CS2006 Haskell 2 Practical Report

15000817

15 April 2017

1 Aim

The aim of this practical was to further develop our Haskell programming skills by implementing a game of Gomoku with an AI and GUI using various Haskell libraries such as Gloss.

2 Playing the game

To play the game you must cabal build then run in the /dist/build/gomoku directory: ./gomoku followed by the desired board size (1-19), target number of pieces in a row, and the game you wish to play (1 = Gomoku, 2 = Renju, 3 = Pente).

Pressing backspace will undo your last move. Pressing h will display a hint. Pressing r will restart the game.

3 Approach

We began by reading the specification and files closely to ensure we had a good understanding of what was expected before splitting up the tasks and attempting to complete the classes. The tasks were split up by file, meaning one person concentrated on implementing functions in each specific file. We have completed all of the basic requirements, and 9 out of the 12 extensions (all of the easy, 5/6 medium (not the time limits/pause extension), 1/3 hard (only the implementing AIs with different skill levels).

3.1 Board.hs

We began by implementing the board as this acts as the basis for the rest of the code. The makeMove function simply checks a piece is being placed on a valid square on the board (in the boundary and no piece already there), and returns the appropriate Maybe Object to be handled in the main game loop. Originally, our checkWon function checked every piece on the board for every direction to see if there were exactly the target number of pieces in a row, however this led

to a multitude of issues, as the way this worked made it impossible to check if there were too many pieces in each row. ie. Figure 1, although when the checker would check that southeast from A, there are 3 pieces in the row and return no winner, it would then check southwest from B and return that red was the winner because it counted exactly two from that point. To overcome this we realised, that although it was less efficient, it would solve our problems to check for a completely solved board by checking every plane and counting how many pieces in a row are encountered for each colour. We implemented this using many helper functions to create lists of positions and directions to check and to actually do the checking using recursion and higher order functions. The evaluate function is made up of 'steps' or 'levels' to evaluate a board, the first level being the most important. For example, the function first checks if anyone has won, which is the most important thing to check, followed by how many 'open' rows (both ends) each player has, followed by open rows (just one end), and so on, of a given size. To do this, many helper functions were implemented which are used to return the number of rows a player has that meets some condition, and the evaluate function uses these to find a score, by calling these functions twice, once for each player, and subtracting the opponent's score. List recursion, comprehension and higher order functions such as map were the main way the board was checked for these conditions. When testing Board.hs we made good use of GHCI to create instances of the board and test if moves are valid, f checkWon is true or false, and if evaluate was returning logical scores for given boards.

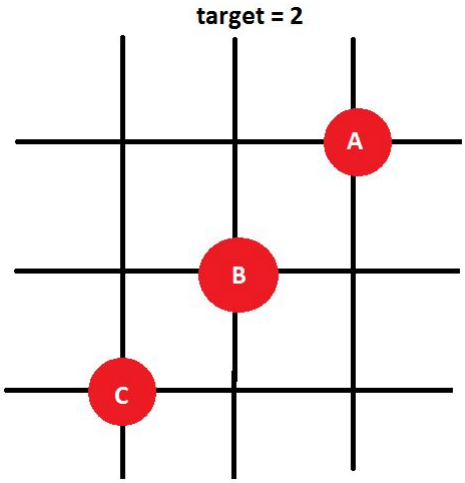

Figure 1: Sample board with a target of 2

When testing Board.hs we made good use of GHCI to create instances of the board and test if moves are valid and if checkWon is true or false.

3.2 Draw.hs

Our draw class covers drawing the board and pieces. A unique feature of our program is that the board size is exible and can be displayed all the way up to size 19. Using gloss for this class, we found that a lot of time was spent calculating how each mouse location on the window translated to the gloss coordinate. After printing the mouse location to terminal and playing around for a while, we eventually got to grips with the coordinates and used this to draw the board and pieces. We also used this information to interpret the mouse location as to which board coordinate the player intended to place a piece on. The grid lines are drawn using recursion, as the number of lines is equal to the size + 1 (to close off the grid). DisplayWinMessage checks for a winning board, and if it finds it, displays a message over the centre of the board. It also checks for a draw by checking that there are no more possible adjacent moves, in which case another appropriate message is displayed. DrawWorld uses all the helper functions to put all the gloss pictures together (the lines, the pieces, the messages and so on).

