|  |
| --- |
| CS246 |
| Design Document |
| For Chess Project |
|  |
| **Eugene Zhang 20415459 & Jackson Lo 20469499** |
| **12/1/2013** |

|  |
| --- |
|  |

Contents

[Feeding Inputs 2](#_Toc373704563)

[Setup Mode 2](#_Toc373704564)

[Starting a Game 3](#_Toc373704565)

[Moving a Piece 3](#_Toc373704566)

[Movement Algorithm 3](#_Toc373704567)

[Resign 4](#_Toc373704568)

[Castling, Pawn Promotion, En Passant 5](#_Toc373704569)

[Assigning Scores 5](#_Toc373704570)

[Checking and Checkmate 5](#_Toc373704571)

[Stalemate Condition 6](#_Toc373704572)

[Computer AI 6](#_Toc373704573)

[Graphical Mode 8](#_Toc373704574)

[QUESTION 1 FROM PLAN OF ATTACK 8](#_Toc373704575)

[QUESTION 2 FROM POA 9](#_Toc373704576)

[QUESTION 3 FROM POA 9](#_Toc373704577)

[Design Document Question 1 10](#_Toc373704578)

[Design Document Question 2 10](#_Toc373704579)

# Feeding Inputs

A while loop with getline and tokens (A token is a piece of the line of input delimited by whitespace) is performed for reading in the input. Initially, the only two valid commands are game human/computer[1-4] human/computer[1-4] and setup.

If game is called, creates a new player object on the heap for human/computer respectively with their respective constructors. Then, the valid commands are move \_ \_ and resign. If move \_ \_ is called, the Grid::movePIece command is called if the current player is human, else the Computer::generateMove() is called. If it was a successful move, the turn is alternated and Grid::isInCheck(char) is called to see if the player is in check and displays the according message. If they are in check, Grid::isInCheckmate(char) is called to see if the player is in checkmate and displays the according message. Also, Grid::isStalemate(char) is called to check if the game is in stalemate, and if it is, the according message is displayed. After this, the loop reiterates until checkmate, stalemate or resign occurs.

If setup is called, the valid commands are + \_ \_ , - \_ , = colour or done. Again with a while getline loop and tokens, the Grid::addPIece(char c, std::string coord) function is called to add a piece for + \_ \_ and Grid::removePIece(std::string coord) function is called to remove a piece for - \_. For = colour, Grid::setTurn(char c) is called to set the turn to whatever c is. Finally, if done is called, Grid::validSetup() is called to check if the board is indeed in a valid state as per the requirements and exits if it is.

# Setup Mode

Setup mode is performed when the setup command is called. Then the valid commands of adding a piece, removing a piece, setting turn and exiting(done) can be executed.

For adding a piece, the Grid::addPiece(char, std::string) function is called, which takes in the piece (K for white king, k for black king, R for white rook etc…) and the coordinate as a string (e1, e5, a1, etc..). Then, addPiece checks if the location is null, if it isn’t, it deletes whatever is there and sets it to null. Then it proceeds to set the location to the according Piece type.

For removing a piece, Grid::removePiece(std::string coordinate) is called. This checks if the location is null and if it is not, then it deletes the piece there and sets it to null.

For setting a turn, Grid::setTurn(char) is called, this takes in a char for ‘w’ or ‘b’: white or black respectively. Then it sets the private field in the Grid class to ‘w’ or ‘b’.

The done command executes Grid::validSetup() which in turn checks if each player has exactly 1 king. Then it checks if there are any pawns on the ends/edge of the board. Finally, it checks if either of the kings are in check. Depending which condition was false, a char is returned and in the main, the char is interpreted resulting in the according error message. (E.g. return ‘c’ and in the main if the return value is ‘c’ then the message: Please fix invalid setup. King cannot be in check.). If all the conditions are met, then the setup will exit.

# Starting a Game

Type game human/computer[1-4] human/computer[1-4], which will create the human or computer players on the heap with their respective constructors. Then, the Grid::startGame() is executed.

