CS2006 Haskell Practical 2 Gomoku Tutor: Jan de Muijinck-Hughes

120018523 130011035 130002839

Introduction

For this practical it was required to implement a board game in Haskell known as Gomoku or Five in a Row. It's a 2-players strategy board game traditionally played on a 19x19 go board. However, in modern days Gomoku is usually played on a 15x15 standard board on the intersections. It's worth mentioning that sometimes it is played on the squares rather than the intersections. The goal of the game is to complete a horizontal, vertical or diagonal line of 5 stones in a row. Black always plays first and then each player takes it in turn to place a piece of their colour on the board. No piece can be moved or removed from the board. Finally, due to the advantage that the Black player has of playing first, some restrictions on its gameplay are usually being placed.

Requirements

The minimum requirements of this practical were implemented as well as some of the given extensions of different difficulties. These are documented below:

Basic requirements implemented:

- The game mechanics of the game.
- The graphical representation of the game.
- Recognise the moves made by the player.
- A clever AI.

Before implementing the AI, we've made sure that we have a correct implementation of the game rules and an accurate graphical display of the board by having both players to be human players. The first AI implemented was a "random AI" with which further testing of the game implementation was made possible. Eventually, it was decided to implement a much more sophisticated AI by implementing the minimax algorithm and a board evaluation function. This was the hardest task of this practical and once it was completed we moved in implementing the additional requirements.

Additional requirements implemented:

- Game options can be set up by using the command line.
- An undo button to roll back the game.
- The "three and three" and "four and four" rules for the black player.
- Bitmap images for the board and the pieces.
- Displaying hints for the human players.
- A save and load feature.
- Multiple AIs of different "characters" and difficulties.

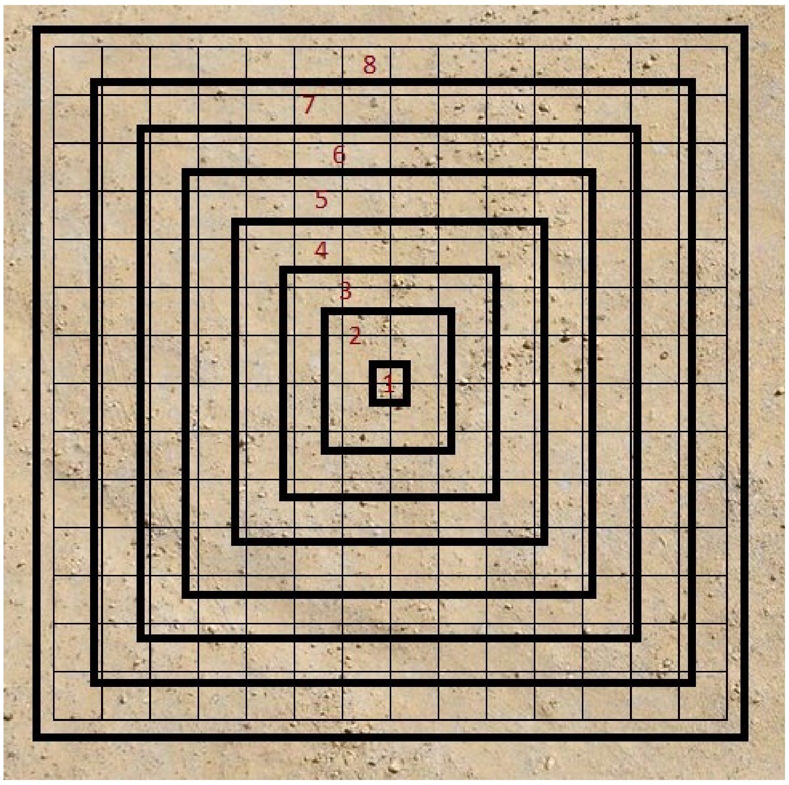
Design

The game code was separated into 5 different files, each one, containing any functions and data types that were related. This allowed the game features to be independently implemented and easily expanded without one interfering with the other.

