CS246 F23 - Design - Kush (k23patel), Rehan (r2anjum), Ali (maaasyed)

Introduction

Chess is a game which has been around for centuries and is known by everyone, regardless of age. Hence, we picked a game which had tons of complexity, with tons of flexibility with both moves and design. This isn't new though, as even today, there are still new variations of chess, like blitz and four-handed chess.

Our group took a very simple approach to the design of chess which was displayed with not only graphics, but also software design. It was evident across all groups and not just ours that you could have the ultimate flexibility with what colours you wanted for the board and how you wanted to display the pieces. Contrarily, the computer algorithm could have been as complex as you wanted. It was the case where some groups chose to make their level 4+ computer a sophisticated algorithm, whereas other groups simply chose a famous, or well-winning opening. The options were endless.

Therefore, we present to you Chess with a spin on graphics, computer levels and a spin on a new game, all implemented in C++ with object oriented design and your very own set of AI computer levels.

Overview

This section will describe the overall structure of our project. To begin, let's discuss the structure of the Board. The Board is a class that contains a 2D vector of Piece pointers. Each element in this 2D vector represents a square on the board and the piece that is on it. The Board class is a concrete Subject which has two concrete observers which are the TextDisplay and GraphicDisplay. The TextDisplay is notified whenever a move is made and the entire TextDisplay is recreated. The GraphicDisplay notify function takes the parameters of the Piece's old and new coordinates.

When initially setting up the board, the board is always created with an 8 by 8 vector of blank Pieces. If using the "setup" command, the Piece at the coordinate would be returned as a Piece pointer and deleted and replaced. The "+" command would swap it out with the preferred Piece and the "-" command would swap it out with another blank Piece.

Now let's take a closer look at each Piece and its functionality. The Piece object follows inheritance where the individual classes of Pawn, Rook, King, Queen, Knight, and Bishop inherit the Piece class. Thus, each Pawn "is a" piece and this relationship works for all pieces. When creating the Board, the 2D vector of Piece is initialised with the superclass Piece. One key field

that the Piece class has is the blank field. Essentially, whenever a Piece is captured or or it moved, two Pieces are swapped and the captured piece is set to blank. When a Piece is moved to an empty square, the Piece on the empty square and the Piece being moved are swapped. When notifying the TextDisplay and Graphic Display, if the Piece's blank field is set to True, then it won't be displayed, that square would be empty.

The Piece class has a virtual method called calculatePossibleMoves() which calculates the possible moves that the given piece can take since each piece has its own rules of movement. The method would return a vector of pairs which would contain the coordinates of the possible square it can move to. This pair has the format of (row,col). This way, the 2D vector of Pieces would be able to get the piece at and set the piece at the given position. This is because the 2D vector also uses the a [row][col] indexing when accessing elements in the 2D vector. The convertToCoord() function converts the letter/number input for the individual squares into coordinates. It gets the pointer to the Piece at the first coordinate and calculates its possible moves which returns a vector. We then loop through this vector to see if the second coordinate is present. If it is, it is a valid move and if it isn't, then the user is requested to choose a different move.

Now, let's take a look at the special move. Castling was one of the special moves that we integrated. The King Piece has a boolean field called moved that is set to True if the King has moved. If the moved boolean is false, the King can move two spaces on either side. These two coordinates are included in its possible moves vector. When the movePiece function encounters the King moving 2 spaces, it checks if the Rook has also moved by checking its boolean field and it also checks whether the spaces between the Rook and the King are all blank. If these requirements are met, then the castling is done by swapping the King and two spaces to either side specified and the Rook climbs right over the King. Another special move was the Pawn moving two spaces. If its boolean field of moves is set to false, then the Pawn can possibly move two spaces up and this is included in its possible moves vector.

The three evaluations of check, checkmate and stalemate were done quite similarly with the key component being the possible moves vector. Every time a move is made, the program checks for these three evaluations. For check, the program would go through the opponent pieces and its possible moves vector. If any coordinate in that vector matches the coordinate of the current player's king, then the current player is in check. If any move of the current player results in the current player's king to not be in check, then that is a valid move. However, if no possible moves exist to get the king out of check, then there is a checkmate and the game is over. For stalemate, the initial state of the board can not result in any player's king being checked. The stalemate algorithm first checks if its king is in check and if it is, then stalemate is not possible. It

then traverses through each of the current player's pieces and its vector of possible moves. If any of these moves results in the king not being in check, then stalemate is not possible.

