Project Chess Final Design

Collaborated by Kunling Yang, Zichu Wu, and Min Suk Kim

University of Waterloo, CS246, F21

Final Project

I. Introduction

We have proposed to create a chess game for the CS246-F21 final project. After dealing with a variety of different design challenges and many ups and downs, we have successfully created a chess game that is both user-friendly, challenging, and most importantly, entertaining. We've thought of ways to divide the sections in most efficient ways to complete the task and as a result, we were able to finish the project days before the final due date. Even though we all lived in different places and despite the fact that it was not possible four us to meet each other in person, we had frequent team meetings throughout the project and we've often discussed together to solve any encountered problems. We wanted our chess game to be unique and fun. Besides completing the required features for the game, we did a lot of research and discussed our ideas to improve the quality of the game. As a result, we were able to add exciting extra features to make the game different and unique compared to other normal chess games. As it is in the description, the chess game is played on an 8x8 chessboard, arranged so that there is a white square at the bottom right. Players of the game should take turns to make one move at a time. As the description specified, we made our chess game so that the white side player makes the first move and black side player makes the next time move. We have successfully implemented all the standard features as it was described under chess project description and it is possible to see all the extra credit features that we've made in later part of this document.

II. Overview

Our chess program consists of files with different functionalities. First of all, Game class in our chess program controls most of the events during the program execution. This particular class process the user input, notify all the observers, have two displays to render the board, record the move histories, store all the chess pieces currently on the board, and in the dead pool. At the very beginning of this specific chess game, user must input single line of command which include, “game” followed by type of players in order of white and black respectively. Type of players can be either human or bot. This may result in different combination of players, such as human vs human, bot vs bot, or human vs bot. Bot can have different level settings from level 1 to level 4, which determines the difficulty of AI. For level 1, bot will move its pieces in random manner. For level 2, bot will move its pieces to make capturing moves, but will eventually make random moves when no capturing moves are available. When the bot level is set to 3, it will have similar properties as level 2, but the bot will move its pieces to avoid getting captured by the opponent pieces. For level 4, we built an algorithm so that the difficulty level is significantly harder than the previous levels. We’ve implemented minimax function in levelFour class, which returns the best moving command possible using recursive search with a specified depth. When inputting “game white-player black-player”, user must enter number of allowable undo moves for the specific game. Initial board will consist of 16 chess pieces for each white-side and black-side player just like the regular chess game. For the text display, pieces for white-side player will be noted as upper case letters and pieces for black-side player will be noted as lower case letters. After collecting initial information from the user of the program, program starts the game by calling start function for Game object. Every time start function is called, each player must enter “move” followed by initial and final coordinates of the piece. Position of the chess piece consists of x and y values. X value ranges from ‘a’-‘h’ and y value ranges from ‘1’-‘8’. Every time the start function is called, our program converts string position to the corresponding integer positions for the Game class’ move function which only takes integer x and y values. There is no need to specify which chess piece that the player wants to move, but only the initial position and the correct final coordinates of the chess piece that the player wants to move. Whenever the players make incorrect input, error message, errorMsg(), from the Game class will be called. We’ve also successfully implemented castling and promotion of pawn as specified in the project description. Also in the case of a bot player, the command “move” makes the bot player make a move as well. If the initial user input is “setup”, program enters the setup mode, instead of starting the game. For scoring, winner gets one point, loser gets zero, and in case of stalemate (draw), both players earn half a point.