At first a random AI was implemented with the sole purpose of testing the basic requirements. Once the testing was completed and the basic requirements were successfully implemented we moved to creating an evaluator function. First of all, an evaluator should calculate all the important characteristics a board may have so that, a score can be calculated. Hence, given a list of stones which has coordinates and colours, we needed to find if there were any 5-in-a-row pieces, 4-in-a-row, etc… Once we had that implemented, after testing and getting to know better the game, we came up with a list of features that if a board satisfied would get a given score:

1. One or more 5-in-a-row would give the maximum of 2500000 points.
2. One or more 4-in-a-row with both ends not blocked would give the maximum 2500000 points
3. One or more 4-in-a-row with one end blocked and two or more 3-in-a-row with both ends unblocked would give 1500000 points.
4. One or more 4-in-a-row with one end blocked would give 1000000 points.
5. Two or more 3-in-a-row with both ends unblocked would give 5000 points.
6. One or more 3-in-a-row with both ends unblocked and one or more 3-in-a-row with one end blocked would give 1000 points.
7. One 3-in-a-row with both ends unblocked would give 200 points plus the zone score.
8. One or more 2-in-a-row with both ends unblocked would give 100 points plus the zone score.
9. One 3-in-a-row with one end blocked would give 50 points plus the zone score.
10. Two or more 2-in-a-row with one end blocked would give 10 points plus the zone score.
11. One 2-in-a-row with both ends unblocked would give 5 plus the zone score.
12. One 2-in-a-row with one end blocked would give 3 points plus the zone score.
13. Any other case would receive just the zone score.

The zone score is a score calculated differently for each point that exists on the board. Given a board 15x15 the zones are as followed:



Any points (there is only one for odd sizes of board) found within zone 1 will get -5 score.
Any points found within zone 2 will get -10 score.
Any points found within zone 3 will get -15 score, etc…
Hence if on a board there are 3 black pieces in zone 2 and 1 piece in zone 1, will receive a zone score of -35.

The final score was calculated by evaluating the board for the pieces of the given colour minus the score calculated by evaluating the board for the pieces of the opponent colour.

The next step was to implement the minimax algorithm. The whole group worked really hard in order to firstly understand how the algorithm works and then, how to implement it in Haskell. With the completion of the minimax algorithm, the addition of extra restriction rules for the black player, were easily added. Furthermore, an AI could be selected to be of an easy, medium or hard difficulty. Moreover, different AIs were created, by overestimating, underestimating or normally considering the score calculated when evaluating the board for the enemy pieces.

The undo button works for only human players. Hence if there is a game where a human plays vs an AI, undo button will “undo” 2 moves. This is the same when a player plays a player, but both players can undo this time and two moves will be removed from the bored, to undo to the current users last move.

When the game settings are given through the command line, a validation process is carried out to validate that everything was given correctly.

Hints are being displayed using the AI already implemented. Saving is implemented using text files to save the current world state and loading is implemented by retrieving the saved world state. Lastly, a bitmap image is added as the board of the game (it’s beneath the lines drawn by the gloss) and whenever a piece is placed a bitmap image is placed in its position.

To run the program, compile the code using the ‘ghc Main.hs’ this will compile each of the files that are used in the game. Once the code has been compiled enter ‘./Main Player AI 19 5 Neutral Neutral 2’ and this will set the game to be a player vs AI game, on a size 19 board, with an aim of getting to 5, the AI being set to Neutral in both cases if each was an AI and the depth of the mini-max search to be set to 2. To change the values, enter either ‘Player’ or ‘AI’ in the first two slots, this will set player one and player two to the values entered. To change the size of the board change the value which was 19 in the example to another integer value. To change the aim change the value of five in the example to another integer value. To change the type of AI which is playing change the slots that say Neutral in the example to either ‘Neutral’, ‘Aggressive’ or ‘Defensive’. Finally to change the depth of the mini-max search change the value at 2 in the example. However, due to speed it is recommended that if you set the depth to three that you set the board size to be no larger than 6.

