CS 246 Final Project

Due Date 2

Final Report

Arion Harinarain
Ishaan Bansal

Dec. 5, 2023

CS 246

University of Waterloo

Fall 2023

Table of Contents

Introduction

The game of chess is among the oldest and most influential games in human history. It is played all over the world, and it is estimated that 70% of adults have played chess at some point in their lives. The game can be traced back to many countries around the world, like China, Japan, and India, with the modern rules of chess forming in Europe at the end of the 15th century. Chess is a game of strategy and contains no elements of luck. Chess also maintains a strict set of rules on piece movement and game flow that players must adhere to at all times. Players compete on an 8x8 grid, each controlling sixteen pieces: one king, one queen, two rooks, two bishops, two knights, and eight pawns. White goes first, then Black follows. Players have to move their pieces strategically (given the rules of movement for each piece), to gain an advantage over their opponent. The game is won by checkmating the opponent's king, threatening it with inescapable capture.

The fields of computer science and chess are heavily intertwined, as the process of selecting moves and creating strategies is a logical decision-making process. Computer science principles can be applied to this process to make it more effective than a human could ever be. One of the largest goals of computer scientists and one of the first practical applications of AI was a chess program, Deep Blue, developed by IBM. In 1997, Deep Blue defeated the world chess champion, Gary Kasparov.

Implementing the game of chess also allows us to use the lesson taught in the course, particularly emphasizing Object-Oriented Programming (OOP) principles. By simulating the strategy and strict rules of chess, this project allows us to explore encapsulation, inheritance, and design patterns. Creating this project allowed us a practical and engaging platform to understand and apply these concepts.

Overview, Design, & Classes

Overview

Our implementation of chess contains five main files (Board, Display, Player, main, window) which hold the separate classes used in our program. The Board file contains the Board class which holds the rules of chess, representing the game's state, managing piece movements, and enforcing legal moves and check/checkmate conditions. The Display file contains the classes for displaying the board both as a text display and a graphics display. The Player file contains the Human class and Bot class, which holds the logic for generating a move. The main file contains our command-line interpreter. The window file supports our graphics display.

Design

The design of our chess program follows the Observer pattern and is centered around the Board class that uses STL maps for an efficient representation of the chessboard. The STL map uses the location of the pieces as keys and the piece types as values. To manage piece placement and movement, we'll check if there's any at the location using maps built-in count feature and make a decision accordingly. Rather than creating classes for a piece and then creating each piece separately, we feel this is simpler as it reduces code while still ensuring encapsulation. This

Board class is the heart of our game logic, mapping board positions to chess piece types, and managing their state and interactions through encapsulated methods. Players interact with the game through the Human and Bot classes, which encapsulate the logic for human input and AI decision-making. These classes interface with the Board class, requesting moves that are validated and executed according to the rules of chess encoded within our Board logic. The TextDisplay and Graphics classes serve as our output modules, rendering the game's current state in both textual and graphical forms. They are designed to observe and reflect changes in the game state, using the Observer pattern to update the display whenever the Board state changes. Our Command Interpreter acts as the bridge between the player inputs and the game logic, parsing commands and translating them into actions on the Board. We've designed our program to ensure that each part operates both independently and in sync with the other classes, and this will allow us to create a robust and flexible chess game.

Board

The Board class in our chess program is an essential component of the Observer pattern we've implemented. It acts as a central point of communication between the state of the chess game and other components that need to respond to changes in that state, such as the Display classes. The class serves as the central point of the program, providing a comprehensive framework for the game's mechanics and state management. It encapsulates the entire game board using an STL map, where keys represent the coordinates on the chessboard and values denote the specific chess pieces. This class is responsible for managing the placement and movement of pieces, ensuring all moves are legal and valid based on the established rules of chess. It checks for move legality, handles special moves like castling and en passant, and maintains the overall game state, including the tracking of turns and the detection of check and checkmate conditions. The Board class also interacts with the Player classes, through the inputs made in the command-line interpreter, to process and execute their moves. The Display classes depend on the Board class as they provide a user interface that visually represents the information held within the Board class.

Player

The Player classes in our chess program are designed to represent the participants in the game. It serves as the interface through which players (Human or Bot) interact with the game, handling inputs and translating them into actionable moves on the chessboard. This class encapsulates all aspects of player interaction, including move selection and response to game conditions. It's structured to communicate with the Board class, sending move choices that are then validated and executed according to the chess's rules.

