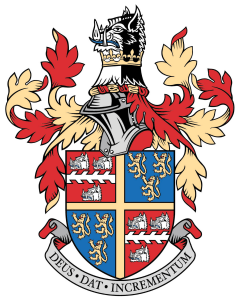


TONBRIDGE
SCHOOL

NEA Project

Gomoku and Renju

A software made for Mr Shafer

Justin Ka Chai Leung

Candidate Number: 5161
Centre Number: 61679

March 21, 2024

Contents

Chapter 1

Analysis

1.1 Introduction

1.1.1 The Client

Mr John Shafer is an English teacher in Tonbridge School. He is familiar with some Asian board games, and Gomoku is one of them. He likes the game Gomoku, but sometimes he is unable to find a board and an opponent to play with. Therefore, he asked if I could build a software for him, so he could play Gomoku on his computer.

1.1.2 Gomoku

Gomoku, or Five in a Row, is a strategy board game. The game is played on a 15×15 board, with black and white stones. The player with the black stones moves first. Players take turns to put stones of their color on the board, and the first player to put five stones of their color consecutively in a straight line wins. The line can be horizontal, vertical, or diagonal. Although this game has simple rules, the complexity of the strategies involved is rather high.

1.1.3 Renju

Gomoku begins with black, so black will have a slight advantage. Renju is designed to make it fair for both players. In Renju, there are extra move restrictions for black, namely the moves "3x3 fork", "4x4 fork", and "overline". The detailed rules are stated in the Renju website[1].

1.2 Interview

The following is a manuscript of an interview with the client, Mr John Shafer.

Question: Why do you want this software?

Mr Shafer: Gomoku is a fun game because of its simplicity. I want to bring it around so I can play with my friends, but the game is not portable because it must be played on a large 15x15 board with many black and white stones. I would also like the software version to have more additional functionalities to improve my skills.

Question: You mentioned about the simplicity of this game. Can you explain how do you play this game?

Mr Shafer: You take turns to play black or white stones on the board. The first to get five stones in a row along a vertical, horizontal, or diagonal line wins. These are the rules of Gomoku, but I also like the "Renju" variation of this game, which poses some restrictions to the black player. This makes the game fairer because Black as the first player always has some advantage. It would be great if you can also add the Renju variant in the software.

Question: Why is any of the existing software not suitable for you? How should this software to be the same or different from them?

Mr Shafer: I have used several online Gomoku websites with very strong AI players. However, I cannot learn anything from them because their moves are too complicated for me to understand. I would like to play with a AI players of similar skill level as me. On the other hand, these websites provided a simple interface for me to carry out post-game analysis. The mobile application, Gomoku Online, has a provided a variety of

[1] `https://www.renju.net/rules/`

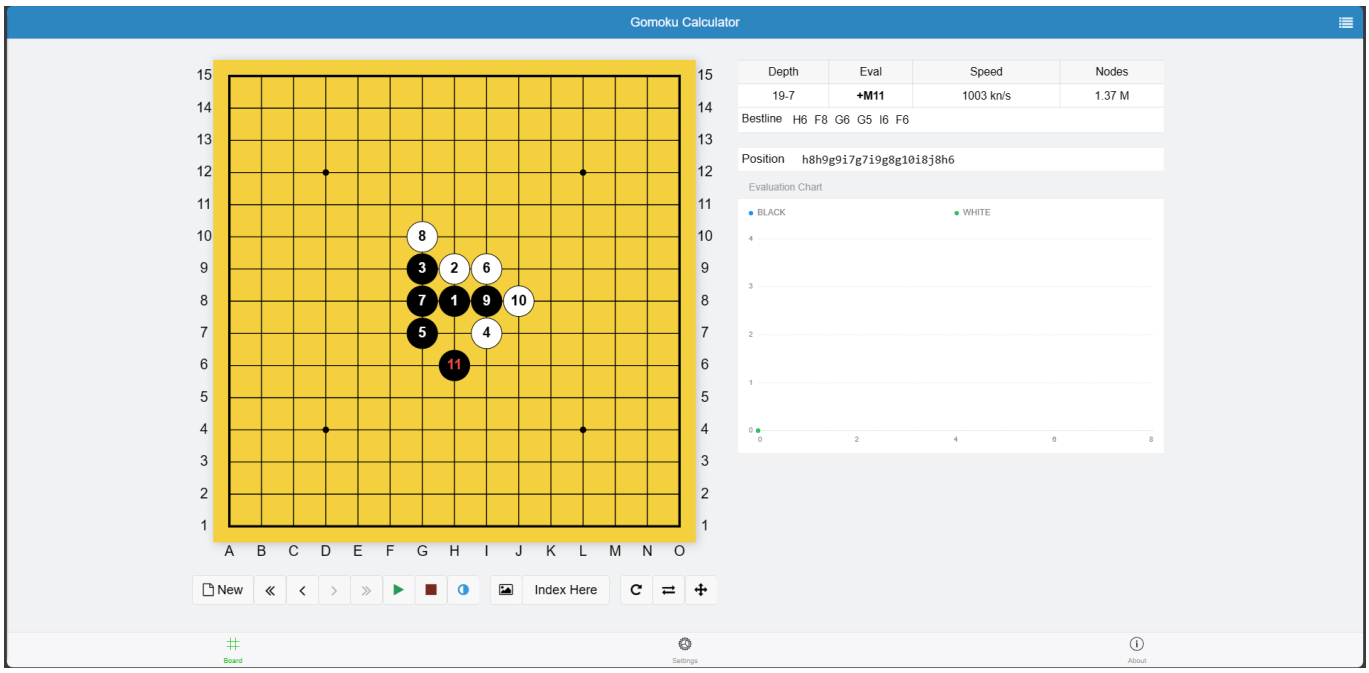


Figure 1.1: The website Gomocalc.

difficulties of AI players, but it is hard to tell whether I have improved. I would like this software to have the best of both worlds. It should provide me with many AI players and my relative skill level comparing to them, while also providing a simple interface for post-game analysis.

Question: What features do you want for post-game analysis in the software?

Mr Shafer: I would like it to tell me that if either black or white is advantageous in some board. This would be very beneficial for me to determine whether a move is good or bad. I would also like it to provide me with a possible continuation of the game. It allows me to see what would happen if I made a different move. This is essential for me to learn from my mistakes.

Question: Is there any other features which you would like to have in the software?

Mr Shafer: I would also like to save a game and load it later, so I can review games again and again.

1.3 Problem

The client wants an electronic version of the board game Gomoku. He would also like to have the Renju variant of the game implemented. He wants features that can improve his strength, with computer players of a variety of strength, a ranking system, and post-game analysis.

1.4 Past solutions

1.4.1 Websites

Multiple apps or websites which allow the users to play Gomoku with a computer player. For example, in the websites https://gomoku-ai.azurewebsites.net/ and https://gomocalc.com/#/, users can play against a powerful Gomoku computer player. Both websites have a user interface that is easy to understand and use. Gomocalc provides additional customisation settings, which the user can configure the game to play with different rulesets, including Renju.

My client, Mr Shafer, said the computer players have deep computation that can calculate forced winning sequences of more than 12 moves long. Although they were great, they were too powerful for him, and he would like to play with players that are of similar level as him. He also mentioned the lack of analysis functions. He wants to setup the board into any position, including artificial ones. He added that he liked the feature showing a possible continuation of the game. In addition, he enjoyed playing Renju, a Gomoku variant, in the website Gomocalc.

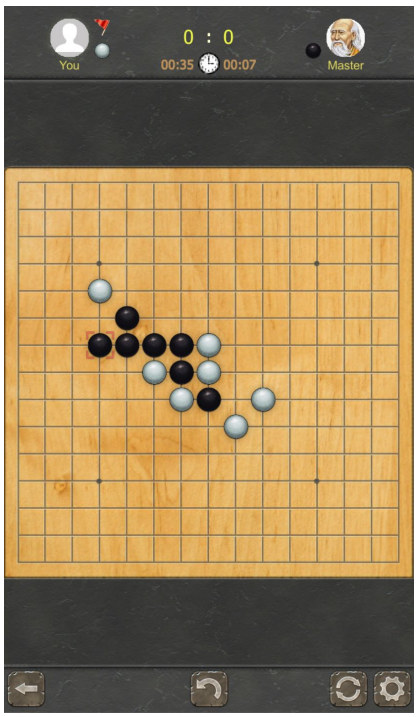


Figure 1.2: The mobile application, Gomoku Online.

1.4.2 Mobile applications

There are many implementations of this game on the mobile platform as well. A notable example is the application called Gomoku Online. It provides a variety of difficulties of computer players to the user, unlike the websites that only provide one. It also allows the user to play with other players online if they have internet access.

My client stated that he liked to play against computers of different skill levels. However, he said it would be better if the application could estimate his skill level comparing to the players, so he could understand whether he had improved and he could decide which player to challenge. He once again emphasised the importance of post-game analysis. He would like to know which side is winning, and the best continuation of some game board.

1.4.3 Conclusion

My client found the above solutions unsatisfactory. In particular, the lack of analysis functions displeased him. In addition, he thinks that multiple computer players of varying difficulties is necessary. Moreover, he would like a feature that estimates his skill level comparing to the computer players.

My client added that he enjoyed using the above solutions because of its simple user interface. He is also fascinated by the long sequence of forced winning moves that the computer is able to calculate in the Gomocalc website. Moreover, he liked the customisation features that Gomocalc provides, he wanted to play Renju. He requested the software that I will build should include all the features he desired.

1.5 Other end users

Other parties might want to add more computer players to the software, to either increase the difficulty or the variety of game strategies. They might need to edit my code and then implement a new computer player. They might implement the players in a compiled language such as C++ to improve the efficiency so as to calculate deeper moves. Therefore, I need to make sure my code can be easily understood and maintained.

1.6 Objectives

After considering the requirements from the client and potential other third parties, the following are the objectives of the software.

1. The software should have a GUI that allows easy navigation to different parts of the software.
    1.1. There is a title screen that contains buttons which navigates to the play, analysis, and rankings screen.

1.2. There are exit buttons in the play screen, analysis screen, and rankings screen to return to the title screen.
1.3. There is a sidebar with buttons to navigate to the moves of a game.

2. The player should be able to play Gomoku and Renju.
   2.1. There is a 15x15 game board with black and white stones.
   2.2. In Gomoku rules, the player should be able to place stones in any empty cell.
   2.3. In Renju rules, the player should be able to place stones in any empty cell that the rules permit.
   2.4. Able to claim victory when they placed five stones in a row.
   2.5. Transparent pieces floating on the board when the user is hovering over a cell.
   2.6. Highlighting the most recent move with the colour red.
3. The player should be able to customise the settings for each game.
   3.1. There is a configuration screen before each game is played.
   3.2. The user should be able to choose the ruleset of the game.
   3.3. The user should be able to choose what player is playing with what colour.
   3.4. The user should be able to spectate a computer vs computer game.
4. The player should be able to save a game after it is played.
   4.1. There is a button to save the game.
   4.2. The user should be able to name the players of the game.
   4.3. The game should be saved in a file.
5. The player should be able to analyse a game after it is played.
   5.1. The user should be able to load a saved game with a button.
   5.2. There are buttons that after pressing, each move places black or white pieces on the board.
   5.3. There is a button that after pressing, each move alternates the colour of the pieces on the board.
   5.4. The user should be able to see which side is winning, analysed by a computer player of the user's choice.
   5.5. The user should be able to see the best continuation of some game state, analysed by a computer player of the user's choice.
6. There should be a variety of computer players.
   6.1. There should be at least 10 computer players of different skill levels.
   6.2. The player should be able to play with any computer player.
   6.3. The computer players should be able to play with each other and themselves.
7. The software should have a ranking system.
   7.1. There should be a rankings screen that shows the rank of each player.
   7.2. The user should also be able to see the skill level of each player.
   7.3. The skill levels of players should be updated after each game.

1.7 Research

1.7.1 Threat-space search

The threat-space search (TSS) algorithm searches for threats and tries to find a threat sequence to win the game [1]. A threat is a move that must be responded by the opponent or else they will lose the game. TSS ignores defensive moves and focus on offensive moves entirely. The Victoria program in [1] plays moves from its database, and when it is out of its database, it uses the TSS algorithm. It won the 1992 Computer Olympiad for Gomoku.

1.7.2 Minimax with alpha-beta pruning

The minimax algorithm with alpha-beta pruning was used for a bot to play Gomoku [2]. In their paper, they used the evaluation function:

$$\begin{aligned} Eval = & w_1 \times \#\text{five-in-row} + w_2 \times \#\text{straight-four} \\ & + w_3 \times \#\text{four-in-row} + w_4 \times \#\text{three-in-row} \\ & + w_5 \times \#\text{broken-three} + w_6 \times \#\text{two-in-row} + w_7 \times \#\text{single marks} \end{aligned} \tag{1.1}$$

They also used the following optimisation methods to reduce search space:

- Eliminate actions of no real value
- Sorted successor list
- Local beam search

These methods are used improve the efficiency of minimax.

1.8 Proposed Solution

1.8.1 Language

I am fluent in C++, Python and Rust. To choose between them, I considered the pros and cons of both languages, and how can the features benefit the development of this project.

C++	
Pros	Cons
Lower execution time due to compilation Has object-oriented programming features Gives low-level control	Hard to use online libraries Longer debug times Unsafe due to lack of garbage collection

Python	
Pros	Cons
Easy memory management due to garbage collection Has many public packages and a package manager High readability	Low efficiency since it is interpreted High memory usage

Rust	
Pros	Cons
High security since memory safety is provided High performance since it is compiled	Longer development times due to complex syntax and concepts Does not strictly support object-oriented Rust is still new, it has fewer packages and libraries

In terms of development time, I think it is important to minimise the development time. Knowing what the client wants is crucial to this project, and usually these meetings are time-consuming. Moreover, the client would like many features, so time must be carefully used to implement all the required features.

In terms of custom libraries, using them will allow me to write a lot less code, hence improving the development time. A convenient GUI library is essential for my project, since the client demands a simple GUI.

In terms of time and memory usage, they are not as important to me since computational heavy calculations are rarely used in this project, it will only be used when the computer player is finding the best move.

In terms of object-oriented programming, I would like to use a language with such feature because it will be beneficial to organise my code in objects in this large project.

Considering these four aspects, I chose Python as my main programming language since it has short development times due to its high readability, the abundance of public packages, and the features of object-oriented programming.

1.8.2 IDE

The three most popular Python IDEs are IDLE, Pycharm, and Visual Studio Code (VS Code). The pros and cons of these IDEs are considered in the following tables.

IDLE	
Pros	Cons
Easy to get started Free	Not suitable for large-scaled projects Fewer customisation functions

Pycharm	
Pros	Cons
Specialises in Python and the development of large-scale projects Reliable debugger	The premium version requires a subscription Less customisable

VS Code	
Pros	Cons
Free and lightweight High customisability including Git integration Has powerful features like IntelliSense code completion and code refactoring	Hard to setup

In terms of the cost, I would prefer to minimise the cost as the project is done without any budget. Therefore, IDLE and VS Code would be more preferable.

In terms of customisability, it will be very useful it can drastically reduce the development time with powerful functions like code completion, or the use of keyboard shortcuts. VS Code is the only IDE that provides such features.

In terms of debugging, a powerful debugger can reduce development time as well. Instead of manually tracing variables, I can instead rely on the debuggers. VS Code can be customised to provide a debugger, and the debugger provided by Pycharm is also preferred.

In terms of the time used for setting up an IDE, it is not a major concern in my opinion since the time used will be negligible comparing to the long development time.

In conclusion, considering these four aspects, I chose to use VS Code due to its cost, customisability, and the Python debugger. Moreover, I am familiar to VS Code so I am more inclined towards it.

1.8.3 Modelling

A game of Gomoku/Renju consists of moves and game states. These features resemble a graph, in which nodes are game states and edges are moves. In fact, the graph is a rooted tree as each game state only have one previous move, i.e., only one parent. Game states consists of the board, and the current player to make a move. Refer to Figure 1.3 for a diagram.

1.8.4 Programming Paradigms

Object-Oriented Programming

Object-Oriented Programming (OOP) is beneficial to code organisation for objects (classes) in large projects. It is usually regarded as good programming principles and it gives many benefits such as modularity, polymorphism, and code reusability. Most importantly, it can greatly reduce development time.

Observer Pattern

By using the observer pattern, objects in my code can notify its dependents (observers) if there are any changes in their state, by calling the methods of its observers. In particular, it can address one-to-many relationships in my objects and is generally regarded as great for notifications among the GUI elements.

1.9 Software Development Plan

Since only 12 weeks of development time are available for me, proper planning is needed to ensure that the project can be finished on time. Therefore, I have created a Gantt chart to plan the schedule of the project, as shown in Figure 1.4.