III. Design

We’ve encountered different design challenges while building the program. First, we have decided to use a strategy design pattern for different levels of an AI. This has been discussed throughout the due date 1 plan of attack. We have made four different level classes, LevelOne, LevelTwo, LevelThree, and LevelFour. Bot class, which inherits Player class, owns a strategy pointer, one of LevelOne, LevelTwo, LevelThree, and LevelFour. Human class inherits Player class as well but there was no need to use any design patterns for this specific class. As we’ve already specified in the due date 1 plan of attack, we used an iterator design pattern for player’s move history. We’ve created a header file, moveHistory, using iterator design pattern and a class MoveHisIter, which manages access to the MoveHistory of the players. This allowed the program to iterate through the move history of all the players and generate a move history display on the right side of the board so the players can have an overview. Also, we used observer design pattern for both text and graphical displays, so whenever the Game class changes its states, text and graphical displays are notified by notifyObservers function from Subject class and show users corresponding displays. We’ve also updated the Player class so that this class inherits the Observer and every time it gets notified, program updates all available movements. In this way, program will detect any checkmates or stalemates by players and Strategies can use this to choose one possible move. Despite the fact that we meant to use the template method design pattern for the movement of pawns for each player, it came out to be unnecessary, since the movement of pieces only depend on the initial and final coordinates of player’s input. Also as it was mentioned throughout our first plan of attack, we’ve used the Model View Controller architecture, so that the Game Class acts as a controller where player interacts with it through the input. The TextDisplay Class acts as a view so the players can get feedbacks from the program about bot’s movement and how the chessboard varies accordingly. The moveHistory acts mainly as a model where actual functionalities are implemented.

IV. Resilience to Change

From what we have learned throughout the course, we were very aware in designing our code to maximize cohesion and minimize coupling. Throughout creating the chess game, we made sure to minimize the amount of dependency between the modules. In case we need to change the game setting, but leave the players settings as it is, we just need to modify the game.cc file that we have created for the project. Also, whenever we wish to update the interface of the chess project that we have created, it is only sufficient to modify the textDisplay and grahpicalDisplay files. We put all the significant features in different files, which include type of players (Human and Bot), Interfaces(TextDisplay and GraphicalDisplay), game(Game), pieces (Rook, Knight, Pawn, etc) , movement (Move), and position (Post). Therefore in case whenever the various changes are needed, we can always change the selected .h and .cc files. We also insure the high cohesion of modules that we have created as well. Everything in the unit that we've created is very closely related as it is possible to see in the UML file that we provided. By maximizing the cohesion of modules, it will make the maintenance easier and decrease the occurrences of errors.

V. Answers to Questions

Question 1: Unlike what we have discussed in due date 1 plan of attack question, we decided not to provide opening move strategies, such as Ruy Lopez, Italian Game, and Sicilian Defense to the players. It was way more complicated than we thought to store such opening moves and provide the best possible moves to the players. Even though we weren't successful at providing such standard opening move strategies, we created minimax function using Minimax algorithm. This particular function is used by AI with LevelFour class and this lets Bot to move its pieces to best possible position in the chess board.

Question 2: Answer to this question has not been changed from the answer to the question in due date 1 plan of attack. We created a vector that stores moveHistory of all the players during the game and whenever players decide to undo their previous moves, program pops the existing vector by the number that each desire to undo and decrease the count of available undos(numUndo) by the desired number. For example, if 2 players are playing the game and if one of two players wants to undo 3 moves, we pop the existing vector by 2x3=6 and decrease the number of undo counts by 3. Before the game starts, program would ask the user to input value for the undo counts (allowUndo). We provide allowable undo counts equivalent to the initial user input for each human player so that every player would have the same number of undo counts throughout the game.

Question 3: Different from what our group planned to implement, we aborted creating a four-handed chess game. Instead, we focused on improving the original two-handed chess game.