Implementation

The code is organised into 5 different files: Main.hs, Board.hs, Draw.hs, Input.hs and AI.hs.

- The Board file contains the data types created for the purpose of representing the board state – “world”. In addition all the functions related to the board are found here such as: “makeMove”, “checkDraw”, “checkIfWon”, “checkRules”, “generateValidMoves”, “evaluate”, etc…
- The Draw file contains all the functions related to drawing using the “graphics.gloss” external library such as “drawUndoButton”, “drawPiece”, “drawWorld”, etc…
- The Input file contains all the function that handle the input events.
- The AI file contains a data type and functions related to minima algorithm and hence the AI.

The state of the board is represented by a data type called “World” and contains another data type called “Board”, the colour of the player that is currently playing and another data type called “Options”. The “Board” data type contains the size of the board (in terms of n x n), the size of each square of the board (which is set to 15 by default), the target of the game (it’s recommended to always be 5), a list of all the pieces currently placed on the board (coordinates and colour) and a list of list of the zones (what coordinates does each zone have). Note that the zones are calculated when the game is initialised and not each time someone access the zone list, to reduce the computation time. The “Options” data type contains the nature of each player (Human or AI), the size of the board, the target of the game, the characteristics of the AI represented by black pieces (if it’s an AI), the characteristics of the AI represented by black pieces (if it’s an AI) and the difficulty of the AI (or AIs) represented by an integer value.

Locating pieces in a row was a challenge in the beginning. In the end, it was decided that each piece would check in all directions to find if it was the first of its line. Hence if it was the 2nd in a row of 4, it wouldn’t calculate anything, but if it was the first it would calculate that there is one 4-in-a-row. This was achieved by implementing various functions to move to one direction, check if there is a piece on a position, check a piece of a certain colour was one a position and many more. The “aiAnalysis” function calculates a score given a list of integers. Each integer represents something important to the game’s logic and is further explained within the code.   Many functions are also implemented to create the zones of a given board as well as return the zone of a given point. Note that because a board can have either an even or an odd size, different formulas were created for each case as zones are different. Last but not least a “generateMoves” function is implemented that is passed in the “buildTree” function of the AI.hs file. Although the logic behind this function is simple, it does increase the AI speed a lot. It only creates a list of positions close to already placed pieces. Hence given a point at (3,3) it will automatically generate the positions (2,2), (2,3), (2,4), (3,2), (3,4), (4,2), (4,3) and (4,3). Obviously, any positions generated that are not empty or are out of bounds are discarded. There is also a “generateAllMoves” function that creates a list of all valid moves currently on the board and it’s only used if the AI plays first and hence there are not pieces on the board.

The “AI.hs” file contains a data type called “GameTree” that contains a “Board” data type, the colour of the player currently playing and a list of tuples of Positions (tuple of x and y coordinates) and “GameTree” data types. This tree is created with the given buildTree function. It contains for all the valid moves a given player can play, a list of new trees for each of these valid moves. This is used through minimax when searching for the best move. Apart from the minimax functions, the “updateWorld” is where the AI is called if the world’s state indicates that it is a computer player’s turn and returns the new world state with the AI’s move.

The “Input.hs” file contains the handleInput method, which is the main way in which user input is processed in the game. When a user clicks on the game screen the handleInput will find out where was clicked, then check if the user had clicked on anything e.g. the undo button. Had the user clicked on the undo button the handle input would have processed called a function in the Board.hs file which would have undone the previous two moves by both player and AI. It also had the ability to process the save, load and hint buttons. If a button was not selected then the function would check if the click had been within the legal bounds of the board. If it was then the correct function was called to update the world, or if the click had been outside of the board then nothing would be updated and the handle input would wait for more input from the user before repeating. The main challenge for this file was that it needed to return a world, but in some of the functions like the save, an IO may be returned. This caused numerous errors as when the return type was different the function would not work properly. This was solved by adding a return statement which returned the world as it was and worked around the problem.