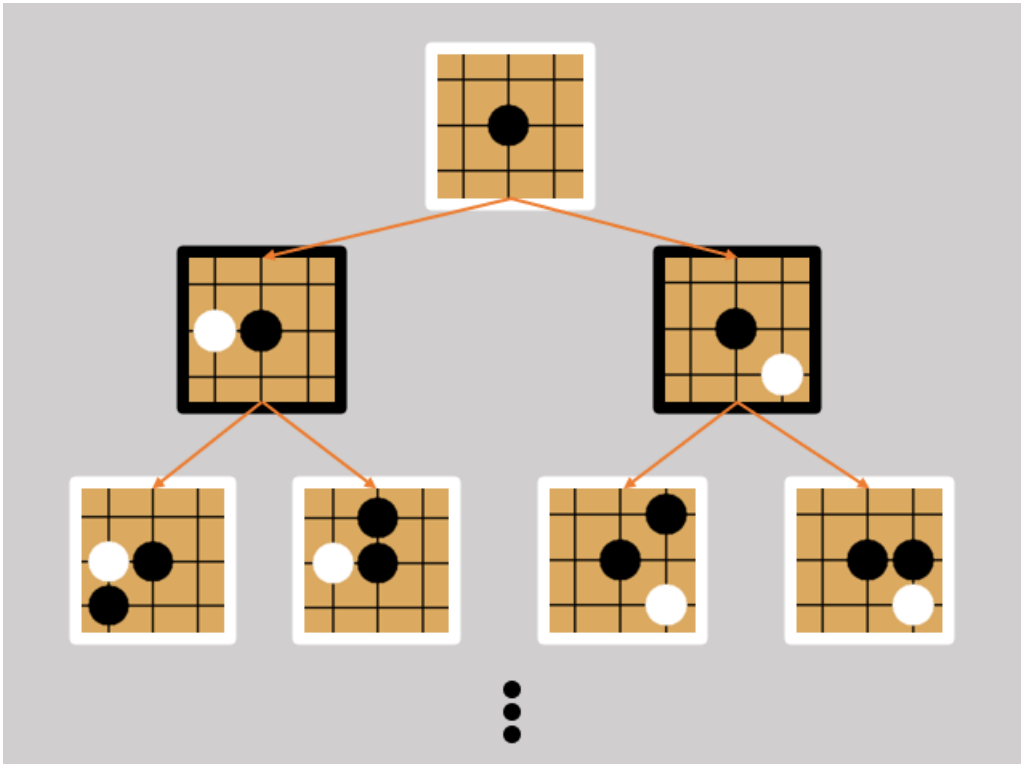


Figure 1.3: The game tree. The color of the outline of the boards represent the current player.

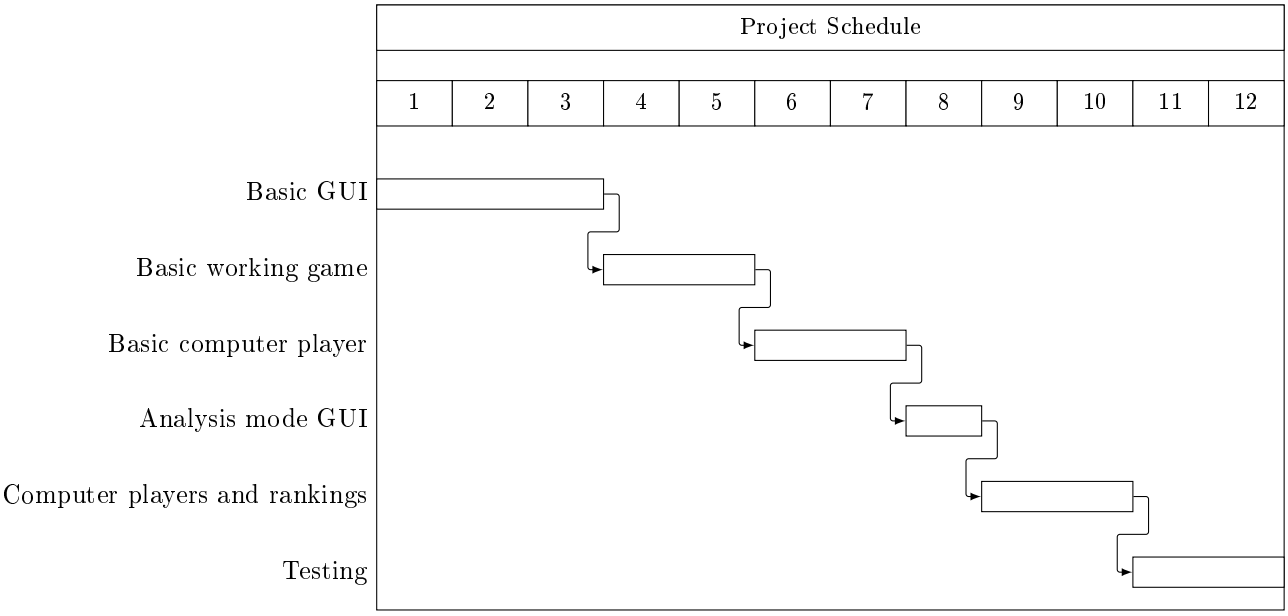


Figure 1.4: Gantt chart for the project schedule.

Chapter 2

Design

2.1 Overview

2.1.1 Top-level

In this subsection, I will describe the top-level overview of how the system works in my program. By abstracting the objects in the system, it allows me to hide all the complexity of the details of the objects and instead focus on the relevant information, such as, the relationships between the objects. Figure 2.1 shows the relationships of the windows in the software.

2.1.2 Title Screen

According to Objective 1.1, the title screen should have buttons that allows the user to navigate to various areas of the software. They include:

- **Play**: This button first asks the user for configuration settings of the game. Then, it navigates the user to the play screen.
- **Analysis**: This button navigates the user to the analysis screen.
- **Rankings**: This button navigates the user to the rankings screen.

A design of the GUI is shown in Figure 2.2.

When the **Play** button is pressed, the software should first ask the user to configure the game, according to the requirements of objective 3. They should be able to:

- Choose to play either Gomoku or Renju.
- Choose who the black player is.
- Choose who the white player is.
- If both players are computer players, the user should be able to choose how many times the game should be played, and the software should then play the game that many times and update the rankings accordingly.

A design of the GUI is shown in Figure 2.3.

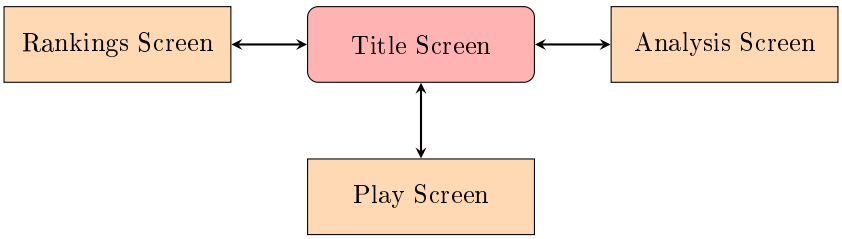


Figure 2.1: Top-level Overview

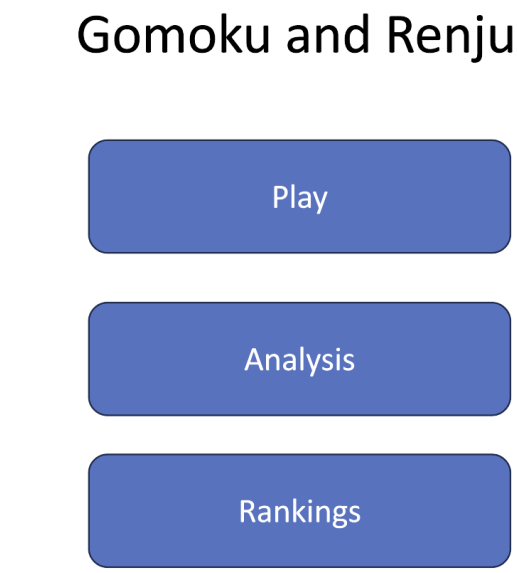


Figure 2.2: The GUI of title screen.

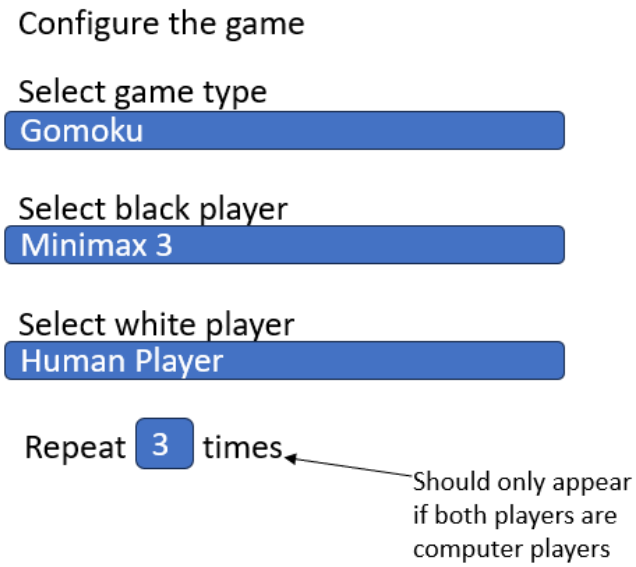


Figure 2.3: The GUI of configuration screen.

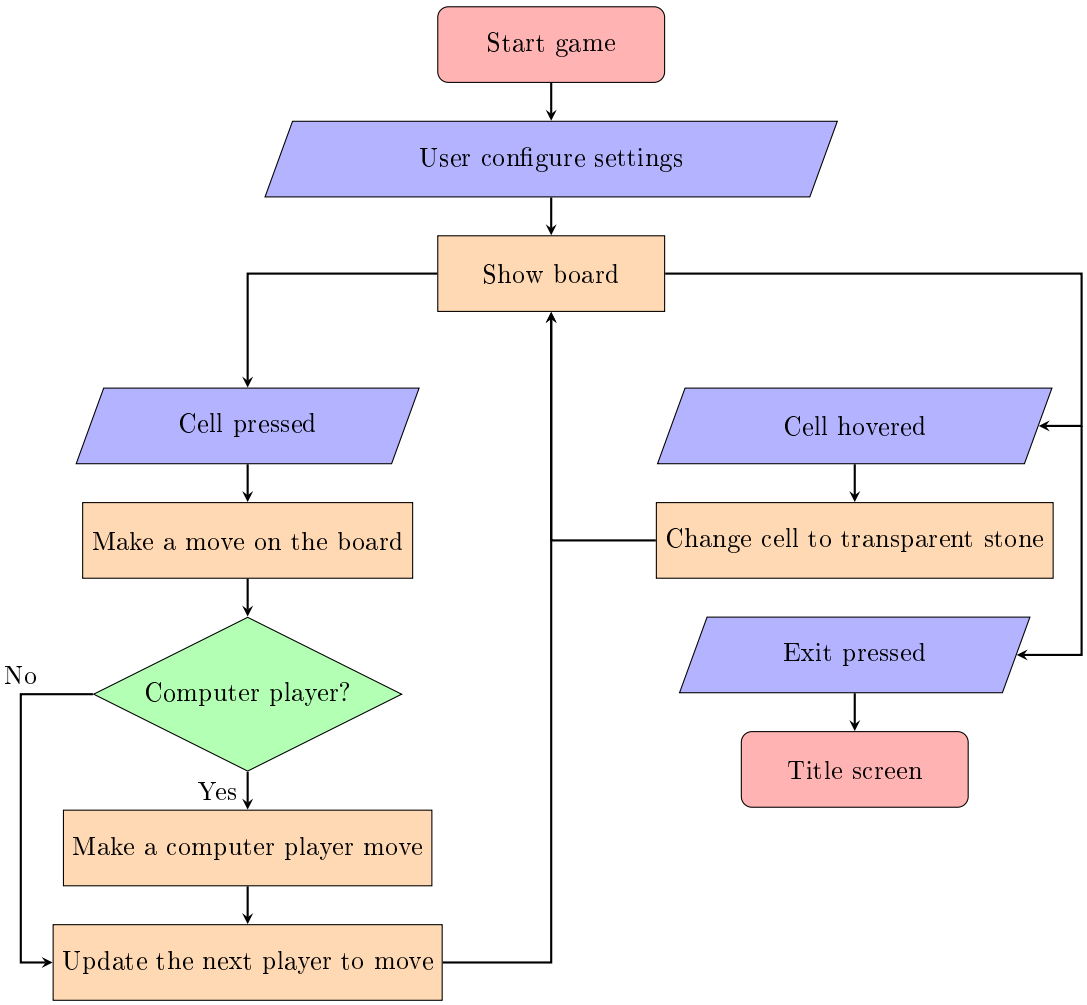


Figure 2.4: Play screen

2.1.3 Play Screen

In the play screen, we wish to achieve objective 2. When the play screen appears (by pressing the **Play** button in the configuration screen), it should show the game board to the player. After that, it should respond to user inputs, including when a cell is hovered, pressed, or when the exit button is pressed. Objective 2.5 is satisfied by showing transparent pieces on the board when a cell is being hovered. A logic flowchart is shown in Figure 2.4.

The GUI is shown in Figure 2.5

2.1.4 Analysis screen

In the analysis screen, the functions in the play screen should be present, with the additional functions of import game, export game, getting the evaluation of the board, and getting the best move of the board.

It should also include three additional buttons to assist setting up the board, according to objective 5.2-3. These three buttons are named as the black, white, and black/white buttons. After the black button is pressed, every cell pressed by the user will be changed to a black stone. The same happens for after the white button is pressed. After the black/white button is pressed, the user now places black and white stones alternatively. Its function is also described in Figure 2.6.

The GUI is shown in Figure 2.7.

2.1.5 Sidebar

The sidebar is present in both the play screen and the analysis screen. It allows the user to navigate to all the moves of the game. Its GUI can be seen in both Figure 2.5 and Figure 2.7. The logic of the buttons in the sidebar is shown in Figure 2.8.

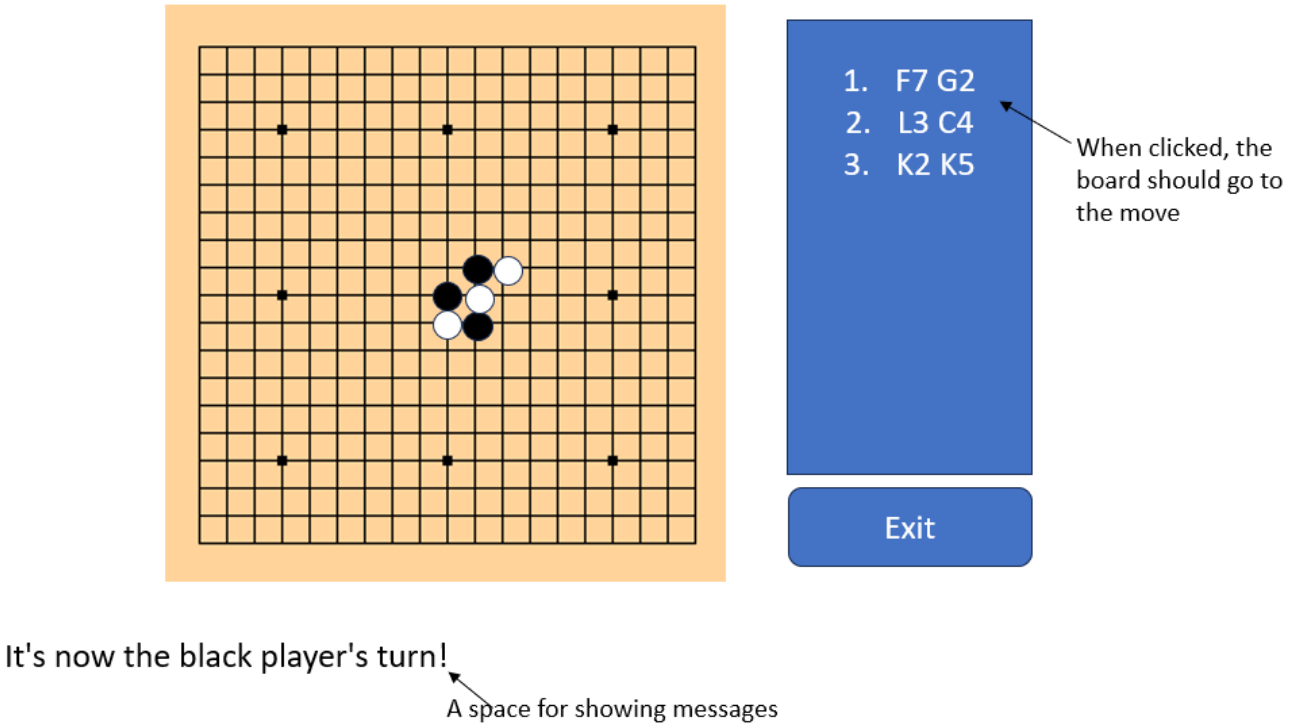


Figure 2.5: The GUI of play screen.