The following paragraph will explain the concept of the computer Levels. For Level 1, we traverse through the computer player's pieces and compute its possible moves vector. For each pair in this vector, we return another pair which consists of the current coordinate of the piece and the move. The computer then randomly selects a move and does it..

Now, let's discuss levels 2 and onwards. The thought process behind level 2 was simple, we wanted to maintain the fact that a level 2 computer was fairly easy to play and was not thinking too far into the future or trying to play against opposing player moves. The way we implemented this was to follow suit with level 1's randomness to maintain ease but to prioritize capture moves. To do this it was fairly simple as our calculatePossibleMoves functions within Piece decorator classes actually already tried to find possible capturing moves, so we simply made another function within the piece decorators called capturingMoves which returned all moves that proved to be captured, essentially stripping away most of the code from calculatePossibleMoves. This proved to work as expected, when the opposing player leads a piece into a capturable position the level two computer will prioritize capturing the piece and won't give thought to whether it's a safe position after capturing in terms of gaining or losing material on the next move of the opposing team.

Level Three is where things started to get interesting in regards to development. We had to create something smart enough to evaluate possible captures on its own pieces and thus, our team instantly thought of the minimax algorithm which most chess evaluation engines. The reason is is because the minimax algorithm runs an evaluation of certain depths on game positions so in example if the computer is the black player our levelThree computer runs our minimax algorithm at a depth of two meaning the function will evaluate two player moves into the future allowing it to evaluate material value scorings(pieces have material strength values i.e. Queen has a material value of 9) of future moves then maximising and minimising tree of player turns and moves until it falls to the maximised score in the favour of the computer player. The algorithm is recursive and creates a tree structure on each depth run then unravels once it hits the base case which is a depth of 0 after which it goes back upwards to give a finalised maximised evaluation in the favour of the computer player and return the bestMove which caused this evaluation. The minimax algorithm actually required a lot more infrastructure then we thought it would, we had to integrate a move stack system specifically for it so that when it runs it future evaluations it can make changes to the board but then reverse them using the reversePiece function. LevelFour actually works based off the same logic of levelThree the only thing that changed was the depth given to the minimax function which instead of a depth of 2 does a depth of 3 allowing it to evaluate 3 player moves into the future. Finally, we wanted to really push ourselves with something better then the well-renowned minimax algorithm and so we decided to add the StockFish chess engine as our levelFive. Simply put, we utilized the StockFish

executable and API to send commands to the StockFish engine and the engine used the board position currently and the current player move to send us back the bestMove in favour of the player. The reason StockFish was levelFive was because although StockFish uses the same minimax logic, its evaluation of board positions is a little different as its material value takes into account a variety of things like all piece positions capturing possibilities and the safety of its own pieces in regards to moves it makes. All-in-all adding the computer player was a daunting task but with some determination and genuine interest in how the heuristics and algorithms of a chess engine works, we were able to persevere.

Design

The most important design pattern that we used was the Observer Pattern. Initially, we planned to have a Board of 64 squares and each square would have a Piece. This would be an aggregation method where a Board “has a” square and each square “has a Piece”. However, we ran into a problem which we were unclear about how to solve: How would a square observe itself? If we implemented this similar to the Lights Out game, it would increase coupling as each square would have to be observed, increasing the interdependence. Each piece would need to know what its neighbouring pieces are to indicate whether or not certain moves can be made. The king needs to know if it can castle and every piece needs to know if it can capture. For a pawn since its default move is to move up, it would need to know if it can capture diagonally or not. Furthermore, if our concrete subject was the individual square, the TextDisplay and GraphicDisplay would have to observe each square. As a result of this complication, we decided to scratch aggregation. Instead, the TextDisplay would observe the entire board.