The “Draw.hs” file contains the main function for displaying the current game board. drawWorld is used to show the current game board. It reads in bitmap images from the ‘Main’ file and from there sets the background before going on to draw the grid over it. The function would then call on other functions to print out all of the pieces that were already on the board by reading from a list of pieces. It would also check if there was a hint currently stored in the board, and if there was it would add the hint to the display. The piecesCol would be called from the drawWorld function and would find out which players turn it was, before in turn calling placeOnCorPiece with the correct image to be printed out. This function would take in the image from the previous function along with the co-ordinates of the user’s click, then return the piece along with the exact co-ordinates so that the piece would line up on the correct square on the displayed grid. Along with these functions which were used to display the pieces and the board, there were other methods which were used to display the buttons. There were four buttons which were used in the display, these were: Undo, Save, Load and Hint. These each had a method of their own, which were all fairly similar. Each was called from the drawWorld function and would take in the image from drawWorld and from there would proceed to calculate the co-ordinates that the button should be displayed at it relation to the grid. Each button would be kept in a line at a set distance from the board no matter the size of the board. Initially there was another method which would be used to draw the circle that would be displayed on the board, but once the bitmap images were used this method became redundant and was removed.

The “Main.hs” file contains the main game loop and a number of external inputs. These inputs included the different images which were used in the game and the arguments which would be read in when the compiled code was called in the command line. The arguments which were to be read in were validated upon input and a relative helpful error message would be displayed if there was an error in the arguments. Assuming the arguments were entered correctly, the main function would load all of the bitmap images into the function and pass them into the drawWorld function when it was called. Once these had been loaded, the function would initiate the game loop by calling the initialise world function which would take in the arguments entered by the user and set the dimensions of the board, players, aim and level that the AI should be playing at in the format ‘./Main Player AI 19 5 Neutral Neutral 2’ (Player 1, Player 2, size of board, aim, AI type for player one if AI, AI type for player 2 if AI, Depth of search for mini-max function’.