VI. Extra Credit Features

1. **Add an undo function for the chess game.** A class named *MoveHistory* is added to the project, every time a *Player* (*Human* or *Bot*) puts a valid move command, an instance of *MoveHistory* adds a piece of move information into itself. This class generally acts the same way as the other Observer classes, but there is one crucial reason not to make it inherits from an Observer, that is AI. The Level Four AI will stimulate the moves on the board but these are not actual moves. Once a step is stimulated, only the board receives the notifications but not the *MoveHistory* so we are not generating unnecessary move information and deleting this information after the best step is calculated by the AI.
2. **The *MoveHistory* is designed with an iterator.** When the *TextDisplay* gets notified, we would like the *TextDisplay* to render several latest movement information on the right side of the console, thus an iterator facilitates this process. Every time the *MoveHistory* is updated, the renderBegin() and renderEnd() is updated. When *TextDisplay* is printing the move histories, a simple for loop with corresponding iterator can be used.
3. ***LevelFour* AI adapts the Minimax algorithm with alpha-beta pruning.** Basically, the AI takes advantage of the *evaluateMove* function inside the board class so the AI can calculate the effect of each move and takes that action. For a given *DEPTH*, the AI anticipates that many moves ahead. First the AI considers the best move (a move calculated by *evaluateMove* that has the greatest effect towards the game), and assumes to take that step, modifying the board accordingly. Then for the second step, the AI tries to identify the worst move for itself (since the AI is now considering the opponent's move) and assumes the opponent takes that action and modifies the board accordingly. The above process is repeated until the desired DEPTH is reached, and the AI will evaluate the whole board at this time returning the effect scores of this board (positive or negative, depending on whether the AI is thinking for the opponent or itself.

   Additionally, alpha-beta pruning is adapted to accelerate this process. The alpha value denotes the best score for AI, and the beta value denotes the best score for the opponent. The initial value for alpha is positive infinity and the initial value for beta is negative infinity. If after searching a specific situation, beta <= alpha, that means choosing this situation will reduce the AI's effect and thus all subsequential searches can be omitted.

4. **Improve ways of evaluating specific moves.** Firstly, a piece itself has an effect, of course, such as a pawn has a weight of 5, and a bishop worth 20, and a queen worth 100. But we are not satisfied with only determining move effect by piece weight. After some further research, we are inspired by chess wiki and it is found out that certain pieces have a higher effect towards the

game compared to other pieces. For example, a knight can control more positions when it is in the middle of the board compared to it at the edge or at the corner of the board; a white king is potentially safer when closer to the lower board (Row 1 or Row 2) since more white pieces should be around there generally speaking, and vice versa for the black king. But such pattern does not apply to pieces with high mobility, such as rook and queen (not bishop, since it's controlling area is significantly limited when itself is cornered), and thus the position weight does not change apparently from cells to cells. A full position weight table can be seen in the *move.h* file.

VII. Final Questions

a) Completing CS246-F21 final project required a lot of team effort. In order to finish the project in a given time, we had to divide the tasks so that everyone could work on the project in the most efficient manner. Our team has three group members and we are all living in different places. We never met each other in person, but we could work very efficiently by having frequent team meetings through Discord and sharing our codes and documents through Github. We had to be strict to ourselves to complete the project in the given time, so we set exact dates to show what each of us had done so far. This final project taught each and every one of us how to cooperate and be punctual to complete large tasks in a short time.

b) Even though the result of our project came out as we expected, if we had the chance to start the project over, instead of integrating most features inside the Game class, we would redefine functions and objects in different classes. Also, we were unable to provide features such as standard opening move strategies and four-handed chess game due to time consuming reasons. If we could start the project over, we would spend more time thinking of ways to provide more user-friendly features such as opening move strategies and four-handed chess games.

VIII.Conclusion

The CS246-F21 final project was somewhat both a fun and challenging experience for all three members in our group. Even though we've encountered a number of problems with high levels of difficulties, we figured out how to solve the encountered challenges as a team. We learned that for a team project, all members in a group must put great efforts to complete the task and it was necessary to discuss the problems as a team and help each other to solve the problems. As a result of putting in a great amount of effort and time, our chess game came out as a more consistent and very accurate piece than what we've all expected. Even though we were unable to implement some features that we planned, most of the features that we planned had been successfully created. As it was mentioned in the plan of attack from due date 1, we've used Github to share and update our files. We were able to collaboratively work on code for the given chess project and the version control system provided by Github certainly benefited our group to work more efficiently throughout the final project. We have made more than 150 commits to our private repository in Github from November 24, 2021 till the due date and this indeed shows how much effort that we put for this chess project. By completing this final project, we all feel very confident about doing any future projects that we may face. We learned that any challenging problems can be solved by planning ahead in time and nothing was impossible to implement with proper research and time.