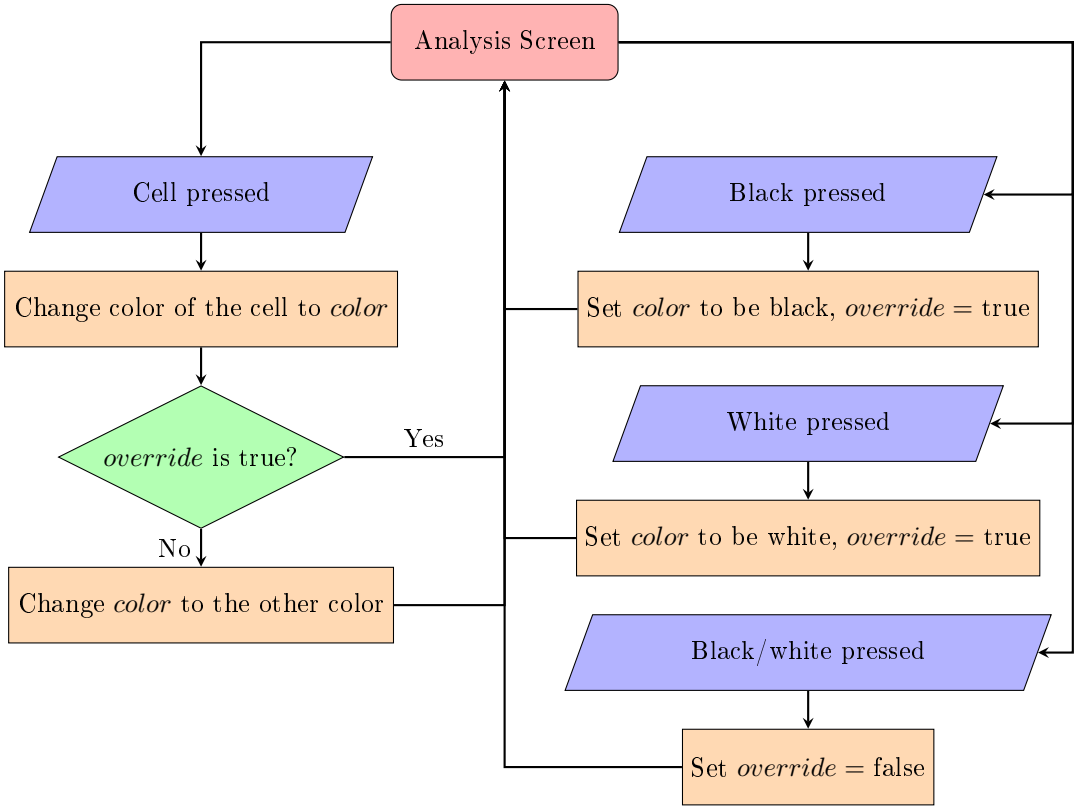


Figure 2.6: The three buttons in analysis screen.

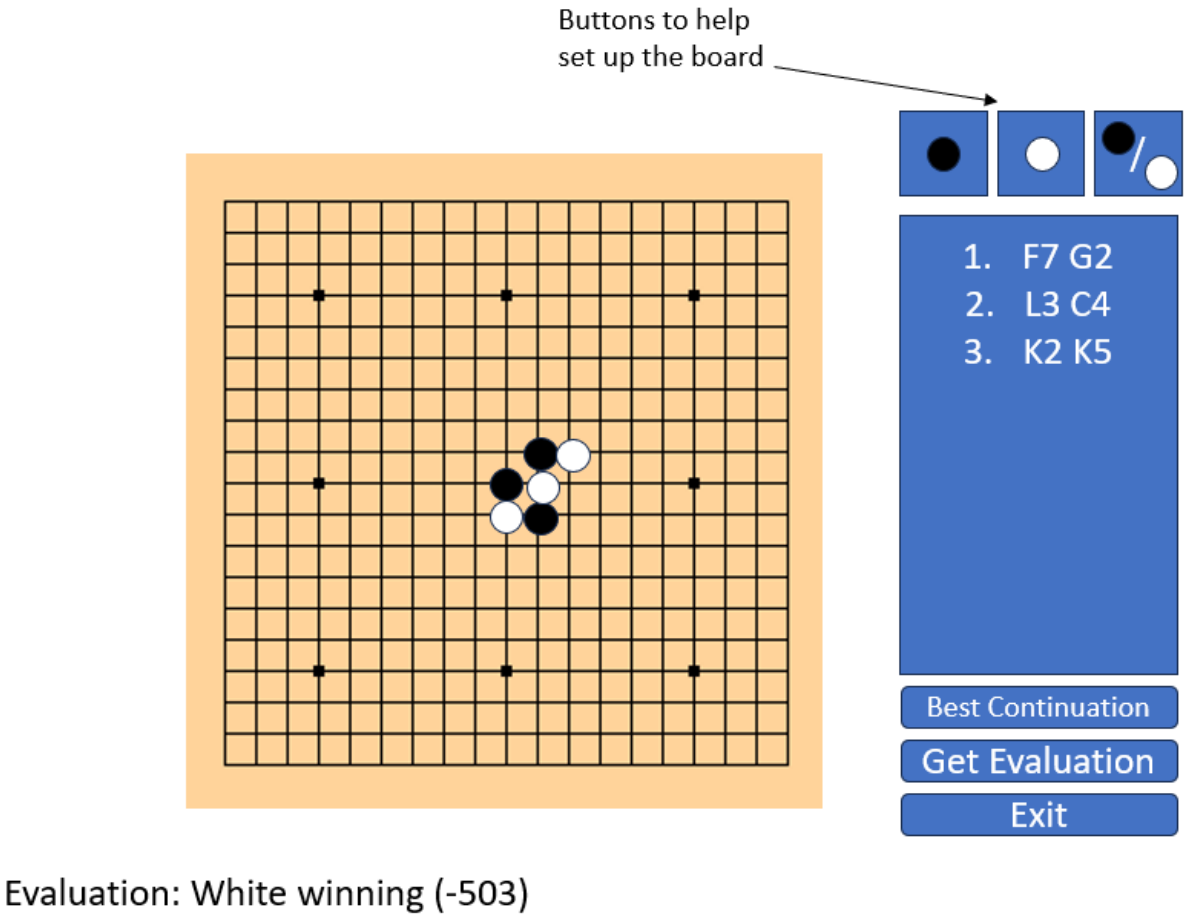


Figure 2.7: The GUI of analysis screen.

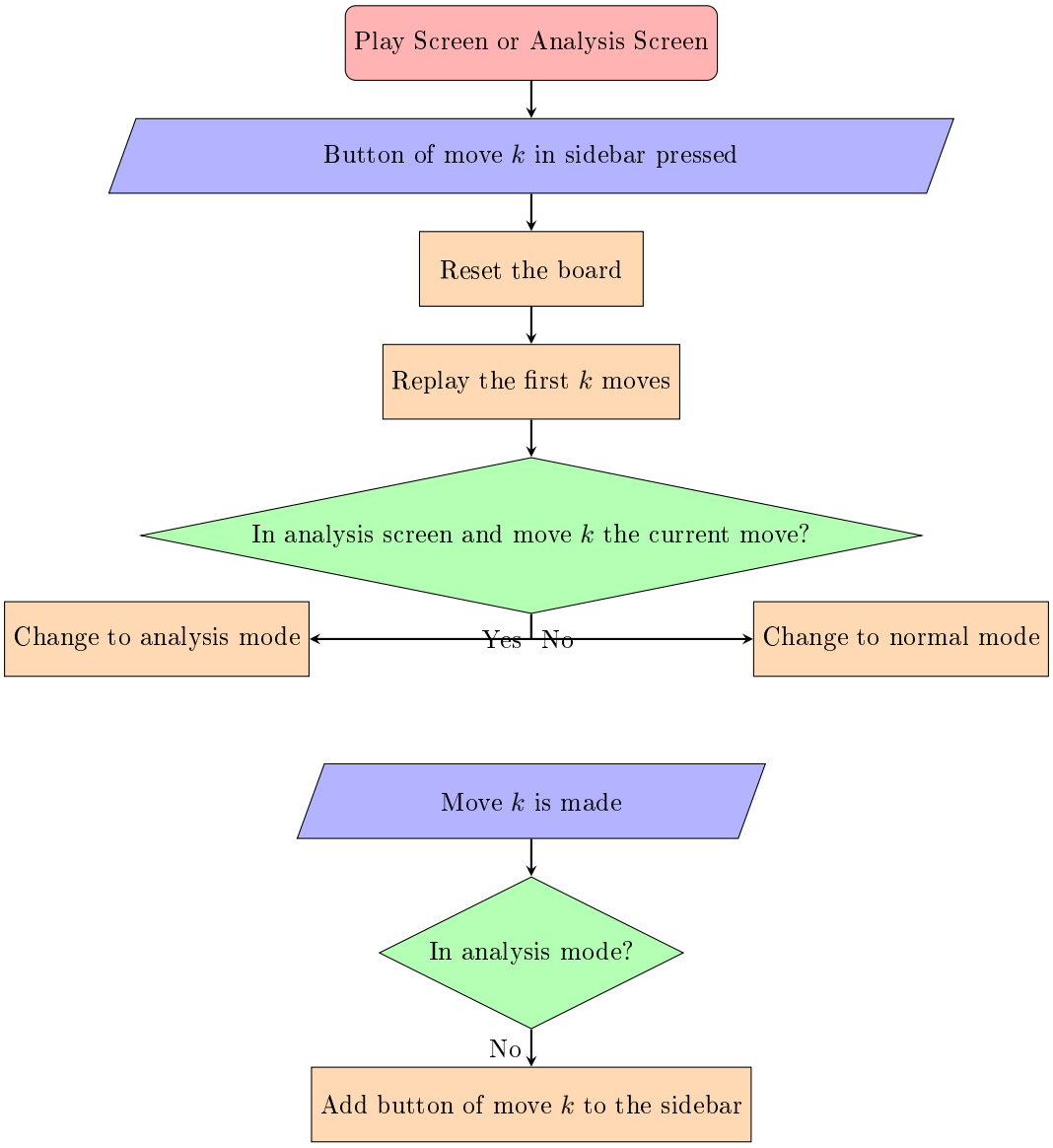


Figure 2.8: Logic of the buttons in the sidebar.

Rank	Name	Elo Rating
1	Minimax 3	1601
2	Human Player	1555
3	Aggressive 2	1539
4	Defensive 4	1438
5	Prioritise Threats 2	1429
6	Random Moves	1328

Exit

Figure 2.9: The GUI of rankings screen.

2.1.6 Rankings screen

In the ranking screen, objective 7 should be met. There should be a table of the rankings of the human player and computer players, and it should be shown sorted by their rating in descending order. The GUI is shown in Figure 2.9. The flowchart for showing the rankings is shown in Figure 2.10.

2.2 Algorithms

2.2.1 Board pattern recognition

According to objective 2, the rules of Gomoku and Renju should be implemented. Therefore, the whole board needs to be checked at every move to disallow illegal moves and to check for winning patterns.

For example, we would like to check for five-in-a-rows, straight fours, and threes. According to the official Renju rules, a straight four is a row with four stones to which you can add one more stone to attain five in a row, and a three is a row with three stones to which you, without at the same time a five in a row is made, can add one more stone to attain a straight four. As these patterns are frequently checked to apply the Renju rules and also by my board evaluation function (in Minimax), these patterns must be checked swiftly. In particular, the questions of "How many given patterns of pieces of a given colour are there? What about the patterns that must involve a given cell?" will be frequently asked. We start with a brute-force approach.

Brute-force search

Here we use a Cartesian grid system to store the state of the board as it is convenient. We can do a brute force search to find fives on the board by Algorithm 1.

In the algorithm I assumed the function *get_color*(), which gets the color in the specified *position* from the board. If the board size is 15x15, this algorithm will call *get_color* at most $15 \times 15 \times 5 \times 4 = 4500$ times.

This method contains some inefficiencies as there are some repeated calls to *get_color*() with the same arguments, especially in the for loop of k from 0 to 4.

Partial sum

Partial sum is a common technique used to optimise the counting of elements in an array. It can count the number of elements in any interval in an array in constant time. It first constructs an array P, where $P[i]$ is the number of elements before i. It can then calculate the number of elements within an interval $[l, r)$ using $P[r] - P[l]$, in constant time.

In the for loop of k from 0 to 4, we aim to count the number of cells of color *color* starting *position* going in the specified *direction*. Here we can use partial sum to optimise the counting.

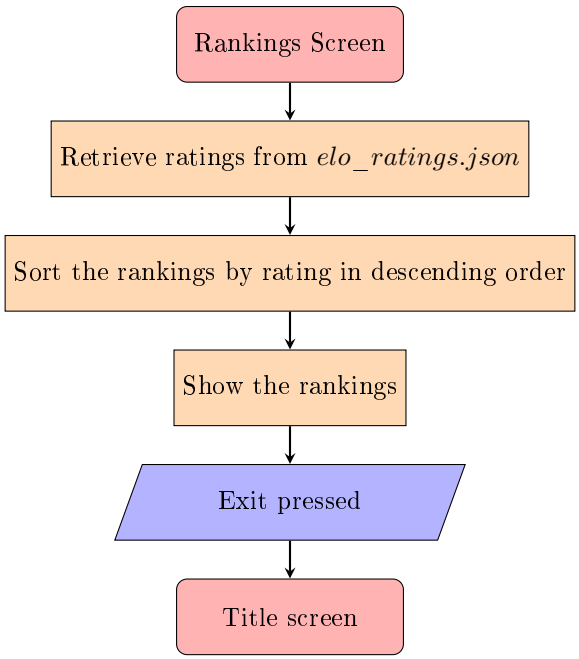
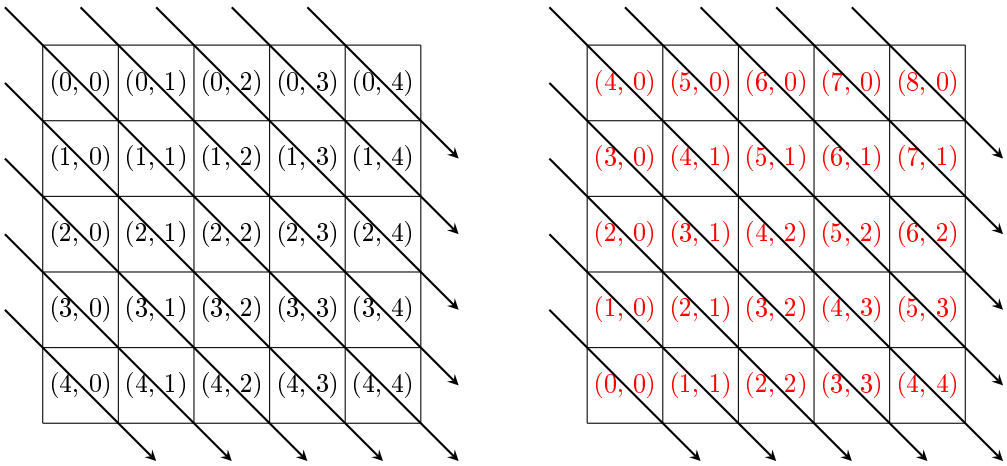


Figure 2.10: Rankings screen

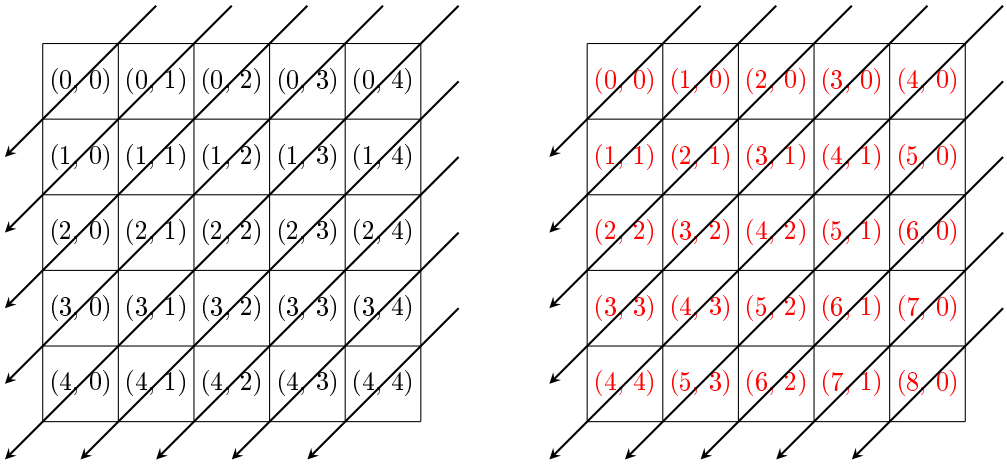
Algorithm 1: Brute-force search

```
function find_five(board, row, col, color):
    # Define the directions:  vertical, horizontal, and the two diagonals
    directions ← {(0, 1), (1, 0), (−1, 1), (1, 1)}
    foreach direction in directions do
        forall position do                                  # for all positions (x, y) on the board
            count ← 0
            for k ← 0 to 4 do
                new_position ← position + k × direction
                if new_position is out of bounds then
                    break
                end
                if get_color(new_position) is color then
                    count ← count + 1
                end
            end
            if count = 5 then
                return True
            end
        end
    end
    return False
end
```



(a) Before conversion (b) After conversion

Figure 2.11: $direction = 2$



(a) Before conversion (b) After conversion

Figure 2.12: $direction = 3$

To use this optimisation, the cells must be in an array. For orthogonal lines, it can be easily implemented, since the cells are ordered in a Cartesian plane system. However, for diagonal lines, it may be hard to implement. We can overcome this problem by using new coordinate systems.