Human

The Human class in our chess program is designed to take input from a human player, and then using the command interpreter, it turns the human's input into an actionable command.

These commands can range from moving a piece on the board to making special moves or accessing other game features (as outlined in the project specifications). The Human class acts as a bridge between the player and the logic of the chess program, ensuring that the player's actions are accurately reflected in the game and adhere to the logic of the program.

Bot

The Bot class in our chess program is designed to simulate a computer opponent, providing a non-human challenger for the player (As well as, having bot vs bot). It encapsulates the logic required for making automated decisions during the game, including move selection and strategic planning. The Bot class operates by analyzing the current state of the Board and determining the most advantageous moves based on a decision-making matrix and difficulty settings. The Bot class holds a pointer to the board, so that it can get the current position of the pieces, and the board and bot are friend classes. As per the project outline, we plan to have 4 different levels of difficulty for bots. This class interacts with the Board, executing moves in a way that adheres to the standard rules of chess.

Bot1

Bot1 first randomly orders all the pieces and then picks the first one and evaluates all possible legal moves for each piece, including complex ones like castling or en passant, then randomly selects and returns one of these moves, ensuring unpredictability in the bot's strategy. If the piece has no moves, it moves on to the next and repeats this process.

Bot2

Bot2 considers different move possibilities, such as forward moves, diagonal moves, captures, promotions, and special moves like en passant and castling. The bot checks for each move whether it would put the opponent's king in check and whether it captures an opponent's piece. It puts all of these preferred moves in a list and then selects randomly. If none exist, Bot2 calls Bot1's getMove() function to randomly get a move.

Bot3

Bot3 is an add-on to Bot2. It performs the same checks that Bot2 does but also evaluates whether the move exposes a piece to immediate capture, helps in capturing an opponent's piece, and how the move affects the overall safety of the king. Using this logic, a list of preferred moves is created and then one is selected. If none exist, Bot3 calls Bot1's getMove() function to randomly get a move.

Bot4

The Bot4 is an upgrade on the Bot3. It implements a more advanced move evaluation and implements a 'points system', which means it prefers moves that capture pieces of greater point value. As well, it builds on the logic of Bot3 by looking for moves that either put the opponent in check, capture a piece, or avoid capture. Then, it first looks for a move that's a combination of the three moves, then isn't captured and checks the opponent, then captures and doesn't get captured, then captures and checks, then a move that captures, then a move that doesn't get

captured, then a move that puts the King in check. This way, the bot ensures it gets the possible move according to its list of moves its prefers. If none exist, Bot4 calls Bot1's getMove() function to randomly get a move.

Display

The Display class in our program serves as the framework that encapsulates both the TextDisplay and GraphicsDisplay classes. It is the central point for which output is put to the user. This class is designed to provide an interface, for both the different forms of display.

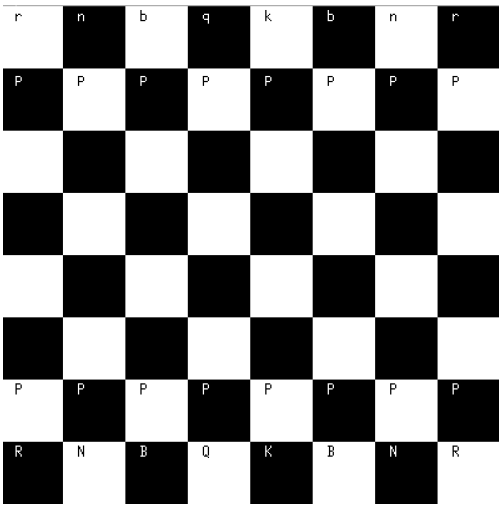
TextDisplay

The TextDisplay class in our chess program serves as the primary interface for presenting the game's visual aspects to the player. It is responsible for rendering the state of the chessboard, including the position of pieces and any relevant game information. This class acts as a bridge between the game's internal logic and the player's understanding of the game state. It updates the display in response to changes in the Board class, ensuring that the player always has an up-to-date view of the game. As per the project outline, the display class with provide the user with a text display, in this format:

8 rnbqkbnr
7 pppppppp
6 _____
5_____
4 _____
3_____
2 PPPPPPPP
1 RNBQKBNR

Graphics

The Graphics class in our chess program is an extension of the Display class, specifically focused on rendering the game's visual elements in a more elaborate and visually appealing format. This class takes charge of creating a graphical representation of the chessboard, using graphics to illustrate the board, the pieces, and any relevant game indicators. It utilizes XWindows graphics. It provides a display in this format:



Resilience to Change

The design of our program was meant to accommodate for future development. The clear definition of classes and their responsibilities, along with the use of inheritance, allows for the introduction of new features without disrupting existing functionality. As an example, the current traditional logic of chess that is held with the Board class can be changed to a variation, such as Chess960, and with auxiliary changes to the logic in developing moves, the other classes, like Display, will still act as expected. As well as this, the Player class can easily accommodate for the addition of other Bot classes, such as Bots with different play styles, and the other classes will still act as expected. The Display classes can also accommodate different types of displays without the need for further changes. However, the largest reason our program is very adaptable to change is our use of STL maps to manage the gameboard, movement, and the state of pieces. If we ever need to change the way pieces move or add new pieces with different rules, these modifications can be implemented with minimal refactoring. If we had a design that had classes for pieces for example, that means we would have to modify that class, and then modify the Board class to accommodate for those changes. We just need to change the way the maps are utilized in Board. Our use of vectors and maps to handle game moves and piece positions also offers scalability, and if needed, we could implement changes to the layout of the board.

Question 1

In the game of chess, many players consider the opening to be the most important stage of the game. It determines the overall strategy you will implement for the match and will set you up for the mid-game and end-game. As such many “openings” (a systematic sequence of moves to begin the match) have been developed. Many openings center around one or two specific first moves. For example, the Sicilian, French, Scandinavian, and Caro–Kann all start with the same move, e4. The Grünfeld, Catalan, Slav, and London System all start with d4. All of these openings may start very similarly, but they do have differences. For example, in the Sicilian Defense, which begins with 1.e4 c5, Black aims to control the center from the side and often leads to asymmetrical and dynamic positions, favoring complex and tactical play. In contrast, the Caro-Kann Defense, starting with 1.e4 c6, is known for its solid and resilient nature. It allows Black to support the d5 advance on the next move, aiming for a more structured and less confrontational pawn structure than the Sicilian. While both openings arise from 1.e4, the Sicilian is generally more aggressive and open, whereas the Caro-Kann is more defensive and positional. From this example we can see that even though openings may differ, the end goal and logic remain the same, to fight for space on the board. Since the goal is the same for every opening and its various variations, there are a few principles that every opening follows:

Control central space with your pawns - This allows for more flexibility in future moves and denies space to the opponent’s pieces.

Castle early - Once pieces have been developed, and some have been taken, it is dangerous for the King to remain in its place, thus the need for castling.

Develop pieces - By developing your pieces, you put them in a position to make impactful moves that can dictate the flow of the game.

Safety first - While it is good to be aggressive and try to attack your opponent, it doesn't make sense to unsafe moves that allow your opponent to win free material. Carefully assess the safety of each move before making it.

If we had to implement a series of standard openings (apart from hardcoding), the way that we would do this would be to give the computer the logic as outlined above so that it can follow these fundamental principles of chess during the opening phase. The bot would be programmed to prioritize controlling the center with pawns, developing its pieces efficiently, and castling early to ensure the king's safety. Additionally, the bot would be designed to make moves that are not only aggressive but also safe, avoiding unnecessary risks that could lead to a loss of material. By incorporating these strategic elements into the decision-making process, it could choose from a variety of standard opening sequences based on these principles, rather than relying on a predefined set of moves. This approach would enable the bot to adapt its opening strategy to different situations, making it more versatile and challenging for human players.

Question 2

Many chess programs offer the functionality of allowing users to undo their move if they so wish. With this comes many problems, as the board reacts dynamically to a user's move, i.e. the rest of the pieces react to a move and when a move tries to be undone, it can be hard to reverse the reactions of the other pieces. To combat this, the design of the program must be able to recall previous information. One design approach to this could be to implement a stack to keep track of all the moves made during the game. Each time a player makes a move, you push the move and board details onto the stack. The move details should include all the information necessary to reverse the move, such as the starting and ending positions of the pieces, any captured pieces, and any special move flags (like en passant or castling rights). To undo a move, you would just pop the top element from the stack. Using the information in this popped element, you reverse the move on the board. This would involve moving the pieces back to their original positions and restoring any captured pieces. If the move involved special rules like castling or en passant, those would need to be reversed as well. For unlimited undo, since the stack keeps track of all the moves, you can continue to pop moves off the stack and reverse them, allowing for an unlimited number of undo. You only need to ensure that the stack isn't empty before attempting an undo.