3.3 AI.hs

The AI class covers the AI within our game when the player chooses to play against the computer. The buildTree function was left untouched, as it was found on Studres. The generateMoves function generates a list of all the adjacent moves rather than all possible moves, which significantly increases the speed at which the AI plays. If there are no adjacent moves, the middle of the board is returned, as this is the best move at the start of the game (when there are no adjacent moves – a tie is handled in Draw.hs). GetBestMove is the main function which the AI uses: if the depth is 0, the next trees' boards are evaluated, and the best one's move is returned. Otherwise, evaluateTree is called with the depth, which recursively calls itself and evaluates the next trees of a given tree plus the given trees evaluation, and does the last evaluation when we reach the given depth (using the evaluate function in Board.hs). Finally, updateWorld is the function used by gloss' Play, which simply checks whose turn it is, and if it is the AI's, uses getBestMove and makeMove to place a piece on the board.

3.4 Main.hs

Our main function imports all of the required bitmap files and takes the user arguments from the command line (size of board, target, and game variant), plus options from standard input, such as if they want to play the AI and at what difficulty/colour, and if they want to load the game from a saveFile (further details on the command line and in-game options can be found in the extension

part of the report). A board size from 1 to 19 is supported by our program, and the target number of pieces in a row must be less than or equal to the board size. The play loop updates every 10ms, drawing the world in its current state, waiting to handle the user's input when it is the user's turn and doing the AI's algorithm otherwise. There are 2 helper functions – one to create a world based on the user's input, and one to start the game using the user's options.

3.5 Input.hs

Finally our Input class focuses on taking input from the user and updating the world state accordingly to what has been input into the game. It then takes this and prints a trace out onto the console which returns its second argument while printing its first, this was seen as a good design choice as it was a very useful way in which we could debug our program and, therefore, test that it works correctly.

4 Extensions

4.1 Command Line Options

The first extension that we attempted was implementing command line options. When executing the program, the user specifies the size of the board wanted, the target length of row for a game and whether or not to play against the AI, along with the location of the Savefile (where you want to store the saved game). Entering the wrong number of command line arguments will display usage instructions. This function makes use of getArgs and getLine to get the user's input.

4.2 Undo

Our list containing all of the pieces on the board appends each new move to the end of the list. Our undo function, in Board.hs, simply takes the last element of this list and let the user play a new move. This function can be called by pressing the backspace key. Since our AI would make the same move in every position, we made the design decision that the undo function undoes both player's moves if backspace is pressed. Therefore, if the AI is playing, undo removes the last 2 pieces, and if not, undo only removes 1 piece. Undo can be called multiple times in a row.

4.3 Extra Rules

We implemented the rules of three and three and four and four in the game by adding them to the Board.hs file. Since Renju makes use of these rules, we decided to only use them when Renju is being played, so they will be explained in further detail below. The rules were implemented using two functions, which make use of the other functions used by evaluate, to check the board for 'open'

rows of a certain number, and so on, returning False if the board didn't follow the rules, and True if it did.

4.4 Options

We implemented extra options for the user to select as to how they want to play the game, which they specify from standard input. If they chose to play gomoku, they are asked if they want to AI to play, what colour the AI should be, and what difficulty, or no AI at all (which they indicate by giving an integer 1-7 to select from the options). Then, in all variants of the game, they are asked if they want to Load a game from a savefile. For the game to work, a 'Savefile.txt' must ALWAYS be present in the file in which the executable is created, even if the user doesn't want to load a game. We have supplied an empty savefile which must be kept there.

4.5 Bitmap Images

In order to make out game more user friendly and nicer looking, we used bitmap images to display the images and the board. We used loadBMP to load the images from the executable folder into the world object created in the game. Our draw class then displays these images as gloss pictures. The board picture is scalable with the board size.The user could potentially change the pieces and background to whatever they wish, if they add the correct files to the dist/build/gomoku directory.

4.6 Hints

We used our AI class to implement hints for the user. When the h key is pressed (handled at Input.hs), the hint function displays using vector graphics the next move that the AI would generate for that particular colour. A depth of 0 was used to generate a move, because we decided that the hint should be very quick to display, and also shouldn't be an extremely good move (that the AI would make), just a good move. When the 'h' key is released, the hint disappears.

4.7 Saving the Game

We implemented game saving functionality to our program, so that the user can load previously unfinished games. This was done by changing the play function by gloss to a playIO function. Therefore, when the 's' key is pressed, handleInput calls the 'save' function, which saves the current state of the world to a 'Savefile.txt' where the executable for the programme is stored. That file must always exist for the game to work, even if it is empty. To load a game, simply relaunch the programme and use the in-game options to load a game. The same rules apply in the loaded game as the ones in the saved game, not the ones chosen with command-line/standard input options. For saving to work, we had to change all of the functions to return 'IO'.