Coordinate conversion

We can use the Cartesian system for orthogonal directions. We can define $direction = 0$ corresponds to horizontal lines and $direction = 1$ corresponds to vertical lines. Let (a, b) be the coordinates in the new system. In this system, when traversing a line, a should remain constant and b should be incremented when the next cell on the line is reached. Therefore we can set a to be row and b to be col for $direction = 0$ and vice versa for $direction = 1$.

Let $direction = 2$ corresponds to southeastwards and $direction = 3$ corresponds to southwestwards. Notice that on a line of $direction = 2$, the quantity $col - row$ is invariant, so can set a based on this value. Same for the quantity $col + row$ for $direction = 3$. We can then set b according to the requirements with some careful handling. An example for a conversion for both directions in a 5 by 5 grid is shown in Figures 2.11 and 2.12.

Partial sum with converted coordinates

Using this coordinate system (called the *line* system hereinafter), we can apply the partial sum technique. We define the 4-dimensional array $partial\_count$, such that $partial\_count[dir][color][a][b]$ is the number of cells with color $color$ on a line of direction dir that is before the cell (a, b). After constructing the array, we can query for the number of cells with color $color$ on a line of direction dir starting from (a, b) with a length of a given $length$ using the function $range\_query$. Its pseudo-code is shown in Algorithm 2.

Now $find\_five$ takes a maximum of $15 \times 15 \times 4 = 900$ calls to $range\_query$ which is an improvement.

Algorithm 2: Partial count

```
n ← 15 # n is the size of the board
function line_to_cartesian(dir, a, b):
    if dir = 0 then
        return (a, b)
    else if dir = 1 then
        return (b, a)
    else if dir = 2 then
        d ← max(n − 1 − a, 0)
        return (b + d, b − d)
    else                                                        # dir = 3
        d ← max(a − (n − 1), 0)
        return (b + d, a − b − d)
    end
end
# Compute partial_count
partial_count ← 4-D array with sizes [4][3][2n − 1][n + 1] initialised with 0
for dir ← 0 to 3 do
    forall (a, b) of direction = dir do
        (row, col) = line_to_cartesian(dir, a, b)
        foreach color in {black, white, blank} do          # We can encode the color with 0, 1, 2
            partial_count[dir][color][a][b + 1]
                ← partial_count[dir][color][a][b] + (1 if get_color(row, col) is color else 0)
        end
    end
end
function range_query(dir, a, b, color, length):
    # Assuming (dir, a, b) is in bounds
    return partial_count[dir][color][a][b + length] − partial_count[dir][color][a][b]
end
# Now we can compute find_five using range_query()
function find_five(board, row, col, color):
    for dir ← 0 to 3 do
        forall (a, b) of direction = dir do
            # Handle the out of bounds cases
            if range_query(dir, a, b, color, 5) = 5 then
                return True
            end
        end
    end
    return False
end
```

2.2.2 Computer player: Minimax

Minimax

Both Gomoku and Renju is a zero-sum game, meaning that an advantage for one side is equivalent to a disadvantage for the other side. Minimax is an algorithm that is commonly used in these games, and the key idea is to maximise the minimum gain (or minimise the maximum loss) of every move, hence the name.

This algorithm requires a measure of value in a board state, and we say the value is positive it is advantageous to black, and vice versa. This algorithm assumes that both players play their best move, which means they play the move that maximises their minimum gain, according to the evaluation of the board state. To determine the best move for black, the algorithm maximises the value of every board state which the current board state can lead to, assuming white plays the best move.

To find the white's best move we can use the same method, but this time we minimise the values of the subsequent board states, and again we assume optimal play from black. Therefore, we can implement this algorithm recursively, finding the best move for both players, until the depth is reached. The pseudo code of this algorithm is shown in Algorithm 3.

Minimax is a algorithm that **traverses** the game tree (refer to 1.8.3). It uses **post-order traversal**. It processes the children of a node before the node itself. It is a **recursive** algorithm, and the base case is when the depth is 0, or when the board is won. In this case, the value of the board is returned. Otherwise, the algorithm will return the maximum or minimum value of the children depending on the colour of the player.

Algorithm 3: Minimax

```
function minimax(board, depth, color):
    if depth = 0 or board is won then
        return eval(board)
    end
    if color is black then
        value ← −∞
        foreach candidate in board.candidates do
            candidate_value ← minimax(candidate, depth - 1, white)
            value ← max(value, candidate_value)
        end
        return value
    end
    else
        value ← ∞
        foreach candidate in board.candidates do
            candidate_value ← minimax(candidate, depth - 1, black)
            value ← min(value, candidate_value)
        end
        return value
    end
end
# Initial call
d ← 4 # depth
minimax(board, d, color)
```

This algorithm has time complexity $O(n^d)$, where n is the maximum number of candidate boards which will be explained in the next subsection, and d is the depth specified in the parameter of the initial call to minimax.

Candidate boards

In minimax, we would consider all possible moves for both players, i.e., they all become candidate moves. The boards created when these moves are played are called candidate boards. In Gomoku, there are at most $15^2 = 225$ candidate boards. Since the time complexity of minimax is exponential in the number of candidate boards, we want to limit the number of candidate boards.

In practice, most candidate moves are obviously bad and is never considered (e.g. a move at a corner of the board). We can take advantage of this, and only consider moves that are at most r distance from any currently exisiting piece (*Eliminating actions of no real value* in [2]). We can get all such candidate boards (called the *search space* in this report) by the flowchart in Figure 2.13.

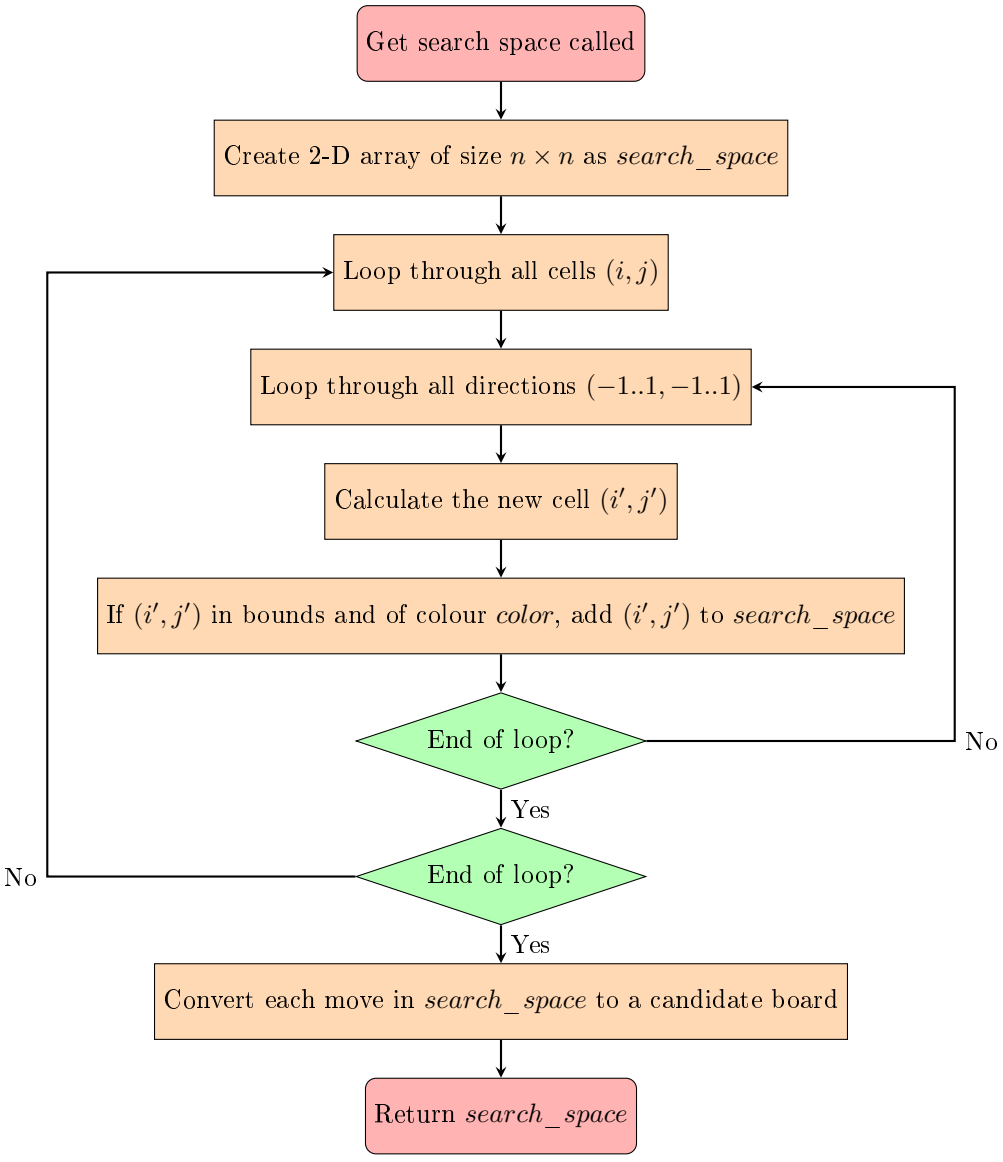


Figure 2.13: Search space

Minimax with local evaluation

As the time complexity of minimax is exponential in n, we want to keep it as low as possible. Therefore, we should limit the number of candidates of each node, which is the *branching factor*.

Instead of considering all possible moves from a board state, we want to limit it down to a chosen value b. We can achieve this by doing local evaluations. We only search the top b best moves according to their current evaluation.

In this way, we reduce the time complexity to $O(b^d)$.

Minimax with alpha-beta pruning

Intuitively, we can see that minimax has some inefficiencies when it searches for moves that is obviously bad for a player. To solve this problem, we can use this optimisation called alpha-beta pruning.

Alpha-beta pruning can reduce the number of nodes in the search tree. It stops evaluating a node when the result of it does not affect the final result of the computation.

This algorithm works by maintaining two variables α and β. They represent the minimum value of the maximising player (black) and the maximum value of the minimising player (white) respectively during the tree search. If at any stage $\alpha \geq \beta$, we can stop searching, since for both players the move will not be advantageous. For black, we do not need to search of any moves that has value less than α, since it is guaranteed that black can get at least a value of α in one of the branches. The same argument can be made with white. It can also be shown that with this pruning, the algorithm gives the same output as the original version of minimax.

To fully utilise this method we want to maximise α and minimise β, or in other words, we want to search the best moves for both players first. This can be done with a sorted candidate list, and can be combined with local evaluation. Algorithm 4 shows the pseudo code with both local evaluation and alpha-beta pruning.

Algorithm 4: Minimax with optimisations

```
b ← 5 # The limit of the branching factor
function minimax(board, depth, color, α, β):
    if depth = 0 or board is won then
        return eval(board)
    end
    candidates ← board.candidates
    sort candidates by eval descending
    candidates ← candidates[0 : b] # Take only the first b candidates
    if color is black then
        value ← −∞
        foreach candidate in candidates do
            candidate_value ← minimax(candidate, depth - 1, white, α, β)
            α ← max(α, candidate_value)
            if α ≥ β then
                break
            end
            value ← max(value, candidate_value)
        end
        return value
    end
    else                                                    # color is white
        value ← ∞
        foreach candidate in candidates do
            candidate_value ← minimax(candidate, depth - 1, black, α, β)
            β ← min(β, candidate_value)
            if α ≥ β then
                break
            end
            value ← min(value, candidate_value)
        end
        return value
    end
end
# Initial call
d ← 4 # depth
minimax(board, d, color, −∞, ∞)
```

This algorithm gives the new best-case performance of $O(\sqrt{b^d})$, if all unnecessary nodes are optimally pruned. With a good evaluation function, we can expect a significant improvement over the original minimax algorithm.

Evaluation function

To correctly use the minimax algorithm, we need a good evaluation function. The evaluation function should be able to evaluate the board state and give a value that represents the advantage (or disadvantage) of black. It should also be able to evaluate the board state quickly.

In light of the evaluation function used in [2], we can use a similar evaluation function. We use the following:

$$\begin{aligned} \texttt{eval}(\text{board}) &= w_0 \text{ if board is won} \\ \texttt{eval}(\text{board}) &= w_1 \times (\#\text{straight four}) + w_2 \times (\#\text{open three}) + \\ &\quad w_3 \times (\#\text{four}) + w_4 \times (\#\text{three}) + \\ &\quad w_5 \times (\#\text{two}) + w_6 \times (\#\text{single marks}) \text{ otherwise} \end{aligned} \tag{2.1}$$

where $w_0, w_1, w_2, w_3, w_4, w_5, w_6$ are weights that can be adjusted. The weights can be adjusted to reflect the importance of each pattern. For example, if we want to prioritise straight fours, we can set w_1 to be a large value. The values I have used can be seen in `constants.py`.

With partial sum and converted coordinates, we can calculate the counts of board patterns quickly. We can then use these counts to evaluate the board state.

Multi-threading

Even with optimisations, the minimax algorithm still requires a lot of time to run. During this time, the GUI will be irresponsive. To tackle this problem, we can use multi-threading to run the minimax algorithm in the background, keeping the GUI responsive.

2.2.3 Computer player: Prioritise Threats

In this subsection, we will use the heuristic "Threat-space Search" developed by [1]. In their paper, they mentioned the idea of threats, which are compelling moves that must be responded by the opponent. Similar to their work, we define the following shapes of pieces as threats with decreasing power (also see Figure 2.14):

- *five*: a line of five squares, of which the attacker has occupied all five. This is a winning move.
- *straight four*: a line of six squares, of which the attacker has occupied the four center squares, while the two outer squares are empty. This is a winning threat.
- *four*: a line of five squares, of which the attacker has occupied any four, with the fifth square empty.
- *open three*: a shape such that the attacker can create a *straight four* in his next move
- *three*: a shape such that the attacker can create a *four* in his next move
- *open two*: a shape such that the attacker can create an *open three* in his next move

Since threats must be responded, we can use this to our advantage. Instead of searching the whole board for moves, we can search for our threats and the opponent's threats, then play or respond to them. Since the number of such moves is usually small compared to the *search space*, the search will be done very quickly. Therefore, instead of decreasing depth by one step, we can decrease it a fraction of a step. In practice, we could first multiply the maximum depth by some number, so we do not need to deal with fractions. Algorithm 5 shows the move-finding algorithm that prioritises threats.

Of course, we will combine this algorithm with alpha-beta pruning and local evaluation to further improve the performance.

2.2.4 Measurement of skill level

Just like many other quantities in real life, the skill level of a player can only be measured relatively to other players. In this case, it can be measured according to the wins and losses of a player. We would like a rating system that can find the probability of winning of a player given the ratings from both players.

With inspiration from the Elo rating system [3], we can use a logarithmic scale. Let $Q_A = 10^{R_A/400}$ and $Q_B = 10^{R_B/400}$, where R_A and R_B are the ratings of player A and B respectively. Now we can calculate the expected win rates for both players, E_A and E_B, by

$$E_A = \frac{Q_A}{Q_A + Q_B}, E_B = \frac{Q_B}{Q_A + Q_B}$$

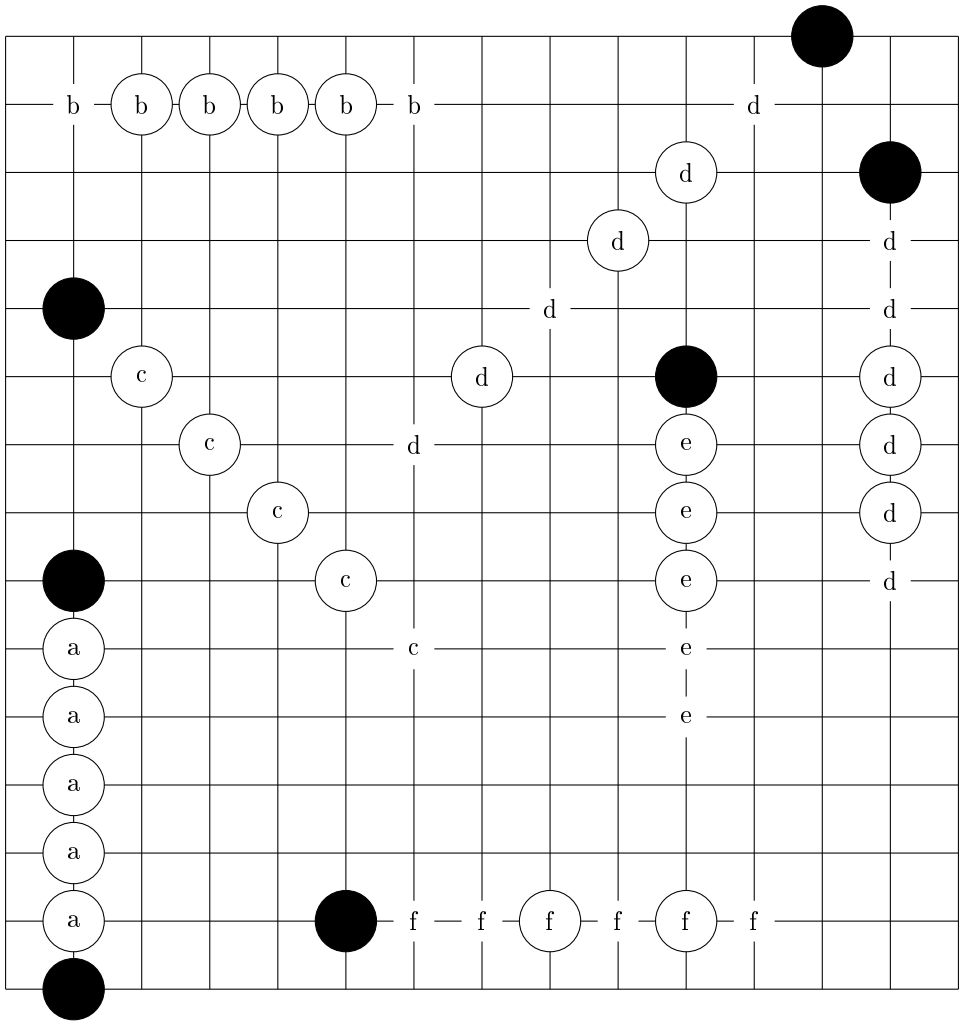


Figure 2.14: The shapes of threats. (a): five, (b): straight four, (c): four, (d): open three, (e): three, (f): open two.

