english opening, opening move
DutchDefense: dutch defense opening move

Replay

replay: generate game replay file
printFormat: works on file directly

3.3 Detailed Description of input and output formats

FIG. 2: Blank Board for Human v. Human (HvH)



During each turn, the board is updated and printed to the screen along with prompts asking the next user to enter their move. To enter a valid move, the user must enter two positions on the same line: the current position of the piece they want to move and the position they want the piece to move to. These positions have to be entered in a [lowercase letter][number] format, i.e. a2.
The two prompts that appear to the user for a move are the following:

1. The first will say ”Player 1/2 pick your piece:”.
2. The first will say ”Where would you like to move this piece?”.

Once a move is entered and before the updated board is printed, the computer checks if the piece is allowed to make the entered move according to the rules of chess. If the piece is not allowed to follow the player's entered move, then this situation would be considered an illegal move and therefore would not be allowed onto the updated board. Cases wherein a player enters an illegal move can also be described as moves that are outside the board's bounds and/or not typed in the [letter][number] format. If any of these conditions are met during a player's turn, an error message would appear. The player who entered the invalid entry will still be on their turn after the error message and will once again be prompted to enter a valid move.
In cases where either players want to quit the game, they can enter 'Q' or 'q' for any of the prompts to quit the game at any time. A reminder for this is printed under every game board.

4 Development plan and timeline

4.1 Partitioning of tasks

Timeline Overview

1. Planning
2. Development
3. Prototyping
4. Testing
5. Pre-launch
   (a) Alpha Release
   (b) Beta Release
6. Launch
   (a) Master Release
7. Competition

1. **Planning** - Week 1
   Whole team discussion on general overview of plans for the Chess Game
2. **Development** - Week 2
   Whole team collaboration on Application Specification/User Manual
3. **Development Prototyping** - Week 3
   Application Specification/User Manual Finalizations
   - 3.2, 3.3 Human v. Human, Human v.
   - AI Games - Rachel Villamor
   - 3.5 Implementing the Official Rules of Chess - Keane Wong
   - 3.7 Advanced Moves - Irania Mazariegos
   - 3.9 Moves Log - Mario Tafoya
   - 3.10 Tournament Support - Paul Lee
   - LaTeX Conversion - Mario Tafoya

   Software Implementation
   - Discussion - whole team
   - Started code - Paul Lee
4. **Development Prototyping** - Week 4
   Software Architecture Specification
   - 1.1, 1.2 (game, board, AI and piece functions),1.3, 1.4,- Keane Wong
   - 3.2 Functions and parameters - Irania Mazariegos

- 3.3 input and output formats, 4 development plan/timeline - Rachel Villamor
- LaTeX Conversion - Mario Tafoya

Software Implementation

- Work on module files - whole team will collaborate to edit and error-check each file

5. **Testing Pre-launch** - Week 5
   Software Implementation
   - Work on module files - whole team will collaborate to edit and error-check each file
   - Compile and run all .c and header files
   - Release Alpha Version
6. **Pre-launch** - Week 6
   Software Implementation
   - Work on module files - whole team will collaborate to edit and error-check each file
   - Tree Implementation Game Replays - Keane Wong Paul Lee
   - GUI research for Project 2 - Irania Mazariegos Rachel Villamor
   - Compile and run all .c and header files
   - Release Beta Version
7. **Launch** - Week 7
   (a) Debugging - whole team
   (b) **Master Release**
8. **Competition** - Week 8

4.2 Team member responsibilities

- **Member and roles**
  - Manager: Paul Lee
  - Presenter: Keane Wong
  - Recorder: Rachel Villamor
  - Recorder: Mario Tafoya
  - Reflector: Irania Mazariegos

Copyright

Error Messages

"Error. Try again":

- Illegal Move: Attempted to move a piece to a space where it cannot occupy or cannot move to normally.
- Incorrect input: Inputted something that did not register as a valid option or coordinate
- Not your Piece: Attempted to move a piece that did not belong to the player of the current turn.
- Out of bounds: Attempted to move a piece to a space outside the gameboard

"Error: Out of bounds column option":

User input in the form of [letter][number] does not correspond to a column labeled in the board, a-g.

"Out of memory! Aborting...":

- Cannot create new piece because of no memory
- Cannot create new move entry for list
- Cannot create new list
- Cannot create new node
- Cannot create empty node

Index

References

[1] "Play Chess Online vs Cpu Play Chess Against Computer Expert-Chess-Strategies.com - Hr.prodaja2021.com." n.d., hr.prodaja2021.com/content?c=play ches online vs cpuid=2. Accessed 27 April 2021.