In Grid::startGame(), it checks the private field setup to find out if the game has been setup through setup mode. If it has been, then nothing is done. If it hasn’t, then the game is setup with the black and white pieces on the grid/board accordingly in the standard starting positions. And the private field in grid, isTurn is set to white’s turn.

# Moving a Piece

We move a piece when we take receive, from the standard input, “move”. We then check if this turn’s player is either a human or a computer. If it is a computer it will use the computer AI to generate a legal move. If it is a human, then it will need two more inputs which are where you want to move a piece from and to. If there is no piece in the coordination of ‘from’, or if the player is moving his opponent’s piece or if that move is illegal then it will not move any piece and wait until the current player makes a legal move.

# Movement Algorithm

Each piece has its own isLegal function and this is inherited from the virtual function of Piece class. It is virtual because in the grid we can create an array of all the pieces whether they are a knight or a queen or anything else, thus when we call isLegal for any particular piece it will call its own isLegal function accordingly.

The general algorithm first checks if the destination of movement is two characters long (such as ‘a1’ ‘b4’). Then it check to see if the destination is legal, making sure the first character is only between ‘a’ and ‘h’ and second ‘1’ to ‘8’. Then it checks if it is capturing its own pieces, if it is then this function will return false and the move will not be legal. If not, then it will continue until it satisfies an illegal condition and return false or finish running the function and return true.

For each different subclass of piece, it has its own conditions it must satisfy besides the general algorithm to make the movement true.

First off, the rook can only move in a straight line. To implement this, we make sure either only row number changes or column number changes, but not both. If anything else, then return false. For horizontal movement and vertical movement we check to see if there is any piece between the rook and his destination, if there is then return false. If no illegal moves are made, then finally return true.

The bishop is similar to the rook. It also checks if there is no pieces between the bishop and his destination. How we make sure that the movement is diagonal is by taking the positive difference between row number and compare then with the positive difference of the column number. If the change of row and column number is the same that means it is a diagonal movement.

The queen basically combines the rook’s algorithm with the bishop’s algorithm.

The knight’s algorithm checks if the change in the column number is 1 then the change in row number has to be 2. If the change in column number is 2 then the change in row number has to be 1.

The pawn can only move 1 direction. So based on its own colour, we check if the pawn is moving either 1 or 2 step vertically. If it is only moving 1, we will check if it is either capturing a piece or not. If not, we check if it is going vertically straight. If it is capturing, then the difference of column number must differ by 1. If is moving 2 steps. Then it must be going vertically and also it must be in the pawn’s initial starting positions. Like the rook, we check if there is a piece in between blocking the way and also we check that the destination of the pawn is an empty space since this move can be only used for moving not capturing.

Finally, the king’s algorithm first check the king is only moving one space. We also check if the king is not moving into check. Since the other player can also protect the checking piece using another piece, we first remove checking piece that is in the destination of king’s path and then we check if any other piece of the other player can move to that space. If it can, then return false and add that piece back in. Since pawns move different from how it captures. We check the spaces beside the desired destination of the king where pawn is able to capture a piece in that space if there is a place there. If it there is a pawn there, then return false. We also need to check if we can’t move near another king. So we check if a king is near that space we want our king to go to. If there is a king like that, then return false.

# Resign

The game is ended and the winning player’s score is increased by 1.

# Castling, Pawn Promotion, En Passant

Castling works when king::isLegal fails, since it is a deviation from the standard movement pattern. But it gets caught by a separate if statement in the Grid::movePIece which checks for a castling movement pattern and the requirements for a castling move. In specific, within that if statement, we check the current position of the king and if there is an existing rook in the correct castling position. Then we check if either of these pieces have moved before and if there are any pieces in between. If both of those are false, then the castling move is performed accordingly.