4.8 Renju and Pente

These games were implemented by adding rules to the World. Renju includes the 'three and three' rule, the 'four and four' rule, and the 'overline' rule, ONLY for the black player, as black goes first. This means that the black player can't have more than 1 open row of 3, more than 1 row of 4 (open or cloesd), and any row with more than 5 pieces. If the black player breaks these rules, they lose the game. This was implemented using the helper functions mentioned above to scan the board for some conditions. In Pente, the only added rule is that of 'capturing', which was implemented using 2 functions in Board.hs, that check whether a player has a row of 2, closed off by the other player. If so, those 2 pieces disappear, and a counter on the board increases for the other player (number of captures). If a player reaches 5 captures, they win the game, which is all checked in checkWon. For these variants, we decided to not give the option of playing the AI, as we didn't have time to adjust the AI's decisions to play to the new rules. To play these variants, use the 3rd command-line argument, where 2 is for Renju and 3 is for Pente.

4.9 Further AI Options

Finally, the final extension we attempted was to implement AIs with different skills. We decided to put 3 – Easy, Medium and Hard. The player chooses which one they want to play (and the colour of the AI) using options from standard input. We decided to make the Easy AI very easy, as it simply tries to win and doesn't do much defending, and generates a best move using depth 1. Medium is similar to Easy, except it uses a depth of 2 to get the best move, so its moves are slightly better than Easy. Finally, Hard uses all the strategies in evalaute (which Easy and Medium don't), to make an optimal move, blocking off any open rows of the opponent. When playing the AI as Black and on hard, it is very challenging to beat it, and quick in making moves. The reason we didn't use a greater depth for better moves is because on large boards, a depth of 3 took too long for the AI to make a move, rendering the gameplay very slow.

5 Testing

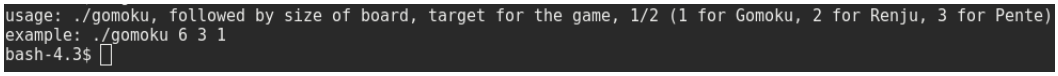

Figure 2: When the wrong number of arguments are supplied, the program provides usage instructions