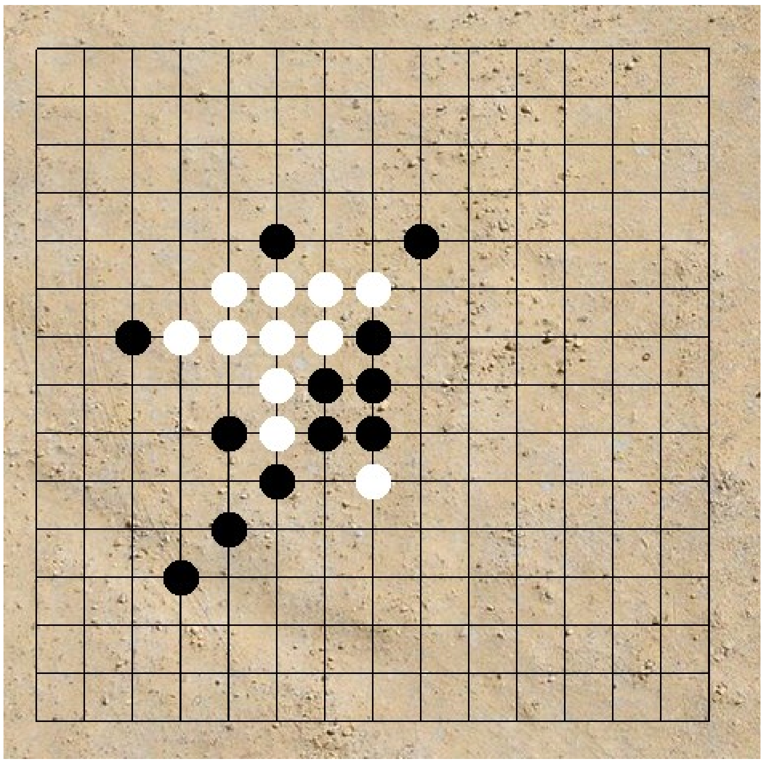
Haskell vs Python

Python vs Haskell We can see advantages of both Haskell and Python as programming languages. We can also see that they both have major disadvantages. The ability to use an object oriented model in python fits very well with creating GUIs in particular when used along with Pygame. Haskell on the other hand has a rather convoluted way of making GUIs while using Gloss. This is due to the fact that Haskell was really not designed with making GUIs in mind. Haskell provides a very fixed environment and if your code compiles it is probably working. Python is the opposite due to the fact it is interpreted. This means that you can have errors in your code that you never spot until the exact right circumstances occur and the program crashes. This makes it very easy to have bugs in your game while using Python. Seemingly simple thing like outputting to a file for saving are trivial in python but complex in Haskell. The lack of global variables in Haskell is a slight annoyance as it means you have to pass the game state between all functions. In Python you just call the game state when you need it. Overall we feel Python is a much better fit for making a simulated representation of a board game despite its draw backs such as lack of speed.

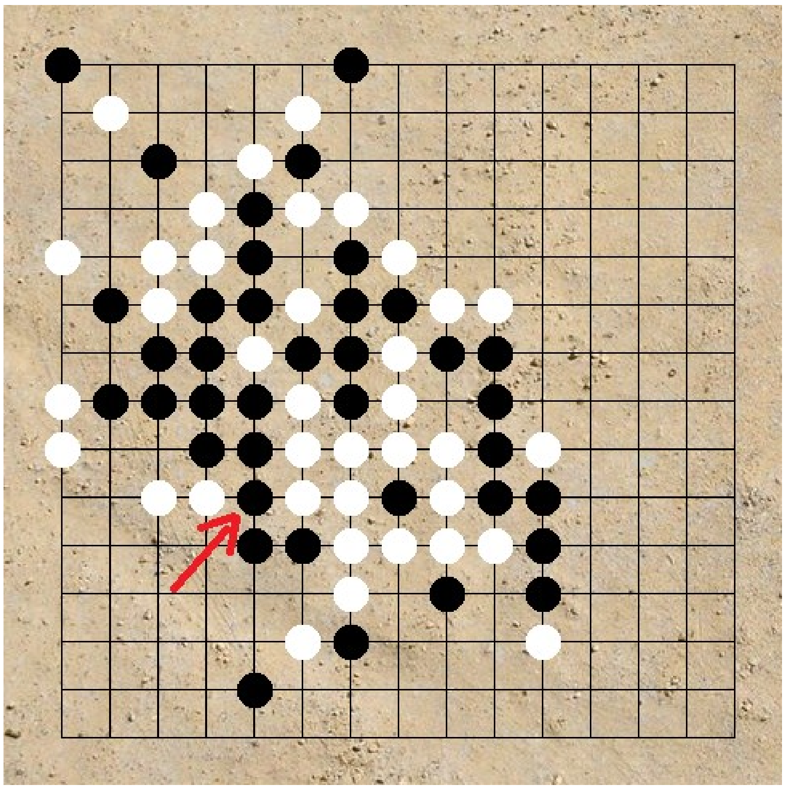
Testing

When testing the game, there were an infinite number of ways that the game could have progressed due to the nature of the AI reacting to human input which could be random or sporadic and if the AI played against a human of a similar level as they would normally cancel each other out resulting in longer games. This can be juxtaposed against a good player against one of the groups more average AI’s. That being said each of the AI’s were tested against each other while also being tested against players. The different functions of the games were also tested, such as the undo or save and load buttons. It would not have been practical to test each and every possibility so we decided to test edge cases, a few normal tests and a few exceptional circumstances.