Algorithm 5: Prioritise threats

```
function search(board, depth, color):
    if depth ≤ 0 or find_five(board, color) or find_five(board, opp_color) then
        return eval(board)
    end
    # Play a four if possible
    fours ← find_four(board, color)
    if fours then
        return search(fours[0], depth − 1, opp_color)
    end
    # Block a four if needed
    fours_opp ← find_four(board, opp_color)
    if fours_opp then
        return search(fours_opp[0], depth − 1, opp_color)
    end
    # Play an open three if possible
    open_threes ← find_open_three(board, color)
    if open_threes then
        return search(open_threes[0], depth − 1, opp_color)
    end
    # Refute a open three if needed
    open_threes_opp ← find_open_three(board, opp_color)
    if open_threes_opp then
        # Refute by creating a four
        threes ← find_three(board, color)
        candidate_boards ← open_threes_opp + threes
        return minimax(candidate_boards, depth − 2, opp_color)
    end
    # No more immediate threats, use minimax on remaining threats and non-threats
    threes ← find_three(board, color)
    open_twos ← find_open_two(board, color)
    candidate_boards ← threes + open_twos+ get_search_space(board)
    return minimax(candidate_boards, depth − 4, color)
end
# Initial call
d ← 4 ∗ MAX_DEPTH # multiply max depth by 4
search(board, d, color)
```

To update one's ratings, we should first define the sensitivity of the system with a constant called the K-factor K. In this project, I used $K = 32$. Say player A and player B played a game. We define the score of A $S_A = 1$ if player A won, $S_A = 0.5$ if the game has drawn, and $S_A = 0$ if player A lost. We define the same for S_B. We update both players' ratings by the formulae

$$R'_A = R_A + K \times (S_A - E_A)$$
$$R'_B = R_B + K \times (S_B - E_B)$$

According the US Chess Federation (USCF) convention, an average player should have a rating of 1500. We can follow this convention and assign a rating of 1500 to the user and every computer player in the beginning.

2.2.5 Sorting

In minimax, sorting is essential for the use of local evaluation. We need to use sorting to find the best candidate moves for both players, then use alpha-beta allowing pruning of nonoptimal moves. We also need to use sorting when we want to arrange the ratings of computer players in descending order. Therefore, we need a efficient sorting algorithm.

For minimax, I used bubble sort to sort the list of candidate boards. This is because we do not need to sort the whole list, we only need to find the best b boards, where b is the branching factor (which is at most 10). Bubble sort is simple and easy to implement, and it is efficient since it does not have recursion overheads when compared to quicksort or merge sort. The time complexity of bubble sort in this case is $O(nb)$, but since b is a small constant, we can treat it as $O(n)$.

For sorting the ratings of computer players, I used merge sort. Merge sort is a famous sorting algorithm that uses the divide-and-conquer strategy. It has a time complexity of $O(n \log n)$. It is a recursive algorithm, the base case is when the list has only 1 element. In this case, the list is already sorted. Otherwise, it divides the list into two halves, sorts them, and then merges them. It has the the optimal worst-case time complexity among all comparison-based[1] sorting algorithms.

2.2.6 Legality of Renju moves

According to Section 9 of the official Renju rules[2], a move on an empty intersection is considered illegal if:

- It creates an *overline* (a line of six or more stones).
- Without creating a *five* (directly winning), it creates two *open threes* in two different directions simultaneously, (also called a *3x3 fork*).
- Without creating a *five* (directly winning), it creates two *fours* in two different directions simultaneously, (also called a *4x4 fork*).

Recall that an *open three* is a shape such that the attacker can make a **legal** move and create a *straight four*, and a *four* is a shape such that the attacker can make a **legal** move and create a *five*.

Therefore, the problem of determining the legality of a move is self-referential. We can solve this problem by using a **recursive** algorithm. We can first make the move, then check if the move creates any threes. If it does, we need to play more moves to determine if the three is an open three, and we are solving the same problem when determining whether those new moves are legal. A similar procedure should be done to determine if the move creates a 4x4 fork. The **base case** is when a move does not create two potential threes or fours, and the move is considered legal if it does not create an overline (which does not need to be checked recursively). It is also worth noting that the algorithm **must terminate**, since in each recursive call, we place a new move on the board, and the board is finite.

2.3 Classes

2.3.1 Class relation diagram

Using object-oriented programming, we can organise the code into objects. This makes the code more modular and easier to understand. I would like to use the following classes to organise my code:

- `MainScreen`: The main screen of the game.
- `TitleWindow`: The window where the title is shown.
- `PlayWindow`: The window where the game is played.

[1] Comparison-based sorting algorithms are algorithms that only uses a single function for comparison of two elements. An example of a non-comparison based algorithm is counting sort.

[2] `https://www.renju.net/rifrules/`

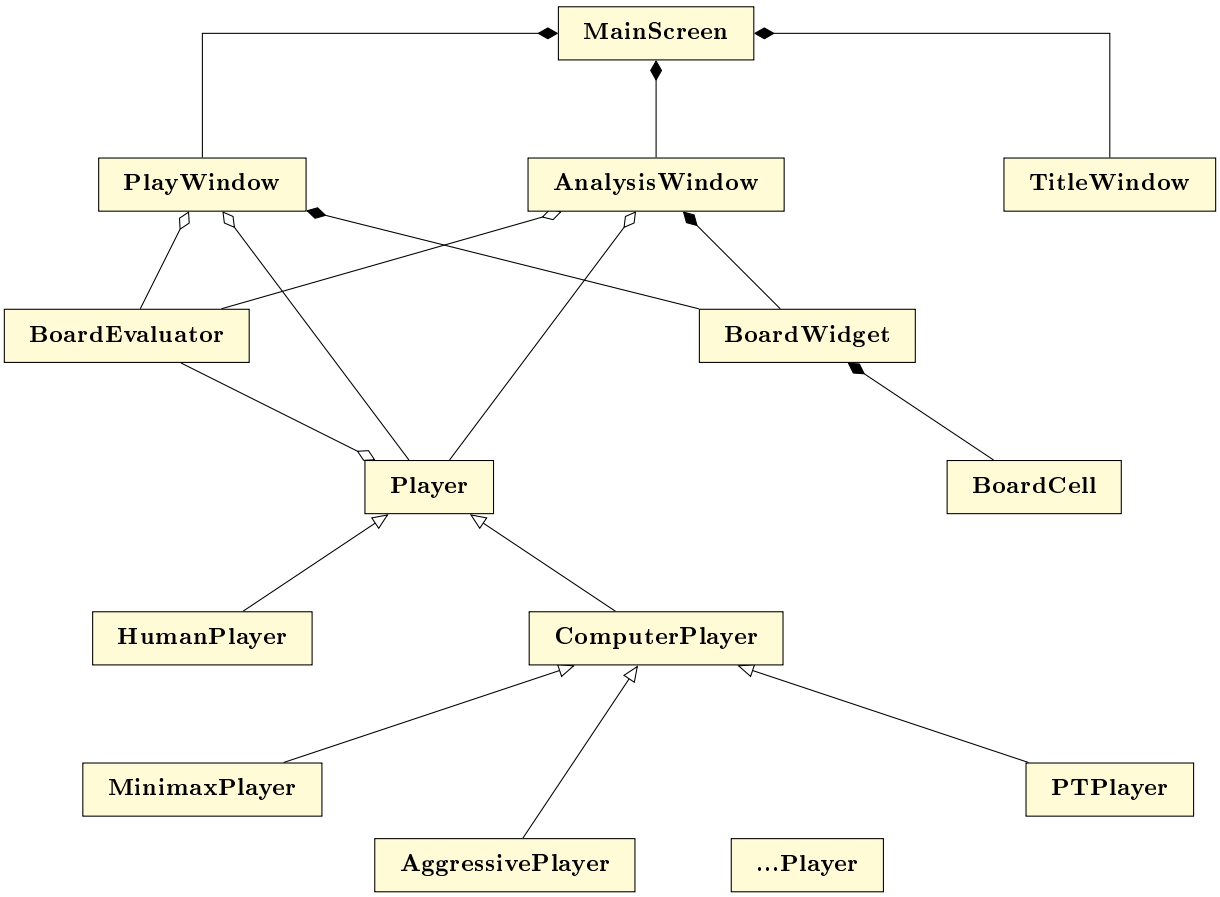


Figure 2.15: The class relation diagram of the system.

- `AnalysisWindow`: The window where the game is analysed.
- `BoardEvaluator`: The class that provides many methods to evaluate the board state.
- `BoardWidget`: The class that manages the GUI of the board.
- `BoardCell`: The class that represents a cell in the board.
- `Player`: The abstract class for a player.
- `HumanPlayer`: The class for a human player.
- `ComputerPlayer`: The abstract class for a computer player.
- `...Player` (e.g. `MinimaxPlayer`): A computer player that uses a certain strategy.

Figure 2.15 shows the relationship between the classes.

2.3.2 Class details

MainScreen

This class is the main screen of the game. It contains title window, the play window, and the analysis window. It also gives public methods to switch between these windows. Its attributes and methods are shown in Figure 2.16.

PlayWindow

This class represents the play window. It contains the board widget. It contains the code to ensure the game is played correctly. It also aggregates the board evaluator and the player (and its subclasses). The board evaluator class is used to determine the legality of a move, and whether the game has ended. The player class is used to represent the two players of the game. If a player is a computer player, its methods are used to make a move. Otherwise, it will wait for the user to make a move. Its attributes and methods are shown in Figure 2.17.

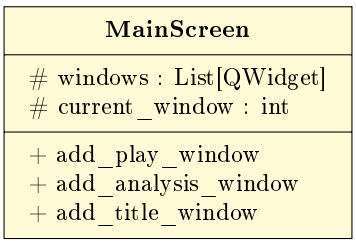


Figure 2.16: The attributes and methods of MainScreen.

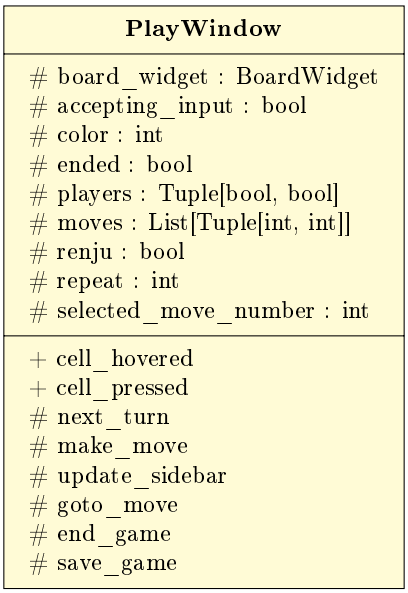


Figure 2.17: The attributes and methods of PlayWindow.

AnalysisWindow

This class represents the analysis window. Similar to PlayWindow, it also contains the board widget. It contains the code for post-game analysis. It aggregates the board evaluator for the same reason as PlayWindow. The methods of Player will be called when the user asks for a computer player to analyse the board or to give continuation moves. Its attributes and methods are shown in Figure 2.18.

BoardCell

This class represents a cell in the board. It can only be instantiated within BoardWidget. It contains the code to display the cell and to handle the user's input (hover or click). Using the observer pattern, it will notify its parent (BoardWidget) when the user clicks on it. Its parent will then notify the PlayWindow or AnalysisWindow that a cell has been hovered or pressed. Its attributes and methods are shown in Figure 2.19.

ComputerPlayer

This class is the abstract class for a computer player. It is a subclass of Player. It gives an interface for all computer players to implement. For example, computer players need to implement how to make a move. Its attributes and methods are shown in Figure 2.20.

Other classes

The other classes will not be explained in detail, as the usage of these classes are straightforward. The details of these classes can be found in the source code.

- `TitleWindow`: Can only be instantiated by MainScreen.
- `BoardEvaluator`: A utility class that is aggregated by PlayWindow, AnalysisWindow, and Player (in particular, ComputerPlayer).
- `BoardWidget`: Composed of BoardCell, and is a part of PlayWindow and AnalysisWindow.

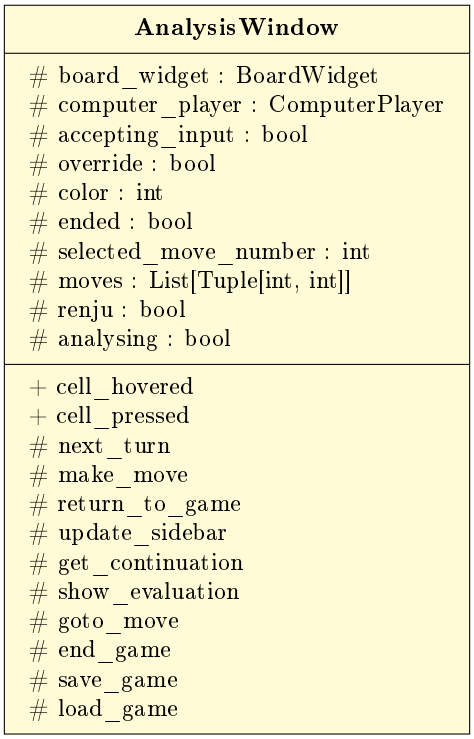


Figure 2.18: The attributes and methods of AnalysisWindow.

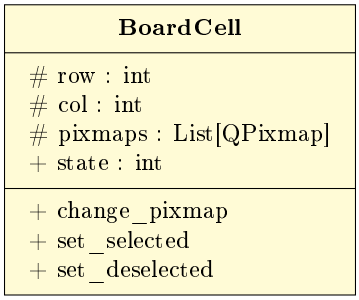


Figure 2.19: The attributes and methods of BoardCell.

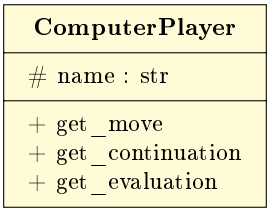


Figure 2.20: The attributes and methods of ComputerPlayer.

- `Player`: An abstract class that is inherited by HumanPlayer and ComputerPlayer. All players must have the attribute `name`.

Chapter 3

Technical Solution

3.1 Summary of Complexity

- Bubble sort. `utils.py` line 24.
- Merging and merge sort. `utils.py` lines 45, 70.
- Coordinate conversion and partial sum technique. `board_evaluator.py` lines 72 - 202.
- Checking for legality of moves recursively. `board_evaluator.py` lines 206 - 359.
- Minimax algorithm. `computer_players/minimax_player.py` line 58,
  `computer_players/aggressive_player.py` line 66,
  `computer_players/defensive_player.py` line 66,
  `computer_players/prioritise_threats_player.py` line 133.
- Recursively searching threats. `computer_players/prioritise_threats_player.py` line 56.
- File handling and JSON parsing. `analysis_window.py` line 363, `elo_ratings.py` line 14.