Question 3

Four-player chess is a variant of chess that allows for 4 players to play against each other at the same time. To accommodate for this variation, many changes to the board, player, bot, and user-interface classes would be needed:

Board: The most significant change would be to the board class, as it would have to be extended by 3 rows on each side to accommodate the extra players. As well as this the logic for legal and valid moves would need to be updated. Also, we would need a new way to distinguish which player holds a specific piece.

Player: The player class would need to be updated to account for four players, instead of the current two.

Bot: The bot class would need to have updated logic on whether to determine the correct next move. Currently, the bot is only evaluating moves with the consideration of one opponent. Having an additional two opponents, the decision-making process would change significantly and the bot would have to determine the trade-off between the positions of the various opponents.

User interface: The user interface classes would need to change to reflect the updated board design.

Extra Features

Feature #1

One extra feature that we have implemented in our chess program is a game log. This game log serves as a dynamic record of all moves made during a game, and this provides players with a comprehensive overview of the game's progression. The log captures each move by a player, making it easy to review and analyze game strategies post-match. We recognize that many chess players, both beginner and advanced like to review how their matches went. This log can be instrumental for players looking to improve their skills by studying their moves and those of their opponents. To implement this feature we used fstream to write the relevant information into a text file called “game_log.txt” that players can access later. The addition of a game log will significantly enrich the user experience, making our chess program not just a tool for entertainment but also a valuable resource for learning and improving chess skills.

Feature #2:

The second feature that we have implemented is the addition of “puzzles”. The puzzle functionality in our program involves creating specific chess scenarios that challenge players to achieve a certain objective within a given number of moves, such as checkmating the opponent. We have implemented two simple layouts of the board for the player to “solve”. The objective of the player is to checkmate the opponent in two moves, and if the player doesn’t achieve this, the puzzle is failed and the player can try again. The puzzle feature would not only provide an enjoyable and engaging way for players to improve their chess skills but also offer a different mode of interaction with the game, apart from standard matches. We developed this feature by

leveraging our current features. We used the current setup functionally to initialize a specific board configuration. Then the player would play against a bot during the puzzle. A count is made of the player's moves, and if they have made more than two moves, they have failed the puzzle.

Final Questions

Q1:

Working on a software project as large as this one provides many valuable lessons, mostly around how to develop software in teams. Developing software in teams is an essential skill for developing large projects. During this project, we have learned several valuable lessons. One of those is the effective communication of ideas. The process of developing complex software like a chess game requires a clear and concise exchange of ideas among team members. This project has taught us how to articulate our thoughts, listen actively to others, communicate technical concepts, and engage in discussions. This was critical in ensuring that we avoided making bad design decisions and faulty implementations. Another significant lesson learned is how to make use of tools like Git and GitHub for collaborative development. These tools have helped manage our codebase, track changes, and integrate the work done by different team members. We learned to create branches for specific features or fixes, merge changes without overwriting others' work, and resolve conflicts that arise during these merges. We also learned the importance of thorough testing and making sure that all members of the team test their implementations extensively.

Q2:

If we had the chance to start over, we would make a few changes. A more thorough initial planning phase could have benefitted our group. If we had a more detailed schedule, we could've been more efficient in our development. As well as this, if we had scheduled specified times to meet and communicate with each other, we could've had better team cohesion and quicker resolution of issues.

Conclusion

Working on a project as large as chess comes with many challenges and lessons to be learned. This project offered us insights into how to develop and collaborate on large software projects. We've both taken many lessons from this project including, how to design a large-scale project, communicate, use collaborative tools, and test large programs. The project not only enhanced our technical skills but also sharpened our ability to work as a team. It also allowed us the chance to implement the course content (design patterns, inheritance, maps, etc.) in a challenging, practical, and interesting way. With both of our group members being avid chess players, it has allowed us the chance to learn more about the game we are interested in. We move forward from this project with a deeper understanding of the technical concepts, the importance of collaboration, and the continued interest in an age-old game.