Pawn promotion works when a pawn reaches the edge/end of the board. This is checked after every successful move. If the pawn is at the end of the board then the piece specified for promotion will be the new piece in place of the pawn. In the case that no piece is specified, the pawn automatically is promoted to a queen.

Pawn en passant works when pawn::isLegal fails, since it is a deviation from the standard movement pattern. But it gets caught by a separate if statement in the Grid::movePiece which checks for pawns. Within that if statement, we check (within the same row) the left and right of that pawn we are moving. If there is a piece on either side of our pawn and it just moved 2 spaces, we check to see if the movement destination matches where the en passant movement should be and if it is we can proceed with deleting the captured piece and moving our pawn accordingly.

# Assigning Scores

In the main, the score is kept track of with 2 double type variables, one for black and one for white. If a resign or a checkmate occurs, the winning player’s score variable increases by 1. If a stalemate occurs, both player’s score increases by 0.5.

When control-D is pressed (cin.eof() = true), then the main cin loop is exited and the score is printed before termination.

# Checking and Checkmate

The Grid::isInCheck(char c) function is called to check if white or black is in check. It checks if white, if c == ‘w’ or c == ‘b’ for black.

The function first locates the king and saves the coordinate. Then the board is traversed and for every enemy piece, it is checked if it has a legal move to the kings location/coordinate. If it does, then the king is in check and true is returned. Else if the traversion ends, return false.

For checkmate, Grid::isInCheckmate(char c) function is called to check if white or black is in checkmate. It checks if white, if c == ‘w’ or c == ‘b’ for black.

It is presumed that Grid::isInCheck has been called before and returned true before Grid::isInCheckmate is called. In Grid::isInCheckmate, it is checked if the king piece has legal moves, by called Piece::hasLegal. If it has legal moves, then false is returned, if it has no more legal moves, then true is returned since it is in checkmate.

# Stalemate Condition

The Grid::isStalemate() is called after each move to determine if the game is in stalemate. In Grid::isStalemate(), the entire grid/board is traversed and on each piece, Piece::hasLegal is called. If a single piece for the current turn’s colour has a legal move (hasLegal return true), then isStalemate() returns false. Else if all the pieces of the current turn’s colour return false for hasLegal, then isStalemate returns true. Since none of the pieces have a valid move to execute.

# Computer AI

**Level 1:**

For level 1, all the AI’s pieces are gathered into an array of pointer of pieces. Then it is filtered into a new array with the pointer to pieces to the only the pieces which have valid moves available/possible.

The first priority is to check if the king is in check by calling Grid::isInCheck(char). If it is in check, the algorithm to move it to a location that is not being threatened by an enemy piece is exectuted.

If the king is not in check, then a random number is generated to determine which piece to move. Then a random number is again generated to determine which move to execute out of the list of possible moves generated by the Piece::possibleMoves(Pieces\*\*\*) function.

**Level 2:**

For level 2, the similar gathering of pieces with valid moves are gathered into an array of pointers to pieces.

The first priority is to check if the king is in check by calling Grid::isInCheck(char). If it is in check, the algorithm to move it to a location that is not being threatened by an enemy piece is exectuted.

If the king is not in check, then before executing the same algorithm as level 1 for a random move, an algorithm to check if any enemy pieces can be captured is executed. In specific, traversing the possible moves of all the playable pieces to see if the location of any of the possible moves contain an enemy piece. If it does then it is taken as the move.

If there is no capturable enemy piece, the algorithm to check if there is an enemy king that one of our pieces can check is executed. This algorithm checks for possible moves which can check the king.

If no enemy piece can be captured and the enemy king cannot be checked, a random move is executed.

**Level 3:**

For level 3, the similar gather of pieces with valid moves are gathered into an array of pointers to pieces. This is also done for a separate array for the enemy pieces.

The first priority is to check if the king is in check by calling Grid::isInCheck(char). If it is in check, the algorithm to move it to a location that is not being threatened by an enemy piece is exectuted.