3.2 Constants

`constants.py`

```
1  DEBUG = False
2  BOARD_SIZE = 15
3
4  # Evaluation function
5  WIN = 10000
6  STRAIGHT_FOUR = 2000
7  OPEN_THREE = 750
8  FOUR = 250
9  THREE = 80
10 TWO = 20
11 ONE = 10
12 AGGRESSIVE_FACTOR = 1.5
13 DEFENSIVE_FACTOR = 1.5
14
15 # Threshold evaluation points for showing messages
16 CLOSE_TO_WINNING_THRES = 2000
17 GREAT_ADVANTAGE_THRES = 1000
18 ADVANTAGE_THRES = 500
19 SMALL_ADVANTAGE_THRES = 200
20
21 # ELO rating
22 K_FACTOR = 32
23 DEFAULT_ELO = 1500
```

3.3 Utilities

This file contains some utility functions that will be used in my code.
utils.py

```
from typing import List, Tuple, TypeVar, Callable, Iterator
from itertools import islice

from constants import *

def previous_of(evaluation: int) -> int:
    """Modifies the evaluation if either player has a forced winning sequence.
    For example, if black has a forced winning sequence of 5 moves, then the evaluation is 9995.
    If white has a forced winning sequence of 3 moves, then the evaluation is -9997.

    Args:
        evaluation (int): The evaluation to be modified.

    Returns:
        int: The modified evaluation.
    """
    if evaluation >= WIN - 100:
        return evaluation - 1
    if evaluation <= -WIN + 100:
        return evaluation + 1
    return evaluation

T = TypeVar('T')
def bubble_sort_k(l: List[T], k: int, key: Callable[[T], int] = id, reverse: bool = False) -> List[T
    ]:
    """Sorts the list in increasing order using bubble sort. Time complexity: O(nk).

    Args:
        l (List[T]): The list to be sorted.
        key (Callable[[T], int]): The function to extract the key from the elements.
        k (int): The number of elements to return.
        reverse (bool): Whether to sort in reverse.

    Returns:
        List[T]: The first k elements of the sorted list.
    """
    n = len(l)
    keys = [key(board) for board in l]
    for i in range(min(k, n)):
        for j in range(n-1, i, -1):
            if (keys[j-1] > keys[j]) ^ reverse: # XOR
                l[j], l[j-1] = l[j-1], l[j]
                keys[j], keys[j-1] = keys[j-1], keys[j]
    return l[:k]

def merge(l1: List[T], l2: List[T], key: Callable[[T], int], reverse: bool) -> List[T]:
    """Merges two sorted lists.

    Args:
        l1 (List[T]): The first sorted list.
        l2 (List[T]): The second sorted list.
        key (Callable[[T], int]): The function to extract the key from the elements.
        reverse (bool): Whether to sort in reverse.

    Returns:
        List[T]: The merged sorted list.
    """
    result = []
    i = j = 0
    while i < len(l1) and j < len(l2):
        if (key(l1[i]) < key(l2[j])) ^ reverse:
            result.append(l1[i])
            i += 1
        else:
            result.append(l2[j])
            j += 1
    result.extend(l1[i:])
    result.extend(l2[j:])
    return result

def merge_sort(l : List[T], key: Callable[[T], int] = id, reverse: bool = False) -> List[T]:
```

```
    """Sorts the list in increasing order using merge sort.

    Args:
        l (List[T]): The list to be sorted.
        key (Callable[[T], int]): The function to extract the key from the elements.
        reverse (bool, optional): Whether to sort in reverse. Defaults to False.

    Returns:
        List[T]: The sorted list.
    """
    if len(l) <= 1:
        return l
    mid = len(l) // 2
    left = merge_sort(l[:mid], key, reverse)
    right = merge_sort(l[mid:], key, reverse)
    return merge(left, right, key, reverse)

def generate_evaluation_message(evaluation: int) -> str:
    """Generates a message based on the evaluation.

    Args:
        evaluation (int): The evaluation of the board.

    Returns:
        str: The message.
    """
    str_color = 'Black' if evaluation > 0 else 'White'
    evaluation = abs(evaluation)
    if evaluation >= WIN - 100:
        return f'{str_color} forced win in {WIN - evaluation} moves ({evaluation})'
    if evaluation >= CLOSE_TO_WINNING_THRES:
        return f'{str_color} close to winning ({evaluation})'
    if evaluation >= GREAT_ADVANTAGE_THRES:
        return f'{str_color} has a great advantage ({evaluation})'
    if evaluation >= ADVANTAGE_THRES:
        return f'{str_color} has an advantage ({evaluation})'
    if evaluation >= SMALL_ADVANTAGE_THRES:
        return f'{str_color} has a small advantage ({evaluation})'
    return f'Game even ({evaluation})'

def next_k(iterator: Iterator, k: int) -> None:
    """Advances the iterator by k steps.

    Args:
        iterator (Iterator): The iterator to be advanced.
        k (int): The number of steps to advance.
    """
    next(islice(iterator, k, k), None)

if __name__ == '__main__':
    # Unit tests
    l = [5, 3, 8, 6, 2, 7, 1, 4]
    print(bubble_sort_k(l, 5)) # [1, 2, 3, 4, 5]
    print(bubble_sort_k(l, 5, reverse=True)) # [8, 7, 6, 5, 4]
    print(merge_sort(l)) # [1, 2, 3, 4, 5, 6, 7, 8]
    print(merge_sort(l, reverse=True)) # [8, 7, 6, 5, 4, 3, 2, 1]
```

3.4 Main

This file is the file that will be ran.
main.py

```
import sys
from typing import List, Tuple

from PyQt6.QtWidgets import QApplication, QStackedWidget
from PyQt6 import QtCore

from title_window import TitleWindow
from play_window import PlayWindow
from analysis_window import AnalysisWindow
from rankings_window import RankingsWindow
from players import Player, HumanPlayer
from constants import DEBUG
```

```
13
14 class MainScreen(QStackedWidget):
15     """The top-level object that contains all other windows.
16
17     Attributes:
18         _windows (List[QWidget]): A list of all the windows in the program.
19
20     Methods:
21         add_play_window: Add a play window to the stack.
22         add_analysis_window: Add an analysis window to the stack.
23         add_rankings_window: Add a rankings window to the stack.
24
25     Overridden Methods:
26         keyPressEvent: Handle key presses.
27     """
28     def __init__(self) -> None:
29         super().__init__()
30         self.setWindowTitle("Gomoku and Renju")
31         self.setStyleSheet(open("./stylesheet.css").read())
32
33         self.addWidget(TitleWindow(self))
34
35         self.setCurrentIndex(0)
36
37         self.showMaximized()
38
39     def add_play_window(self, renju: bool, players: List[Player], repeat: int = 1):
40         """Add a play window to the stack. Called by the title window.
41
42         Args:
43             renju (bool): Whether renju rules are enabled.
44             players (List[Player]): The players in the game.
45             repeat (int): The number of games to play.
46         """
47         if self.count() != 1: return
48         if all([type(player) is not HumanPlayer for player in players]):
49             print(f'{repeat} match(es) remaining.')
50         self.setWindowTitle(f"Play - {'Renju' if renju else 'Gomoku'}")
51         self.addWidget(PlayWindow(self, renju, players, repeat))
52         self.setCurrentIndex(1)
53
54     def add_analysis_window(self):
55         if self.count() != 1: return
56         self.setWindowTitle("Analysis")
57         self.addWidget(AnalysisWindow(self))
58         self.setCurrentIndex(1)
59
60     def add_rankings_window(self):
61         if self.count() != 1: return
62         self.setWindowTitle("Rankings")
63         self.addWidget(RankingsWindow(self))
64         self.setCurrentIndex(1)
65
66     def remove_window(self):
67         if self.count() != 2: return
68         self.removeWidget(self.currentWidget())
69         self.setCurrentIndex(0)
70         self.setWindowTitle("Gomoku and Renju")
71
72     def keyPressEvent(self, e):
73         if DEBUG and e.key() == QtCore.Qt.Key.Key_Escape:
74             self.close() # closes the current window
75         if e.key() == QtCore.Qt.Key.Key_F11:
76             if self.isMaximized():
77                 self.showNormal()
78             else:
79                 self.showMaximized()
80
81
82 if __name__ == '__main__':
83     app = QApplication(sys.argv)
84
85     screen = MainScreen()
86     screen.show()
87
88     app.exec()
```

3.5 TitleWindow

title_window.py

```
from __future__ import annotations
from copy import deepcopy
from typing import TYPE_CHECKING
if TYPE_CHECKING:
    from main import MainScreen

from PyQt6.QtWidgets import QMainWindow, QDialog
from PyQt6 import uic

from players import HumanPlayer
from computer_players.random_player import RandomPlayer, COMPUTER_PLAYERS

class TitleWindow(QMainWindow):
    """The window that shows the title screen.

    UI Elements:
        play_button (QPushButton): The button that opens the configuration menu, then shows the board.
        analysis_button (QPushButton): The button that opens the analysis window.
        rankings_button (QPushButton): The button that opens the rankings window.
    """
    def __init__(self, parent: MainScreen):
        """Initialises the TitleWindow.

        Args:
            parent (MainScreen): The parent MainScreen widget.
        """
        super().__init__()
        # Some UI files are stored in the ui folder.
        # They are XML files that describes the properties of the UI elements.
        # They are created using the application Qt Designer.
        # Qt Designer is a powerful tool that uses the WYSIWYG (What You See Is What You Get) approach to create UIs.
        # It allows me to create the UIs by dragging and dropping UI elements, drastically lowering the development time.

        # The uic module loads the UI file and creates the UI elements.
        # For example, here it creates the buttons of the title window, and
        # allows me to reference the buttons with self.play_button, self.analysis_button, and self.rankings_button.
        # In some other classes, their UI files are loaded in the same way,
        # so you may see me referencing UI elements without defining them.
        uic.loadUi('ui/title_window.ui', self)

        # Define the behaviour when the buttons are pressed
        def play_button_pressed():
            """Called when the play button is pressed. It opens the configuration menu, then shows the board."""
            configuration_dialog = QDialog(self)
            uic.loadUi('ui/configuration.ui', configuration_dialog)
            configuration_dialog.setWindowTitle("Configuration")
            configuration_dialog.setModal(True)

            # add the players' names to the combo boxes
            all_players = {player.name: player for player in ([HumanPlayer()] + COMPUTER_PLAYERS)}
            all_players['Random'] = RandomPlayer()
            configuration_dialog.black_combo_box.addItems(all_players.keys())
            configuration_dialog.white_combo_box.addItems(all_players.keys())

            # when both players are computer players, the repeat option is shown
            def combo_box_changed():
                if configuration_dialog.black_combo_box.currentText() == 'Human Player' or \
                        configuration_dialog.white_combo_box.currentText() == 'Human Player':
                    configuration_dialog.repeat_option.hide()
                else:
                    configuration_dialog.repeat_option.show()
            configuration_dialog.black_combo_box.currentIndexChanged.connect(combo_box_changed)
            configuration_dialog.white_combo_box.currentIndexChanged.connect(combo_box_changed)
            configuration_dialog.repeat_option.hide()

            def configuration_play_button_pressed():
                renju = bool(configuration_dialog.game_combo_box.currentIndex())
                # RandomPlayer has a state (the instance), so players must be deepcopied
```

```
                black_player = deepcopy(all_players[configuration_dialog.black_combo_box.currentText()])
                white_player = deepcopy(all_players[configuration_dialog.white_combo_box.currentText()])
                repeat = configuration_dialog.repeat_spin_box.value()
                if type(black_player) == HumanPlayer or type(white_player) == HumanPlayer:
                    parent.add_play_window(renju, (black_player, white_player))
                else:
                    parent.add_play_window(renju, (black_player, white_player), repeat)
                configuration_dialog.close()
            configuration_dialog.play_button.pressed.connect(configuration_play_button_pressed)

            configuration_dialog.show()
        self.play_button.pressed.connect(play_button_pressed)

        def analysis_button_pressed():
            parent.add_analysis_window()
        self.analysis_button.pressed.connect(analysis_button_pressed)

        def rankings_button_pressed():
            parent.add_rankings_window()
        self.rankings_button.pressed.connect(rankings_button_pressed)
```

3.6 PlayWindow

play_window.py

```
from __future__ import annotations
from typing import TYPE_CHECKING, Tuple, List
if TYPE_CHECKING:
    from main import MainScreen
from datetime import datetime
import json

from PyQt6.QtWidgets import QMainWindow, QPushButton, QLabel, QDialog
from PyQt6.QtCore import Qt, QThreadPool
from PyQt6 import uic

from players import Player, HumanPlayer
from computer_players.random_player import RandomPlayer
from board_widget import BoardWidget
from board_evaluator import BoardEvaluator
from worker import Worker
from elo_ratings import EloRatings
from constants import DEBUG

class PlayWindow(QMainWindow):
    """The window where the game is played.

    UI Elements:
        board_widget_placeholder (QWidget): The placeholder for the board widget.
        bottom_text (QLabel): The label that shows the bottom text.
        status_bar (QStatusBar): The status bar that shows the game status.
        return_to_game_button (QPushButton): The button that returns to the latest move.
        exit_button (QPushButton): The button that exits the game.
        save_action (QAction): The action that saves the game.
        black_player_name (QLabel): The label that shows the name of the black player.
        white_player_name (QLabel): The label that shows the name of the white player.
        sidebar (QWidget): The widget that shows the sidebar.

    Attributes:
        _board_widget (BoardWidget): The widget that displays the board.
        _accepting_input (bool): Whether the player can make a move.
        _color (int): The color of the player whose turn it is. 1 for black, 2 for white.
        _ended (bool): Whether the game has ended.
        _players (Tuple[bool, bool]): A tuple of two booleans, indicating whether each player is a player.
        _moves (List[Tuple[int, int]]): A list of the moves made in the game.
        _renju (bool): Whether renju rules are enabled.
        _repeat (int): The number of games to play.
        _selected_move_number (int): The move number that is currently selected.

    Methods:
        cell_hovered: Called when a cell of board_widget is hovered over.
        cell_pressed: Called when a cell of board_widget is pressed.
```

```
48          _show_message: Shows a message in the status bar.
49          _make_move: Makes a move on the board.
50          _next_turn: Changes the turn.
51          _update_sidebar: Updates the sidebar with the move that was made.
52          _goto_move: Goes to a move in the game.
53          _end_game: Called when the game ends.
54          _save_game: Saves the game to a file.
55
56      """
57      def __init__(self, parent: MainScreen, renju: bool, players: Tuple[Player, Player], repeat: int
        = 1):
58          """Initialises the PlayWindow.
59
60          Args:
61              parent (MainScreen): The parent MainScreen widget.
62              renju (bool): Whether renju rules are enabled.
63              players (Tuple[Player, Player]): The players in the game.
64              repeat (int): The number of games to play.
65          """
66          super().__init__()
67          self.parent = parent
68
69          # Initialise GUI elements
70          uic.loadUi('ui/play_window.ui', self)
71
72          self.board_widget = BoardWidget()
73          self.left_layout.replaceWidget(self.board_widget_placeholder, self.board_widget)
74
75          def return_to_game_pressed():
76              move_number = len(self._moves) - 1
77              if move_number < 0:
78                  return
79              self.sidebar.layout().itemAtPosition(move_number // 2 + 1, move_number % 2 + 1).widget()
    .setChecked(True)
80              self._goto_move(move_number)
81          self.return_to_game_button.pressed.connect(return_to_game_pressed)
82
83          self.exit_button.pressed.connect(parent.remove_window)
84
85          self.save_action.triggered.connect(self._save_game)
86
87          for player in players:
88              if type(player) is RandomPlayer:
89                  player.__init__() # reroll random player
90          self.black_player_name.setText(players[0].name)
91          self.white_player_name.setText(players[1].name)
92
93          # Initialise PlayWindow attributes
94          self._selected_move_number = -1
95          self._threadpool = QThreadPool()
96          self._accepting_input = True
97          self._color = 2
98          self._ended = False
99          self._players = players
100         self._moves = []
101         self._renju = renju
102         self._repeat = repeat
103         self._next_turn()
104
105     def cell_hovered(self, row: int, col: int):
106         """Called when a cell is hovered over. It updates the bottom text.
107
108         Args:
109             row (int): The row of the cell.
110             col (int): The column of the cell.
111         """
112         text = f"{chr(row + ord('A'))}{col+1}"
113         if DEBUG:
114             text += f"({row}, {col})"
115         self.bottom_text.setText(text)
116         board_cell = self.board_widget.board_cells[row][col]
117         if not self._ended and self._accepting_input and board_cell.state == 0:
118             board_cell.change_pixmap(self._color + 2)
119
120     def cell_pressed(self, row: int, col: int):
121         """Called when a cell is pressed. It makes a move if it is legal.
```

```
122
123         Args:
124             row (int): The row of the cell.
125             col (int): The column of the cell.
126         """
127         if self._ended or not self._accepting_input:
128             return
129         self.status_bar.clearMessage()
130         if self.board_widget.board[row][col] != 0:
131             return
132         # place the piece in a board evaluator to check if it is legal
133         board_evaluator = BoardEvaluator(self.board_widget.board, self._color, row, col, self._renju
    )
134         legal, open_three, four, five, overline = board_evaluator.is_legal_detailed()
135         if not legal:
136             if overline:
137                 self._show_message("Illegal, it is an overline.")
138             elif open_three >= 2:
139                 self._show_message("Illegal, it is a 3x3 fork.")
140             elif four >= 2:
141                 self._show_message("Illegal, it is a 4x4 fork.")
142             else: # should be unreachable
143                 self._show_message("Unknown illegal move.")
144             return
145         self._make_move((row, col))
146
147     def _show_message(self, message: str):
148         """Shows a message in the status bar.
149
150         Args:
151             message (str): The message to be shown.
152         """
153         self.status_bar.showMessage(message)
154         print(message)
155
156     def _make_move(self, pos: Tuple[int, int]):
157         """Makes a move on the board.
158
159         Args:
160             pos (Tuple[int, int]): The position of the move.
161         """
162         self.return_to_game_button.pressed.emit() # The game should be returned to the latest move
    before making a new move.
163         row, col = pos
164         self.board_widget.make_move(row, col, self._color)
165         self._moves.append((row, col))
166         self._update_sidebar(row, col)
167         self.return_to_game_button.pressed.emit() # Goes to the newly placed move
168         # place the piece in a board evaluator to check if the game has ended
169         board_evaluator = BoardEvaluator(self.board_widget.board, self._color, renju=self._renju)
170         if board_evaluator.find_five(self._color) or board_evaluator.is_full():
171             self._end_game(draw=board_evaluator.is_full())
172             return
173         self._next_turn()
174
175     def _next_turn(self):
176         """Changes the turn. It tells a computer to make a move if it is its turn."""
177         self._color = 3 - self._color
178         current_player = self._players[self._color - 1]
179         if type(current_player) is HumanPlayer:
180             self._accepting_input = True
181         else: # Current player is a computer player
182             self._accepting_input = False
183             worker = Worker(current_player.get_move,
184                             self.board_widget.board, self._color, self._renju)
185             worker.signals.result.connect(self._make_move)
186             self._threadpool.start(worker)
187
188     def _update_sidebar(self, row: int, col: int):
189         """Updates the sidebar with the move that was made.
190
191         Args:
192             row (int): The row of the move.
193             col (int): The column of the move.
194         """
195         cur_move_number = len(self._moves) - 1
```

```
196         # add a button that when pressed, goes to this move
197         button = QPushButton(f"{chr(row + ord('A'))}{col+1}")
198         button.pressed.connect(lambda: self._goto_move(cur_move_number))
199         button.setCheckable(True)
200         self.sidebar.layout().addWidget(button, cur_move_number // 2 + 1, self._color)
201         if self._color == 1:
202             # add a new row if the move is from black
203             label = QLabel(f"{cur_move_number // 2 + 1}.")
204             self.sidebar.layout().addWidget(label, cur_move_number // 2 + 1, 0)
205 
206     def _goto_move(self, move_number: int):
207         """Goes to a move in the game. Called when a move is clicked in the sidebar.
208 
209         Args:
210             move_number (int): The move number to go to.
211         """
212         if self._selected_move_number == move_number:
213             return
214         # accept input according to the move pressed
215         if move_number < len(self._moves) - 1 or type(self._players[self._color-1]) is not
    HumanPlayer:
216             self._accepting_input = False
217         else:
218             self._accepting_input = True
219         if self._selected_move_number >= 0:
220             self.sidebar.layout().itemAtPosition(self._selected_move_number // 2 + 1, self.
    _selected_move_number % 2 + 1).widget().setChecked(False)
221         self._selected_move_number = move_number
222         self.board_widget.reset()
223         # reset the board and make the moves up to the selected move
224         for i in range(move_number + 1):
225             row, col = self._moves[i]
226             self.board_widget.make_move(row, col, 1 + i % 2)
227             if i == move_number:
228                 self.board_widget.board_cells[row][col].set_selected()
229 
230     def _end_game(self, draw: bool = False):
231         """Ends the game and updates the rankings. Repeats the game if there are more games to play.
    """
232         if DEBUG:
233             for i, move in enumerate(self._moves):
234                 print(move, end=' 'if i % 2 == 0 else '\n')
235             print()
236         self._ended = True
237         if not draw:
238             str_color = "Black" if self._color == 1 else "White"
239             self._show_message(f"{str_color} wins with a five in a row.")
240         else:
241             self._show_message("The game is a draw.")
242 
243         EloRatings().update(
244             winner = self._players[self._color - 1].name,
245             loser = self._players[2 - self._color].name,
246             draw = draw
247         )
248         if self._repeat > 1:
249             if DEBUG:
250                 print(f"{self._repeat - 1} matches remaining.")
251             # remove self and add a new play window, with one less repeat
252             self.parent.removeWidget(self)
253             self.parent.add_play_window(self._renju, self._players, self._repeat - 1)
254             return
255 
256     def _save_game(self):
257         """Saves the game to a file. Called when save game in the menu bar or Ctrl+S is pressed."""
258         save_game_dialog = QDialog(self)
259         uic.loadUi('ui/save_game.ui', save_game_dialog)
260         save_game_dialog.setWindowTitle("Save Game")
261         save_game_dialog.black_player_name.setText(self._players[0].name)
262         save_game_dialog.white_player_name.setText(self._players[1].name)
263         def save_pressed():
264             # retrieve information from the input and save the game to a json file
265             black_player_name = save_game_dialog.black_player_name.text()
266             white_player_name = save_game_dialog.white_player_name.text()
267             current_time = datetime.now().strftime('%Y%m%d')
268             file_name = f'{current_time}_{black_player_name}_vs_{white_player_name}'
```

```
269             file_content = {
270                 'black_player_name': black_player_name,
271                 'white_player_name': white_player_name,
272                 'moves': self._moves,
273                 'ended': self._ended,
274                 'win': self._color if self._ended else 0,
275             }
276             json.dump(file_content, open(f'games/{file_name}.json', 'w'))
277             save_game_dialog.close()
278         save_game_dialog.save_button.pressed.connect(save_pressed)
279         save_game_dialog.show()
```