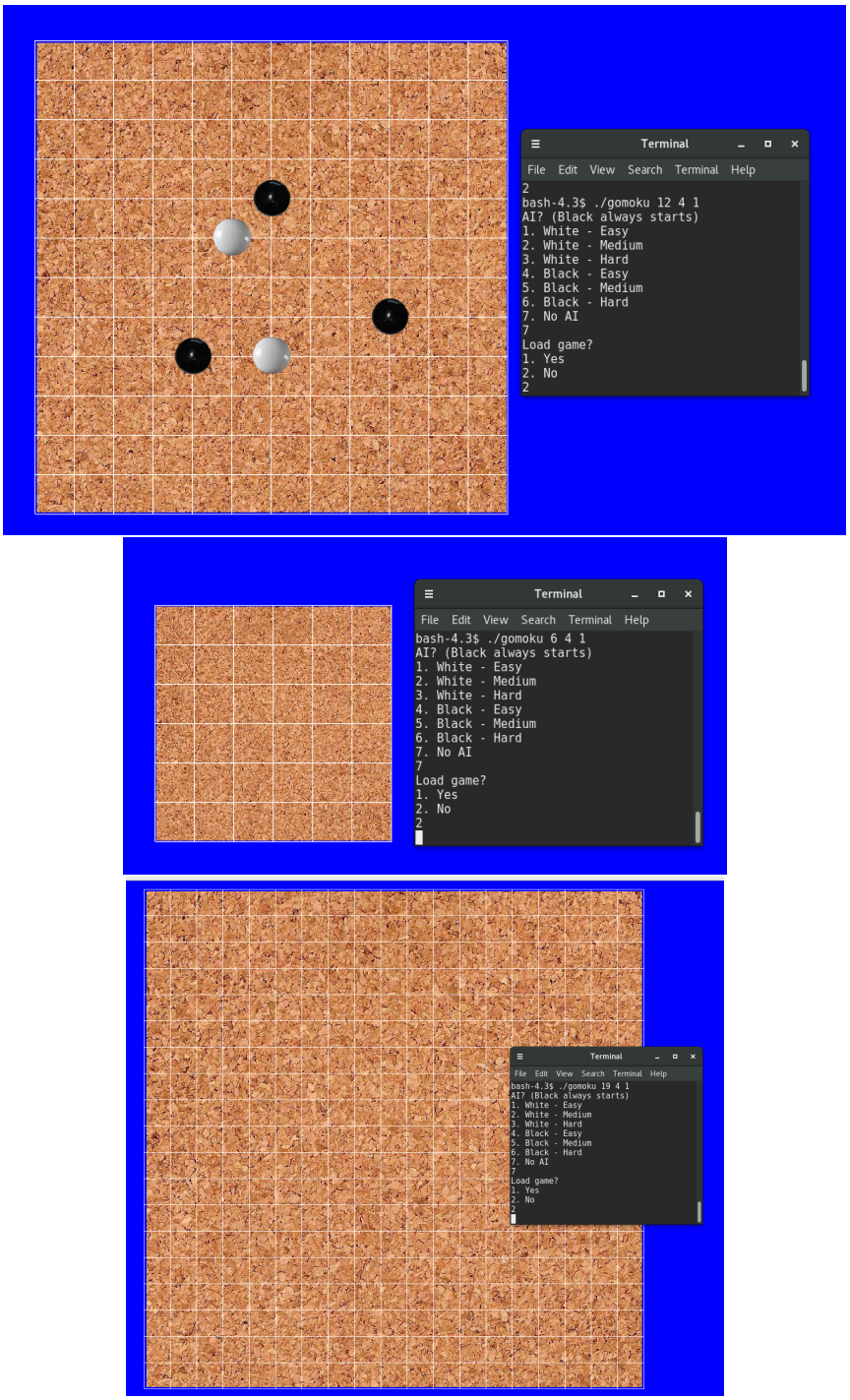

Figure 3: The board size is flexible from 1-19 and is determined from commandline arguments