Every time a move is made or the 2D vector of pieces is modified in any way possible, we would call notifyObservers() and the TextDisplay would update. It would change and recreate the entire TextDisplay board. The issue came when working with the GraphicDisplay. We noticed that if the GraphicDisplay recreated the whole board, it would take a lot of time to compile. In order to make this more efficient, we created another notify function within GraphicDisplay which would be passed in two coordinates representing the coordinate of the two squares which are being modified. The GraphicDisplay would only recreate these two squares.

The second Object Oriented Programming method was Inheritance. This relationship is between the Piece superclass and its specific Piece subclasses. The relationship essentially indicates that each Piece “is a” Pawn, Rook, Knight, Bishop, Queen, or a King.

Additionally, we wanted to implement the Decorator Pattern for our Computer Levels. The default would be Level 1 where the computer can make any random legal moves. The decorators on top of that would be the additional levels including levels 2, 3, 4, 5, which simply adds on to the default functionality. If any new features for the computer functionality were to be

added, it would be added under the Decorator Pattern. However, due to the time limit, we were unable to implement this correctly. It was planned

In terms of Cohesion and Coupling, we attempted to follow the Model-View-Controller pattern as much as possible. We also tried to make each class less dependent on each other while making sure that each function within a class is tightly related to each other. We implemented low coupling by making sure the changes to one module would require none if not any changes to the other modules.

Resilience to Change

If one of the changes was adding a piece, our program can easily adapt to that by simply creating the Piece's class and in the Piece's calculatingPossibleMoves() function, we would simply calculate the moves based on the Piece's functionality. Overall, if any changes were to be made, they can easily be kept track of. Furthermore, we always kept the state of the board on a stack, had our own custom syntax for the state of the board and developed our own algorithm, so our overall software design was highly modular. This was, and would be helpful in the future if any new changes or last minute changes would be needed.

Answers to Questions

Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. See for example https://www.chess.com/explorer which lists starting moves, possible responses, and historic win/draw/loss percentages. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

Although there exist many different ways of implementing standard opening move sequences, two viable options which come to mind are the graph and hashtable data structures.

Firstly, a graph data structure could be used to store opening sequences and the moves following said opening sequences. A node would be representative of a specific move, and the respective adjacency list of that node would contain possible responses to the opponent's moves. Historic win, draw and loss percentages would be held in each node in the graph, as part of the move, depending on whether it would be stored as a struct or as an object.

Contrarily, a hashtable implementation could suffice. Many popular chess engines implement standard opening move sequences via hashing or "Zobrist Hashing".[1] By amortized analysis, a hashtable would allow search, insert and delete operations in O(1) time. Each unique store associated with the hashtable would contain evaluations of historic win, draw and loss percentages.

How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?

To implement undo functionality, a stack could be associated with each player. After each move either player would make, the state of the board and the move itself would be pushed onto the stack. Likewise, boolean flags would be utilized to keep track of the game's current state, letting either player undo more moves. This would result in both players having the ability to undo one or more moves depending on which player's turn it is. An empty stack for either player would indicate that the board is now at a starting state (i.e. no moves have been made).

- A stack that will store the state of the board after each move.
- A function that performs the push() function to add the current state of the board to the top of the stack
- Once the undo() function is called for either player, the state on the top of the stack is popped off. The current state of the board is switched to the new state which is on the top of the stack.

Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game.

Regarding the structure of the Chess Board by itself, the text and graphics displays must be modified to accurately, visually represent the four-handed chess board. With new dimensions, there would have to be a new board which has to be set up with enough pieces and appropriate positioning for each piece. In terms of the players and their respective rules, we must increase the maximum number of players to 4, with functionality for any combination of human and computer players. Additionally, we must keep track of which player's turn it is and constantly check for a possible checkmate or stalemate for each player following each move. It is also important to consider the possible interactions between pieces on the same team and those on the opposing players. A piece can interact with any of the 3 opposing player's pieces.

[1] Lv Huizhan, Xiao Chenjun, Li Hongye and Wang Jiao, "Hash table in Chinese Chess," 2012 24th Chinese Control and Decision Conference (CCDC), Taiyuan, China, 2012, pp. 3286-3291, doi: 10.1109/CCDC.2012.6244521.