3.7 AnalysisWindow

analysis_window.py

```
1  from __future__ import annotations
2  from typing import TYPE_CHECKING, Tuple, List
3  if TYPE_CHECKING:
4      from main import MainScreen
5  from datetime import datetime
6  import json
7
8  from PyQt6 import uic
9  from PyQt6.QtWidgets import QMainWindow, QFileDialog, QDialog, QPushButton, QLabel
10 from PyQt6.QtCore import Qt, QThreadPool
11
12 from board_widget import BoardWidget
13 from board_evaluator import BoardEvaluator
14 from worker import Worker
15 from constants import DEBUG
16 from computer_players.random_player import COMPUTER_PLAYERS
17 from utils import generate_evaluation_message
18
19 class AnalysisWindow(QMainWindow):
20     """The window where the game is analysed.
21
22     UI Elements:
23         board_widget_placeholder (QWidget): The placeholder for the board widget.
24         black_button (QPushButton): The button that lets the user override the color of a cell to
       black.
25         white_button (QPushButton): The button that lets the user override the color of a cell to
       white.
26         black_white_button (QPushButton): The button that lets the user play normally with black and
        white alternating.
27         evaluation_button (QPushButton): The button that shows the evaluation of the current board.
28         continuation_button (QPushButton): The button that shows the best continuation of the game.
29         return_to_game_button (QPushButton): The button that returns to the latest move.
30         game_combo_box (QComboBox): The combo box that selects the ruleset of the game.
31         comp_combo_box (QComboBox): The combo box that selects the computer player.
32         exit_button (QPushButton): The button that exits the window.
33         status_bar (QStatusBar): The status bar that shows messages.
34         bottom_text (QLabel): The label that shows the position of the cursor.
35         sidebar (QWidget): The widget that shows the moves made in the game.
36         black_player_name (QLabel): The label that shows the name of the black player.
37         white_player_name (QLabel): The label that shows the name of the white player.
38         analysis_text (QLabel): The label that shows the analysis mode.
39
40     Attributes:
41         _board_widget (BoardWidget): The widget that displays the board.
42         _copmuter_player (ComputerPlayer): The computer player selected.
43         _accepting_input (bool): Whether the player can make a move.
44         _override (bool): Whether the player can override the color of a cell.
45         _color (int): The color of the player whose turn it is. 1 for black, 2 for white.
46         _ended (bool): Whether the game has ended.
47         _selected_move_number (int): The move number that is currently selected.
48         _moves (List[Tuple[int, int]]): A list of the moves made in the game.
49         _renju (bool): Whether renju rules are enabled.
50         _analysing (bool): Whether the game is being analysed.
51
52     Methods:
53         cell_hovered: Called when a cell of board_widget is hovered over.
54         cell_pressed: Called when a cell of board_widget is pressed.
55         _show_message: Shows a message in the status bar.
56         _return_to_game: Returns to the latest move.
```

```
        _reset: Resets itself.
        _make_move: Makes a move on the board.
        _update_sidebar: Updates the sidebar with the move that was made.
        _goto_move: Goes to a move in the game.
        _get_continuation: Gets the best continuation of the game determined by the seleted computer player.
        _show_evaluation: Shows the evaluation of the current board determined by the selected computer player.
        _end_game: Called when the game ends.
        _save_game: Saves the game to a file.
        _load_game: Loads a game from a file.
    """
    def __init__(self, parent: MainScreen):
        """Initialises the PlayWindow.

        Args:
            parent (MainScreen): The parent MainScreen widget.
        """
        super().__init__()

        # Initialise GUI elements
        uic.loadUi('ui/analysis_window.ui', self)

        self.board_widget = BoardWidget()
        self.left_layout.replaceWidget(self.board_widget_placeholder, self.board_widget)

        self.save_action.triggered.connect(self._save_game)
        self.load_action.triggered.connect(self._load_game)

        self.exit_button.pressed.connect(parent.remove_window)

        def black_pressed():
            self._accepting_input = True
            self._override = True
            self.analysing = True
            self._color = 1
        self.black_button.pressed.connect(black_pressed)

        def white_pressed():
            self._accepting_input = True
            self._override = True
            self.analysing = True
            self._color = 2
        self.white_button.pressed.connect(white_pressed)

        def black_white_pressed():
            self._override = False
            if self._ended:
                self._accepting_input = False
            else:
                self._accepting_input = True
        self.black_white_button.pressed.connect(black_white_pressed)

        self.evaluation_button.pressed.connect(self._show_evaluation)

        self.continuation_button.pressed.connect(self._get_continuation)

        self.return_to_game_button.pressed.connect(self._return_to_game)

        self._renju = False
        def game_combo_box_changed(index):
            self._renju = index == 1
            self.status_bar.clearMessage()
        self.game_combo_box.currentIndexChanged.connect(game_combo_box_changed)

        self.comp_combo_box.addItems([player.name for player in COMPUTER_PLAYERS])
        self.comp_combo_box.setCurrentIndex(10)
        def change_comp_player(index):
            self._computer_player = COMPUTER_PLAYERS[index]
        self.comp_combo_box.currentIndexChanged.connect(change_comp_player)

        # Initialise AnalysisWindow attributes
        self._computer_player = COMPUTER_PLAYERS[10]
        self._threadpool = QThreadPool()
        self._accepting_input = True
        self._override = False
```

```
131         self._color = 1
132         self._ended = False
133         self._selected_move_number = -1
134         self._moves = []
135         self._analysing = False
136         self.analysis_text.hide()
137         self._player_names = ("Black", "White")
138 
139     @property
140     def analysing(self):
141         return self._analysing
142 
143     @analysing.setter
144     def analysing(self, b):
145         if b:
146             self.analysis_text.show()
147         else:
148             self.analysis_text.hide()
149         self._analysing = b
150 
151     def cell_hovered(self, row: int, col: int):
152         """Called when a cell is hovered over. It updates the bottom text.
153 
154         Args:
155             row (int): The row of the cell.
156             col (int): The column of the cell.
157         """
158         bottom_text = f"{chr(row + ord('A'))}{col+1}"
159         if DEBUG:
160             bottom_text += f"({row}, {col})"
161         self.bottom_text.setText(bottom_text)
162         board_cell = self.board_widget.board_cells[row][col]
163         if self._accepting_input and board_cell.state == 0:
164             board_cell.change_pixmap(self._color + 2)
165 
166     def cell_pressed(self, row, col):
167         """Called when a cell is pressed. It makes a move if it is legal.
168 
169         Args:
170             row (int): The row of the cell.
171             col (int): The column of the cell.
172         """
173         if not self._accepting_input:
174             return
175         self.status_bar.clearMessage()
176         if self._override:
177             # if the cell is already the color of the player, remove it
178             # otherwise, change it to a cell of the selected color
179             if self.board_widget.board[row][col] == self._color:
180                 self.board_widget.make_move(row, col, 0)
181             else:
182                 self.board_widget.make_move(row, col, self._color)
183             return
184         if self.board_widget.board[row][col] != 0:
185             return
186         # place the piece in a board evaluator to check if it is legal
187         board_evaluator = BoardEvaluator(self.board_widget.board, self._color, row, col, self._renju
    )
188         legal, open_three, four, five, overline = board_evaluator.is_legal_detailed()
189         if not legal:
190             if overline:
191                 self._show_message("Illegal, it is an overline.")
192             elif open_three >= 2:
193                 self._show_message("Illegal, it is a 3x3 fork.")
194             elif four >= 2:
195                 self._show_message("Illegal, it is a 4x4 fork.")
196             else: # should be unreachable
197                 self._show_message("Unknown illegal move.")
198             return
199         self._make_move((row, col))
200 
201     def _show_message(self, message: str):
202         """Shows a message in the status bar.
203 
204         Args:
205             message (str): The message to be shown.
```

```
206         """
207         self.status_bar.showMessage(message)
208         print(message)
209 
210     def _return_to_game(self):
211         move_number = len(self._moves) - 1
212         if move_number >= 0:
213             self.sidebar.layout().itemAtPosition(move_number // 2 + 1, move_number % 2 + 1).widget()
    .setChecked(True)
214         self._goto_move(move_number)
215 
216     def _reset(self):
217         """Resets the AnalysisWindow."""
218         self._accepting_input = True
219         self._override = False
220         self._color = 1
221         self._ended = False
222         self._selected_move_number = -1
223         self._moves = []
224         self._analysing = False
225         self.analysis_text.hide()
226         self.board_widget.reset()
227         for i in reversed(range(3, self.sidebar.layout().count())):
228             self.sidebar.layout().itemAt(i).widget().deleteLater()
229 
230     def _make_move(self, pos: Tuple[int, int]):
231         """Makes a move on the board.
232 
233         Args:
234             pos (Tuple[int, int]): The position of the move.
235         """
236         self.status_bar.clearMessage()
237         row, col = pos
238         self.board_widget.make_move(row, col, self._color)
239         if not self.analysing:
240             self._moves.append((row, col))
241             self._update_sidebar(row, col)
242         # check if the game has ended
243         board_evaluator = BoardEvaluator(self.board_widget.board, self._color, renju=self._renju)
244         if (win := board_evaluator.find_five(self._color)) or board_evaluator.is_full():
245             str_color = "Black" if self._color == 1 else "White"
246             if not self.analysing: # this block is placed before show message, because this calling
    return to game clears message
247                 self._return_to_game() # go to the newly placed move
248             if win:
249                 self._show_message(f"{str_color} wins with a five in a row.")
250             else:
251                 self._show_message("The game is a draw.")
252             if not self.analysing:
253                 self._end_game()
254                 return
255         # if the game has not ended, change the turn
256         if not self.analysing:
257             self._return_to_game() # go to the newly placed move (which changes the turn)
258         else:
259             self._color = 3 - self._color
260 
261     def _update_sidebar(self, row: int, col: int):
262         """Updates the sidebar with the move that was made.
263 
264         Args:
265             row (int): The row of the move.
266             col (int): The column of the move.
267         """
268         cur_move_number = len(self._moves) - 1
269         # add a button that when pressed, goes to this move
270         button = QPushButton(f"{chr(row + ord('A'))}{col+1}")
271         button.pressed.connect(lambda: self._goto_move(cur_move_number))
272         button.setCheckable(True)
273         self.sidebar.layout().addWidget(button, cur_move_number // 2 + 1, self._color)
274         if self._color == 1:
275             # add a new row if the move is from black
276             label = QLabel(f"{cur_move_number // 2 + 1}.")
277             self.sidebar.layout().addWidget(label, cur_move_number // 2 + 1, 0)
278 
279     def _goto_move(self, move_number: int):
```

```
280         """Goes to a move in the game. Called when a move is selected in the sidebar.
281 
282         Args:
283             move_number (int): The move number to go to.
284         """
285         self.status_bar.clearMessage()
286         self._override = False
287         if self._selected_move_number >= 0:
288             self.sidebar.layout().itemAtPosition(self._selected_move_number // 2 + 1, self.
    _selected_move_number % 2 + 1).widget().setChecked(False)
289         self._selected_move_number = move_number
290         self._color = (move_number+1) % 2 + 1
291         # turn on analysis mode if the move is not the last move and vice versa
292         if move_number == len(self._moves) - 1:
293             self.analysing = False
294             if self._ended:
295                 self._accepting_input = False
296         else:
297             self._accepting_input = True
298             self.analysing = True
299         # reset the board and make the moves up to the selected move
300         self.board_widget.reset()
301         for i in range(move_number + 1):
302             row, col = self._moves[i]
303             self.board_widget.make_move(row, col, 1 + i % 2)
304             if i == move_number:
305                 self.board_widget.board_cells[row][col].set_selected()
306 
307     def _get_continuation(self):
308         """Gets the best continuation of the game determined by the seleted computer player."""
309         self._show_message(f"{self._computer_player.name} is thinking...")
310         worker = Worker(self._computer_player.get_continuation, # use polymorphism to get the
    continuation
311                         self.board_widget.board, self._color, self._renju)
312         def show_continuation(moves: List[Tuple[int, int]]):
313             s = ' '.join([f"{chr(move[0] + ord('A'))}{move[1]+1}" for move in moves])
314             self._show_message(f"Continuation: {s}")
315         worker.signals.result.connect(show_continuation) # when worker finishes, show the
    continuation
316         self._threadpool.start(worker)
317 
318     def _show_evaluation(self):
319         """Shows the evaluation of the current board."""
320         self._show_message(f"{self._computer_player.name} is thinking...")
321         worker = Worker(self._computer_player.get_evaluation, # use polymorphism to get the
    evaluation
322                         self.board_widget.board, self._color, self._renju)
323         worker.signals.result.connect(
324             lambda x: self._show_message(generate_evaluation_message(x))) # when worker finishes,
    show the evaluation
325         self._threadpool.start(worker)
326 
327     def _end_game(self):
328         """Ends the game."""
329         self._ended = True
330         self._accepting_input = False
331         # for i, move in enumerate(self.moves):
332         #     print(move, end=' 'if i % 2 == 0 else '\n')
333         # print()
334 
335     def _save_game(self):
336         save_game_dialog = QDialog(self)
337         uic.loadUi('ui/save_game.ui', save_game_dialog)
338         save_game_dialog.setWindowTitle("Save Game")
339         save_game_dialog.setModal(True)
340         save_game_dialog.black_player_name.setText(self._player_names[0])
341         save_game_dialog.white_player_name.setText(self._player_names[1])
342 
343         def save_pressed():
344             # retrieve information from the input and save the game to a json file
345             black_player_name = save_game_dialog.black_player_name.text()
346             white_player_name = save_game_dialog.white_player_name.text()
347             current_time = datetime.now().strftime('%Y%m%d')
348             file_name = f'{current_time}_{black_player_name}_vs_{white_player_name}'
349             file_content = {
350                 'black_player_name': black_player_name,
```

```
                'white_player_name': white_player_name,
                'moves': self._moves,
                'ended': self._ended,
                'win': self._color if self._ended else 0,
            }
            with open(f'games/{file_name}.json', 'w') as file:
                file.write(json.dumps(file_content))
            save_game_dialog.close()
        save_game_dialog.save_button.pressed.connect(save_pressed)

        save_game_dialog.show()

    def _load_game(self):
        # show file dialog and wait for the user to choose a file
        filename = QFileDialog.getOpenFileName(
            self,
            "Load Game",
            "./games",
            "JSON Files (*.json)",
        )[0]
        if not filename: # load game was cancelled
            return
        game_content = json.load(open(filename, 'r'))
        try:
            # load in the game content
            self._player_names = (game_content['black_player_name'], game_content['white_player_name
    '])
            self._reset()
            self.black_player_name.setText(self._player_names[0])
            self.white_player_name.setText(self._player_names[1])
            self._ended = game_content['ended']
            for move in game_content['moves']:
                self._make_move((move[0], move[1]))
        except KeyError:
            # the file is not a valid game file
            self._show_message("Invalid game file.")
            return
        except Exception as e:
            self._show_message(f"Unexpected error while loading file.")
            print(e)
            return
```