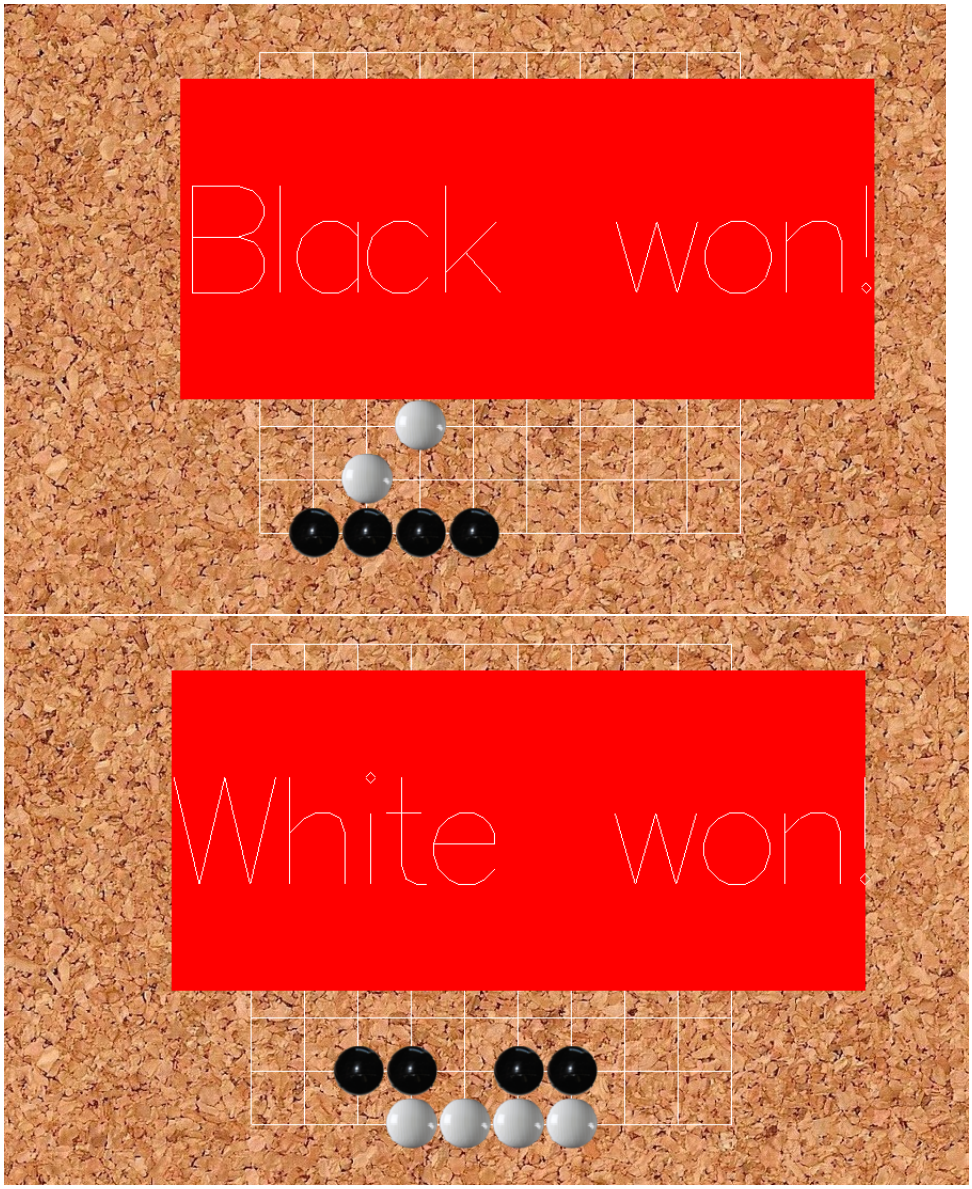

Figure 4: Both black and white can win the game, regardless of target length

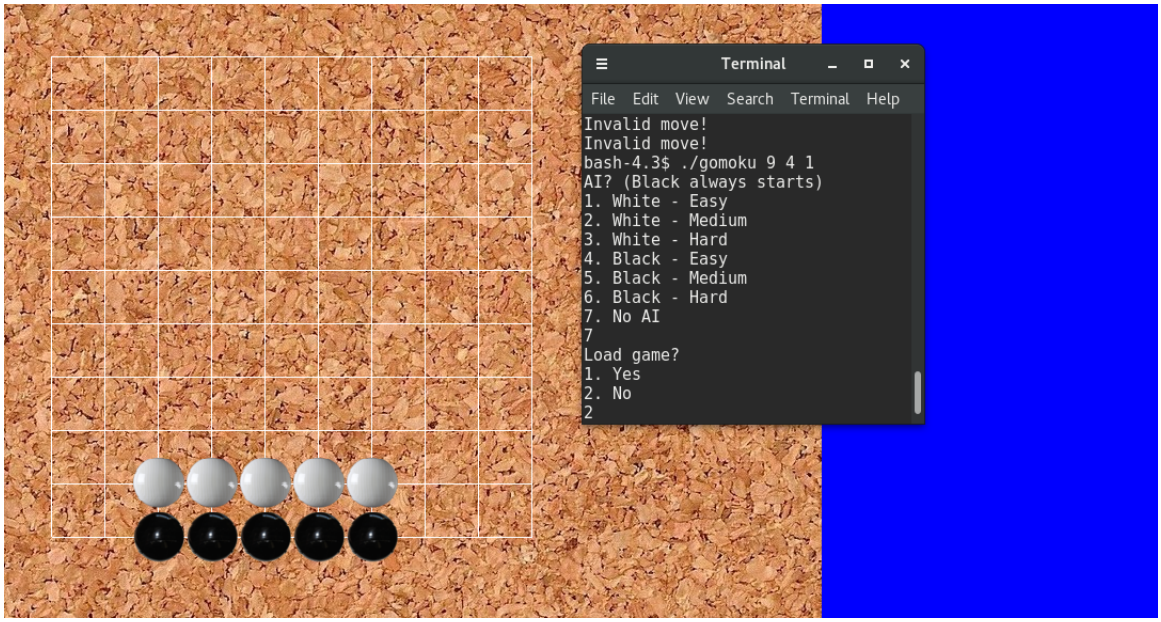

Figure 5: In the case of more than the target number of pieces in the row, nobody wins. In this case, with target 4, 5 pieces in a row means that there is still no winner and the game continues.

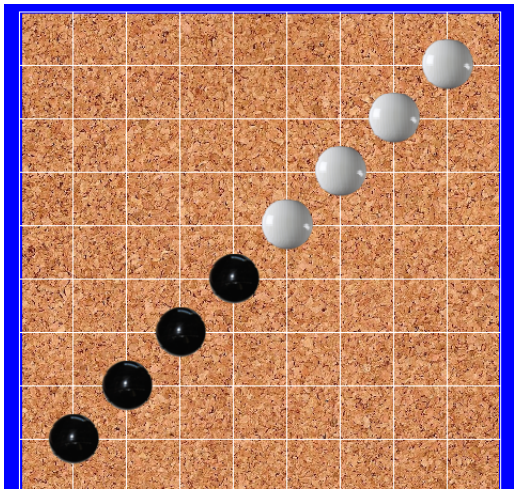

Figure 6: The easy (white here) AI tries to win rather than trying to stop the other player from winning