Extra Credit Features

Enhanced Graphic Display

The first extra feature we implemented was the enhanced graphic display. Our graphic display consists of aesthetically pleasing colours that try to duplicate the look of various chess programs that already exist. Additionally, we added images for each piece on the board. This required a lot of research as we were initially unfamiliar with X11 and its functionalities. With a couple hours of research we were able to figure out how to draw these images. We ran a Python script using OpenCV that converted the image into a NumPy array. The array was a matrix of pixels. We then copied this matrix of pixels and converted each pixel value into colours. We then developed and used our own bitmap function to draw these images at a given location. The challenge came when the Board class would notify the Graphic Display. We overcame this challenge by making the different colours and images global so that the program would have access to it. With this, the Graphic Display was created.

Graphical Score Board and Piece

Another feature we added was the scoreboard. In the bottom right of the screen, we display a block which shows who's turn it currently is, what the score for each player is, and in the case of a check/checkmate, we would display that in all capital letters as well. Additionally, when a king is placed under check, the background colour for the king's square is turned into the colour of light red. When the game is in checkmate stage, the square is turned into dark red, indicating the game has ended.

Level 5 Chess Engine

Aside from our level 4 for our CPU player, we also implemented a stockfish algorithm which utilised an external binary. The implementation of an external algorithm was extremely useful as it implemented features that our own algorithm didn't have, for instance, alpha-beta pruning. Some of these optimizations would have highly decreased the time for the algorithm. Our level 4 cpu currently goes up to depth level 3 or 4, but our level 5 binary would allow the cpu to go up to extremely high depths with sophisticated and calculated openings.

Undo Moves

As an extension and core part of our level 4+ cpu, we had to implement the functionality to undo moves. Consequently, we implemented our own syntax to keep track of the board's state at any given moment, including properties for each piece. Furthermore, we implemented a stack

to keep track of the board's moves. As a result, we would be able to keep track of the board's different states with low memory usage and have the ability to apply any state at any given moment. This was a major key development as it allowed us to use different levels of depths with our minimax algorithm that our level 4+ cpu utilised.

Final Questions

What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

The first lesson that each of us learned about developing large software in teams is communication. A good first step was sitting together to go through our initial plan and the UML. We knew that the UML was bound to change once we had started to develop the actual program, but attempting to discuss every aspect of the program and the potential classes, fields, and methods to create gave each of us an idea of how to approach this project.

However, as we split our roles and the project into different parts, we started to realise that communication played a significant part in our success. There were many times where one of us needed functions, access to fields, output format, input format and other information for our own functions to work properly. At that time, we would all have to get together and discuss the best way of passing information from class to class or function to function. For example, passing the computer move to the board to make the move which then passes it to the notify functions for the text and graphic displays.

The next lesson we learned was the importance of comments, documentation, and code readability. Oftentimes, we would help each other debug and fix segmentation faults or simple output errors as a fresh mind would have to read and understand all of the code. In order to ensure we didn't torture ourselves when debugging was by documenting our code and organizing our files in folders for easy access. By doing so, we were able to debug much more efficiently. This would also help whoever is reading our code to understand it.

The last and probably the most important lesson we learned was the significance of Git. To give everyone access to the project, we created a shared repository on GitHub. In the beginning, we had to do research to see how everyone can work on their parts individually and then how we can merge all the code back together. We realised that if we waited too long to merge, there would be plenty of errors, gaps, and overlaps between our code. Thus, we frequently committed and pushed our code back to GitHub. When merging and pulling our code, we created individual branches that each of us can work on and would merge each other's branches into our own branch when we needed their code. If any merge conflicts were to exist, we would hop on call or meet up in person to resolve these conflicts.

Overall, when working on software development projects in a team environment, it is essential to communicate and make sure everyone is on the same page, document code so it is easy for others to read/debug, and learn how to effectively utilise git to work on the project together.

What would you have done differently if you had the chance to start over?

To begin, we would implement unit testing and test cases every time we made a part of the project instead of creating test cases at the end. Since we deeply tested our code towards the end of the due date, there were many bugs that we had to fix in a short amount of time. If we implemented testing at every stage in the development of the project, when we merged our code together, the amount of bugs and output errors would drastically decrease. It would also successfully indicate which part of the code is most likely to cause the error, further making the process of debugging easier.

We would also attempt to write more efficient code and try to plan out our UML so that we don't need to add or change it up while creating our program.