The second priority is check if any of our pieces are threatened of being captured. If any piece is threatened of being captured, it then checks if any of it’s possible moves is safe location (ie. the new location won’t be threatened of being captured). If there is a safe location, the piece is moved there.

If there are no pieces in threat of being captured, then the same algorithm in level 2 for capturing enemy pieces is executed.

If there are no pieces need to avoid capture, nor are there any pieces that can be captured, then the same algorithm in level 2 for checking the king is executed.

If none of the above are possible, then a random move is executed with the same algorithm as level 1.

**Level 4:**

For level 4, the similar algorithm to level 3 for gathering of pieces and enemy pieces is performed. The only difference between level 4 and level 3 is that, if there is an enemy piece that we can capture, it checks if that location is either safe from the enemy OR is protected by allied pieces, if that is the case, then it captures, else it does not.  
If a capture is not performed, avoiding capture algorithm is done.

If there are no pieces in threat of capture, then the algorithm for checking the king is executed.

If none of the above are possible, a random move is executed.

# Graphical Mode

We made use of main(int argc, char\* argv[]) to take terminal commands. If the number of terminal command is more than 2 (which includes the program itself), then it will initiate graphics created by Xwindow. We set it up as a pointer so that even if there is no extra terminal commands, it will still run. When we wanted graphics, it will only be initialized when the number of terminal command is more than 2. We have if statements everywhere we need to print the chess board so that if we are using graphics then it will draw the board using X11 instead of standard outputting the grid. Since the graphical interface is much slower than the standard output, when we move a piece we only update the changes that happened rather than redrawing the entire board. Of course, once we end the program we delete the Xwindow.

# QUESTION 1 FROM PLAN OF ATTACK

The chess program will search through the book for the first move of the game and search for the response that follows such move. For any subsequent moves the opponent makes; it searches for next move the same way. For example, if the sequence of moves were sorted in a tree structure, then you would parse the tree based on the moves that was previous made.

Another option would be to take a list of accepted openings and rewrite them so that they are listed in an increasing move format (Example. Move 1 2 3 will be listed in, {1}, {1, 2}, {1, 2, 3}). Then the entire list will be parsed for a possible matching sequence. After a match has been found, it will look at the next sequence in the increasing move format of an opening which includes the next move (For example, if the current of the game is {1, 2}. It will be matched with {1, 2} in the opening “Move 1 2 3” and see the next sequence {1, 2, 3} and carry out the new move of that sequence which is {3}. This is essentially a tree in a list format.

# QUESTION 2 FROM POA

To allow a player to undo his last move, we record both players’ moves in order of occurrence within a linked list such as:

struct Move {

std::string from;

std::string to;

Move\* last;

};

For a single undo, we just from “to” to “from” of the first node (struct Move) in the linked list, and then you delete the first node (Move).

Similarly, for unlimited amount of undos, you iterate the undo action for a single undo (see above) for the number of undos desired. This could be achieve by a for/while loop.

# QUESTION 3 FROM POA

We would need to change the size of the board, and add more conditions on where a piece can legally move (due to the cross shape of the new board). In the command-line interface, the alternation of turns would be for four players. When you move a pawn to the end of the board, we need to take in account of the left and right ends of the board for a pawn promotion. Also when a players resigns, it doesn’t end the game until 3 players end the game. Also, when the game starts, in the command interface 2 more players will need to be specified. The display of the pieces will have to be changed such that there are two more distinct ways to represent the two new player’s pieces. As a result, in the setup mode, adding a new piece to the board will need for the user to specify the specific player that the piece belongs to.

# Design Document Question 1

This project taught us about working together and assigning different part of the program to each member of the team. Furthermore, we learned communicating how each class worked with each other and how each function is used when combining the split work. We also had to use source control, specially git, to manage our source code and version control.

# Design Document Question 2

If we had the chance to start over we added more features to our programs and also do more real play testing (actual playing the game with each other). Completed the UML to more details including private and protected member so that we can have a greater overview of the structure of the program when we program.