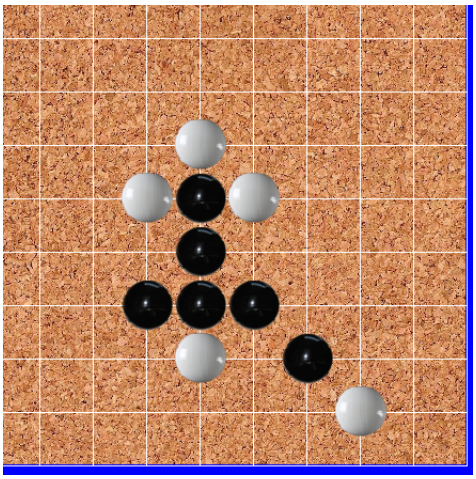

Figure 7: The hard AI (black here) tries to win by trying to get lots of rows with 2 open sides and blocking the other player, as well as trying to get long streaks in a row

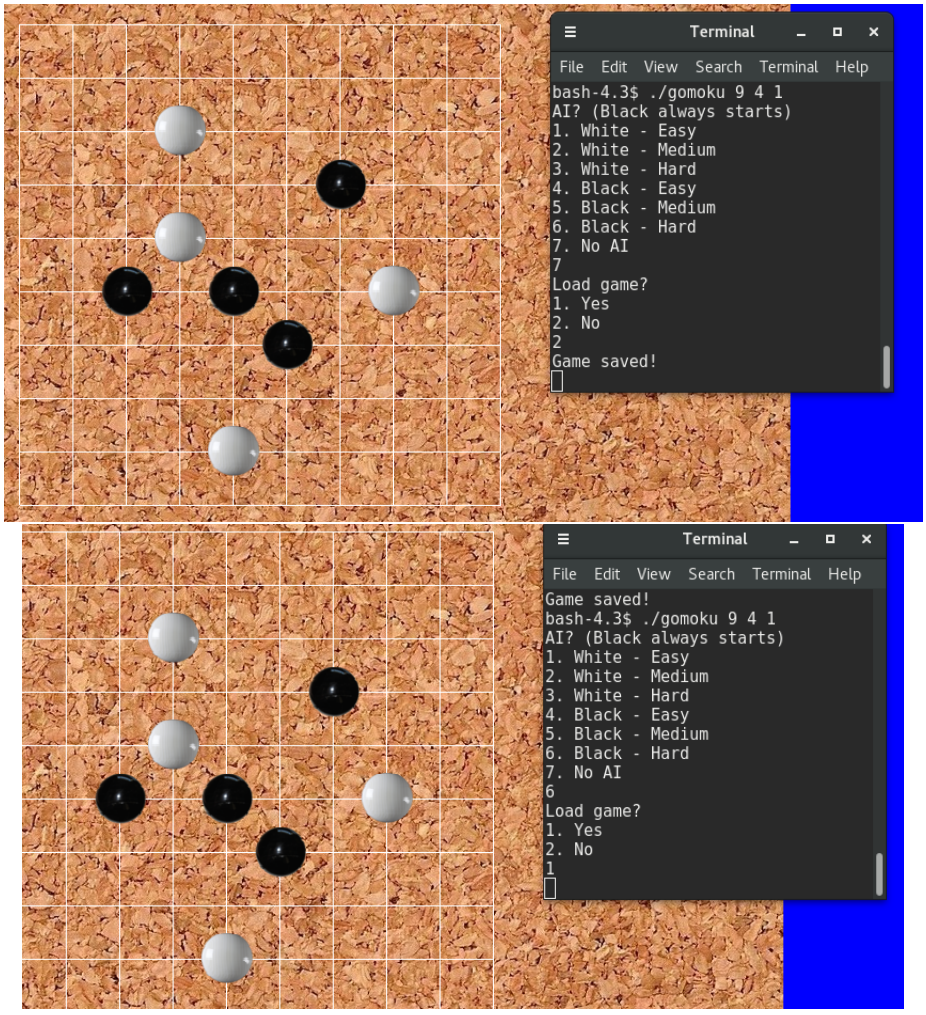

Figure 8: Loading and saving the game into the Savefile

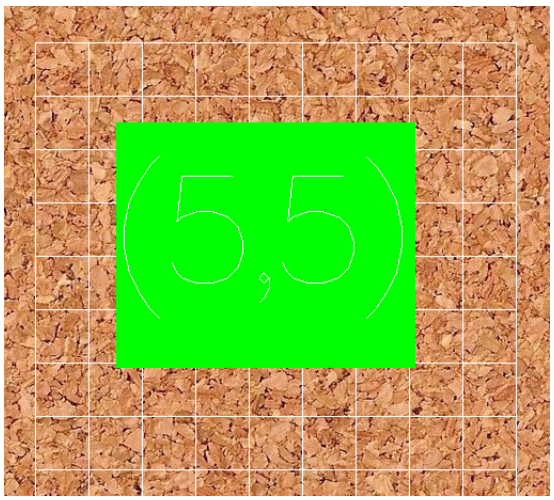

Figure 9: Pressing the "h" key displays a hint

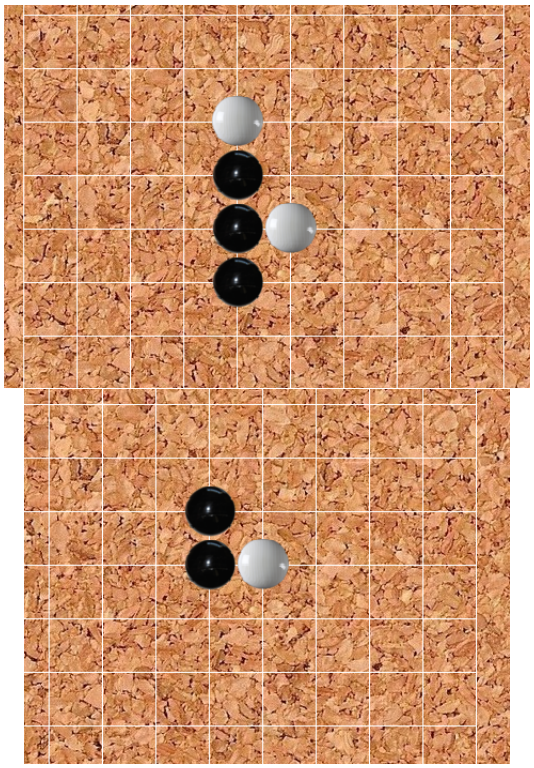

Figure 10: Before and after pressing the backspace key to undo the AI's move, taking away the last two pieces played