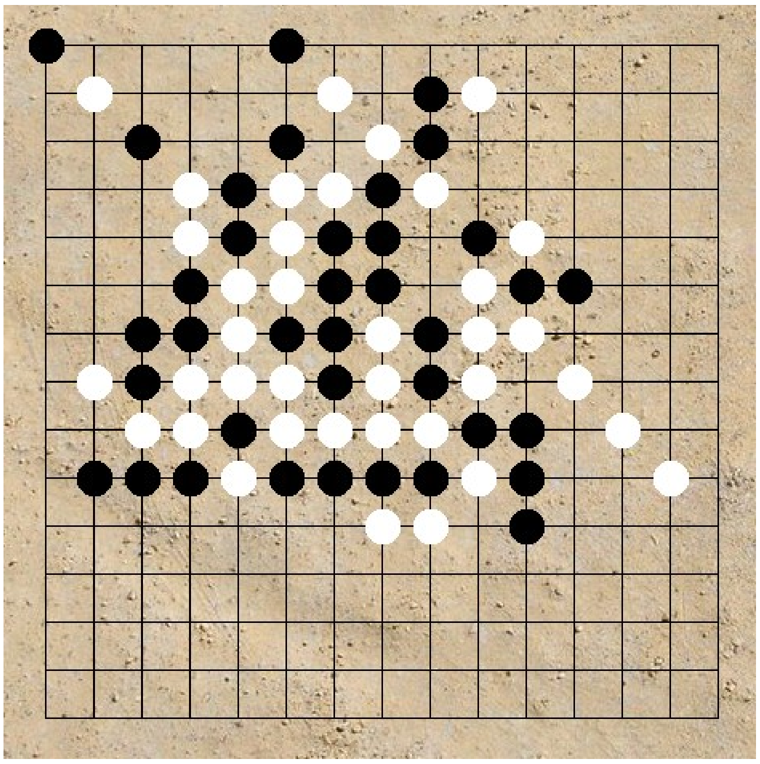
Test 1: Black (Neutral) AI vs White (Aggressive) AI. This was our test to see if the AI which would try to beat the opponent while also blocking them off from winning could beat the AI which would attempt to win and would not care as much about stopping their opponent. As the picture shows the better AI won in the neutral AI.



Test 2: Black (Defensive) vs White (Aggressive): aggressive surprisingly won by forcing the black player to play an illegal move -> four and four, as white would do a 5 in a row



Test 3: Black (Defensive) vs White (Neutral) this was testing how the AI which would try to block off before winning would fair against the AI which would try to win and block off the opponent. It was expected to be a victory for the Neutral AI and indeed it was.



After testing each of the AI against one another it was found that the defensive AI is the least likely to win and most unstable. Aggressive AI and Neutral AI both play well with different techniques. It was also found that with the exception of the defensive AI that when the AI were played against one another that they would end in a draw.

Individual Contributions:

We have completed the all the basic requirements together as a group. Once, we had the requirements done, we split the extensions between ourselves.

120018523
I have personally implemented The multiple AIs (both difficulties and characters) and included the "three-three" and "four-four" rules for the player with the Black stones.

130011035
I have personally implemented the undo and the save/load buttons. Furthermore i have implemented the ability for a player to get a hint.

130002849
I have personally allowed the the game to be set up through the command line. I have also added bitmap images for the board the pieces.

Problems:
The AI is a good AI which makes good decisions, unfortunately as more pieces populate the list the AI does slow down noticeably, but if it is kept to a 6x6 board then the AI still runs quickly keeping the same high level of decision making. So when there are fewer pieces it works quickly and well but as it gets to be more pieces it only works well. This meant that we had to limit the depth that our mini-max function worked at, to 2 in most cases or 3 on the smaller boards

Conclusion:
This practical went well overall and allowed us to explore the various libraries that allowed Haskell to build games, in this case Gloss. It was fascinating to use this language and to learn more about it's intricacies. We feel as though we completed the basic requirements and the extensions to a high level.

Provenance:
We used gloss for the graphics, we created all other code and graphics except those that were provided.