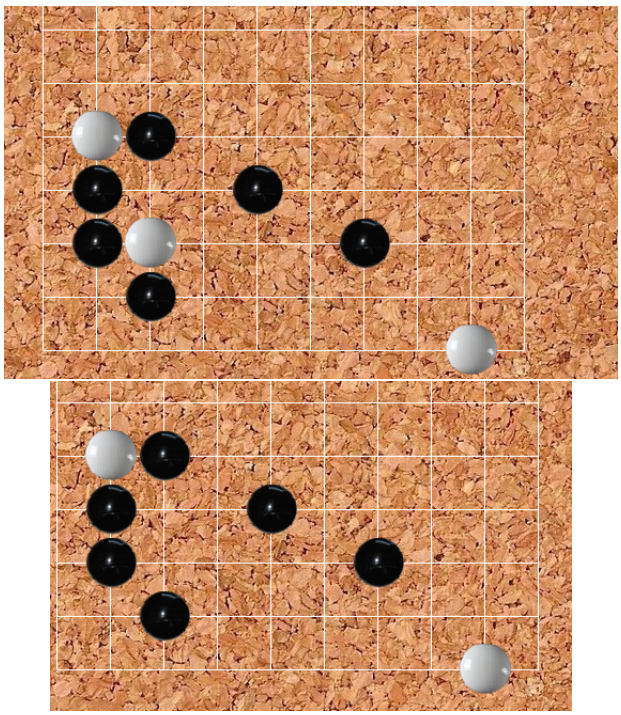

Figure 11: Our Pente extension captures pieces

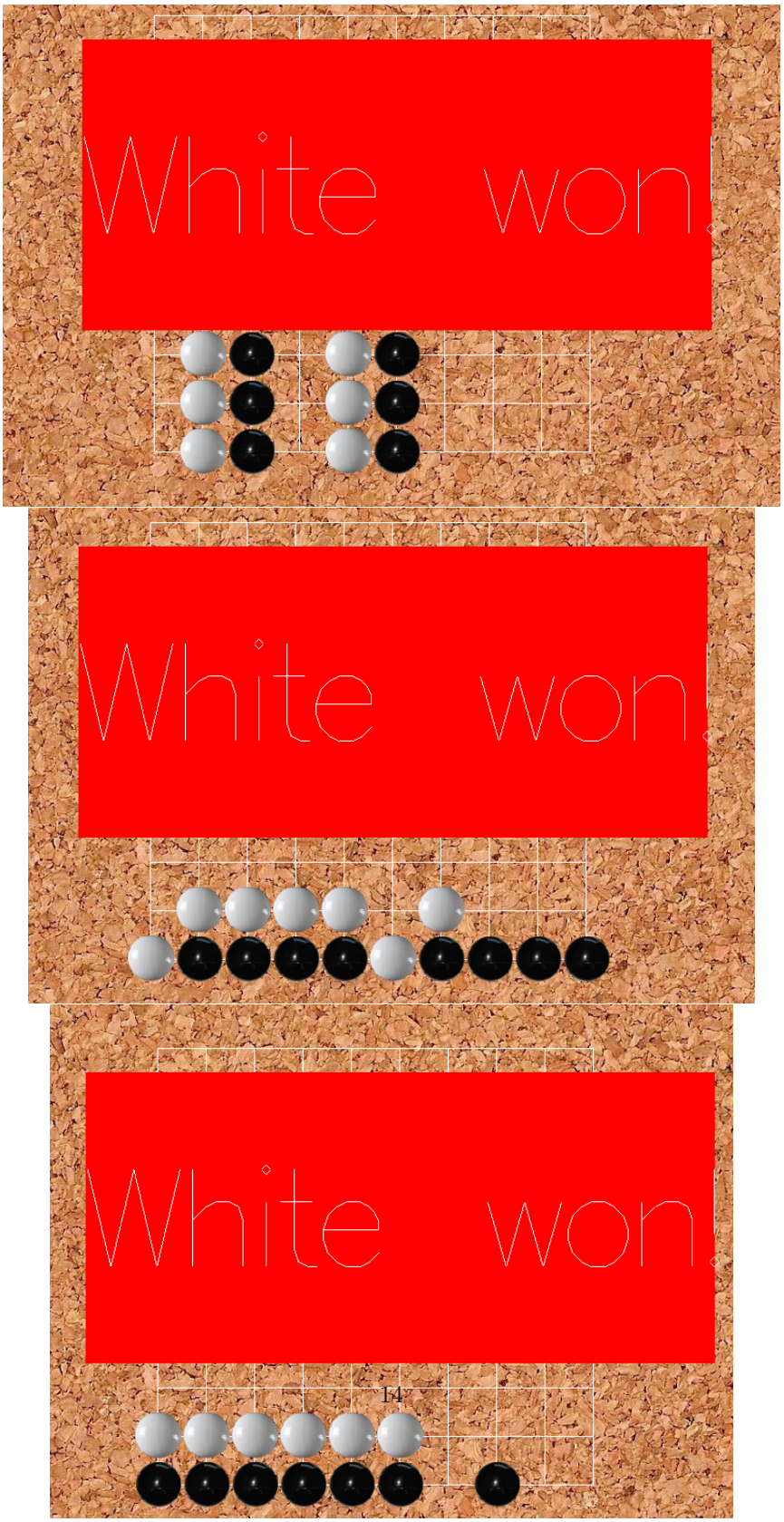

Figure 12: Our Renju extension works with the 3 and 4 rules and the override rule. The opponent will win if you play a move that is not allowed.

6 Personal Input

As we decided to split up the basic requirements, I took on the challenge to draw the initial board, which included implementing all the functions in Draw.hs. I also did the whole AI file to implement a basic AI, as well as the evaluation function on the board. Once these were done, we worked together to implement the Input file to handle input and translate coordinates. We then split up all the extensions, and I was in charge of the following: getting command-line arguments, implementing 'undo', implementing the extra rules (three and three, four and four), getting in-game options from the user from standard input, adding the functions to display the hints, implementing the loading and saving, implementing the other variants 'Renju' and 'Pente' using the extra rules, and finally implementing the AI with multiple difficulties. As we also implemented the bitmap images, the original Draw.hs file was edited.

7 Conclusion

I feel that our program meets all of the basic requirements of the practical and we have made a good attempt at the vast majority of the extensions. I am proud of how well we worked as a team and balanced multiple deadlines while still completing this project to the best of our abilities. Given more time I would have liked us to attempt more of the extensions and spent more time on improving the appearance of our GUI, as it is currently quite basic when it comes to displaying text.