3.8 BoardWidget

board_widget.py

```
from typing import List
from PyQt6.QtWidgets import *
from PyQt6.QtCore import *

from board_cell import BoardCell
from constants import BOARD_SIZE

class BoardWidget(QWidget):
    """The widget that displays the board.

    Attributes:
        board_cells (List[List[BoardCell]]): A list of the cells in the board.
        board (List[List[int]]): A list of the colors of the cells in the board. 0 for empty, 1 for
    black, 2 for white.

    Methods:
        make_move: Makes a move on the board.
        reset: Resets the board.
    """
    def __init__(self):
        """Initialises the BoardWidget."""
        super().__init__()

        self.board_cells: List[List[BoardCell]] = [[None for _ in range(BOARD_SIZE)] for _ in range(
    BOARD_SIZE)]
        self.board = [[0 for _ in range(BOARD_SIZE)] for _ in range(BOARD_SIZE)]

        # add corners
        self.board_cells[0][0] = BoardCell(0, 0, 2, 0)
```

```
        self.board_cells[0][BOARD_SIZE - 1] = BoardCell(0, BOARD_SIZE - 1, 2, 1)
        self.board_cells[BOARD_SIZE - 1][BOARD_SIZE - 1] = BoardCell(BOARD_SIZE - 1, BOARD_SIZE - 1, 2, 2)
        self.board_cells[BOARD_SIZE - 1][0] = BoardCell(BOARD_SIZE - 1, 0, 2, 3)

        # add edges
        for i in range(1, BOARD_SIZE - 1):
            self.board_cells[0][i] = BoardCell(0, i, 1, 0)
        for i in range(1, BOARD_SIZE - 1):
            self.board_cells[i][BOARD_SIZE -
                               1] = BoardCell(i, BOARD_SIZE - 1, 1, 1)
        for i in range(1, BOARD_SIZE - 1):
            self.board_cells[BOARD_SIZE -
                             1][i] = BoardCell(BOARD_SIZE - 1, i, 1, 2)
        for i in range(1, BOARD_SIZE - 1):
            self.board_cells[i][0] = BoardCell(i, 0, 1, 3)

        # add centres
        for i in range(1, BOARD_SIZE - 1):
            for j in range(1, BOARD_SIZE - 1):
                self.board_cells[i][j] = BoardCell(i, j, 0, 0)

        board_layout = QGridLayout(self)
        # add the cells in board_cells to layout
        for i in range(BOARD_SIZE):
            for j in range(BOARD_SIZE):
                board_layout.addWidget(self.board_cells[i][j], i+1, j+1)

        board_layout.setSpacing(0)
        board_layout.setRowStretch(0, 1)
        board_layout.setRowStretch(BOARD_SIZE + 1, 1)
        board_layout.setColumnStretch(0, 1)
        board_layout.setColumnStretch(BOARD_SIZE + 1, 1)

    def make_move(self, row: int, col: int, color: int):
        """Makes a move on the board.

        Args:
            row (int): The row of the move.
            col (int): The column of the move.
            color (int): The color of the move. 1 for black, 2 for white.
        """
        self.board[row][col] = color
        self.board_cells[row][col].change_pixmap(color)

    def reset(self):
        """Resets the board."""
        for i in range(BOARD_SIZE):
            for j in range(BOARD_SIZE):
                self.board[i][j] = 0
                self.board_cells[i][j].change_pixmap(0)
```

3.9 BoardCell

board_cell.py

```
from PyQt6.QtWidgets import QLabel
from PyQt6.QtGui import QPixmap, QTransform
from PyQt6.QtCore import Qt

class BoardCell(QLabel):
    """A cell in the board.

    Attributes:
        _row (int): The row of the cell.
        _col (int): The column of the cell.
        _pixmaps (List[QPixmap]): A list of the pixel maps of the cell.
        state (int): The current index of the pixel map shown.

    Overridden Methods:
        enterEvent: Called when the mouse enters the cell.
        leaveEvent: Called when the mouse leaves the cell.
        mousePressEvent: Called when the cell is pressed.

    Methods:
```

```
        change_pixmap: Changes the pixmap of the cell.
        set_selected: Sets the cell as selected.
        set_deselected: Sets the cell as deselected.
    """
    def __init__(self, row: int, col: int, cell_type: int, rotation: int):
        """Initialises the BoardCell.

        Args:
            row (int): The row of the cell.
            col (int): The column of the cell.
            cell_type (int): The type of the cell. 0 for centre, 1 for edge, 2 for corner.
            rotation (int): The rotation of the cell. 0 for 0 degrees, 1 for 90 degrees, 2 for 180 degrees, 3 for 270 degrees.
        """
        super().__init__()

        self._row = row
        self._col = col
        self._pixmaps = [None] * 7

        if cell_type == 0:
            self._pixmaps[0] = QPixmap("assets/centre.png")
            self._pixmaps[1] = QPixmap("assets/centre_black.png")
            self._pixmaps[2] = QPixmap("assets/centre_white.png")
            self._pixmaps[3] = QPixmap("assets/centre_black_alpha.png")
            self._pixmaps[4] = QPixmap("assets/centre_white_alpha.png")
            self._pixmaps[5] = QPixmap("assets/centre_black_selected.png")
            self._pixmaps[6] = QPixmap("assets/centre_white_selected.png")
        elif cell_type == 1:
            self._pixmaps[0] = QPixmap("assets/edge.png")
            self._pixmaps[1] = QPixmap("assets/edge_black.png")
            self._pixmaps[2] = QPixmap("assets/edge_white.png")
            self._pixmaps[3] = QPixmap("assets/edge_black_alpha.png")
            self._pixmaps[4] = QPixmap("assets/edge_white_alpha.png")
            self._pixmaps[5] = QPixmap("assets/edge_black_selected.png")
            self._pixmaps[6] = QPixmap("assets/edge_white_selected.png")
        elif cell_type == 2:
            self._pixmaps[0] = QPixmap("assets/corner.png")
            self._pixmaps[1] = QPixmap("assets/corner_black.png")
            self._pixmaps[2] = QPixmap("assets/corner_white.png")
            self._pixmaps[3] = QPixmap("assets/corner_black_alpha.png")
            self._pixmaps[4] = QPixmap("assets/corner_white_alpha.png")
            self._pixmaps[5] = QPixmap("assets/corner_black_selected.png")
            self._pixmaps[6] = QPixmap("assets/corner_white_selected.png")

        for i in range(7):
            self._pixmaps[i] = self._pixmaps[i].transformed(QTransform().rotate(rotation*90))
            self._pixmaps[i] = self._pixmaps[i].scaled(42, 42, Qt.AspectRatioMode.KeepAspectRatio)

        self.state = 0
        self.setPixmap(self._pixmaps[self.state])

    def enterEvent(self, ev) -> None:
        """Called when the mouse enters the cell. It calls the cell_hovered method of the current window."""
        self.window().currentWidget().cell_hovered(self._row, self._col)

    def leaveEvent(self, ev) -> None:
        """Called when the mouse leaves the cell. It changes the pixmap from a ghost piece to an empty cell."""
        if self.state == 3 or self.state == 4:
            self.change_pixmap(0)

    def mousePressEvent(self, ev) -> None:
        """Called when the cell is pressed. It calls the cell_pressed method of the current window.
        """"""
        self.window().currentWidget().cell_pressed(self._row, self._col)

    def change_pixmap(self, index: int):
        """Changes the pixmap of the cell.

        Args:
            index (int): The index of the pixmap to change to.
        """
        self.state = index
        self.setPixmap(self._pixmaps[index])
```

```
92
93      def set_selected(self):
94          """Sets the cell as selected."""
95          if self.state == 1 or self.state == 3:
96              self.change_pixmap(5)
97          elif self.state == 2 or self.state == 4:
98              self.change_pixmap(6)
99
100     def set_deselcted(self):
101         """Sets the cell as deselected."""
102         if self.state == 5:
103             self.change_pixmap(1)
104         elif self.state == 6:
105             self.change_pixmap(2)
```

3.10 BoardEvaluator

board_evaluator.py

```
1  from __future__ import annotations
2
3  from copy import deepcopy
4  from typing import List, Tuple, Iterator
5
6  from utils import next_k
7  from constants import BOARD_SIZE
8
9  class BoardEvaluator:
10     """Evaluates a board.
11
12     Attributes:
13         board (List[List[int]]): The board to evaluate.
14         partial_count (List[List[List[List[int]]]]): A list of partial counts of the board.
15         row (int): The row of the current move.
16         col (int): The column of the current move.
17         color (int): The color of the player whose turn it is. 1 for black, 2 for white.
18         renju (bool): Whether renju rules are enabled.
19
20     Methods for partial sum and coordinate conversion: (Refer to Section 2.2.1 of the report for
       more details)
21         cartesian_to_line: Converts cartesian coordinates to line coordinates.
22         line_to_cartesian: Converts line coordinates to cartesian coordinates.
23         get_max_a: Gets the maximum a coordinate for a given direction.
24         get_max_b: Gets the maximum b coordinate for a given direction and a coordinate.
25         _get_partial_count: Gets the partial count of the board.
26         range_query: Queries how many stones of a color starting from a  point given in line
       coordinates with a given range on a given line.
27         range_query_cartesian: Queries how many stones of a color starting from a point given in
       cartesian coordinates with a given range on a given line.
28
29     Methods for checking if a move is legal: (Refer to Section 2.2.6 of the report for more details)
30         is_overline: Checks if the current move creates an overline.
31         is_five: Checks if the current move creates a five.
32         count_open_three: Counts the number of open threes created by the current move.
33         count_four: Counts the number of fours created by the current move.
34         count_ends_four: Counts the number of open ends of a four created by the current move.
35         is_legal_detailed: Checks if the current move is legal, and returns the number of threes and
        fours created by the move.
36         is_legal: Checks if the current move is legal.
37
38     Methods for finding threats and board evaluation: (Refer to Section 2.2.2 and 2.2.3 of the
       report for more details)
39         find_five: Checks if there is a five of a given color.
40         find_four: Checks if there is a four of a given color.
41         find_open_three: Checks if there is an open three of a given color.
42         find_open_two: Checks if there is an open two of a given color.
43         find_any_three: Checks if there is any three of a given color.
44         count_three: Counts the number of threes of a given color.
45         count_two: Counts the number of twos of a given color.
46         count_one: Counts the number of ones of a given color.
47         get_search_space: Gets the search space of the board.
48         is_empty: Checks if the board is empty.
49         is_full: Checks if the board is full.
50     """
```

```
51      def __init__(self, board: List[List[int]], color: int, row: int=None, col: int=None, renju: bool
        =True):
52          """Initialises the BoardEvaluator.
53
54          Args:
55              board (List[List[int]]): The board to evaluate.
56              color (int): The color of the player whose turn it is. 1 for black, 2 for white.
57              row (int, optional): The row of the current move. Defaults to None.
58              col (int, optional): The column of the current move. Defaults to None.
59              renju (bool, optional): Whether renju rules are enabled. Defaults to True.
60          """
61          self.board = deepcopy(board)
62          self.partial_count = None
63          self.row = row
64          self.col = col
65          self.color = color
66          self.renju = renju
67          if row != None and col != None:
68              self.board[row][col] = color
69
70      """Methods for partial sum and coordinate conversion:"""
71
72      def cartesian_to_line(self, direction: int, row: int, col: int) -> Tuple[int, int]:
73          """Converts cartesian coordinates to line coordinates.
74
75          Args:
76              direction (int): The direction of the line.
77              row (int): The row of the cell.
78              col (int): The column of the cell.
79
80          Returns:
81              Tuple[int, int]: The converted line coordinates.
82          """
83          if direction == 0:
84              a = row
85              b = col
86          elif direction == 1:
87              a = col
88              b = row
89          elif direction == 2:
90              a = col - row + (BOARD_SIZE - 1)
91              b = min(row, col)
92          elif direction == 3:
93              a = row + col
94              b = row - max(a - (BOARD_SIZE - 1), 0)
95          return (a, b)
96
97      def line_to_cartesian(self, direction: int, a: int, b: int) -> Tuple[int, int]:
98          """Converts line coordinates to cartesian coordinates.
99
100         Args:
101             direction (int): The direction of the line.
102             a (int): The a coordinate of the cell.
103             b (int): The b coordinate of the cell.
104
105         Raises:
106             ValueError: If b is out of range.
107
108         Returns:
109             Tuple[int, int]: The converted cartesian coordinates.
110         """
111         if b >= self.get_max_b(direction, a) or b < 0:
112             raise ValueError
113         if direction == 0:
114             return (a, b)
115         elif direction == 1:
116             return (b, a)
117         elif direction == 2:
118             d = max(a - (BOARD_SIZE - 1), 0)
119             return (BOARD_SIZE - a - 1 + d + b, b + d)
120         elif direction == 3:
121             d = max(a - (BOARD_SIZE - 1), 0)
122             return (d + b, a - b - d)
123
124     def get_max_a(self, direction: int) -> int:
125         """Gets the maximum a coordinate for a given direction.
```

```
126
127         Args:
128             direction (int): The direction of the line.
129
130         Returns:
131             int: The maximum a coordinate.
132         """
133         if direction == 0 or direction == 1:
134             return BOARD_SIZE
135         elif direction == 2 or direction == 3:
136             return 2*BOARD_SIZE - 1
137
138     def get_max_b(self, direction: int, a: int) -> int:
139         """Gets the maximum b coordinate for a given direction and a coordinate.
140
141         Args:
142             direction (int): The direction of the line.
143             a (int): The a coordinate of the cell.
144
145         Returns:
146             int: The maximum b coordinate.
147         """
148         if direction == 0 or direction == 1:
149             return BOARD_SIZE
150         elif direction == 2 or direction == 3:
151             return BOARD_SIZE - abs((BOARD_SIZE-1) - a)
152
153     def _get_partial_count(self):
154         """Computes the partial count of the board."""
155         self.partial_count = [[[[0 for _ in range(
156             BOARD_SIZE + 1)] for _ in range(2*BOARD_SIZE - 1)]for _ in range(3)] for _ in range(4)]
157
158         for direction in range(4):
159             for a in range(self.get_max_a(direction)):
160                 for b in range(self.get_max_b(direction, a)):
161                     row, col = self.line_to_cartesian(direction, a, b)
162                     for k in range(3):
163                         self.partial_count[direction][k][a][b+1] = self.partial_count[direction][k][
    a][b] + \
164                             (1 if self.board[row][col] == k else 0)
165
166     def range_query(self, direction: int, a: int, b: int, color: int, length: int) -> int:
167         """Queries how many stones of a color starting from a point given in line coordinates with a
     given length on a given line.
168
169         Args:
170             direction (int): The direction of the line.
171             a (int): The a coordinate of the cell.
172             b (int): The b coordinate of the cell.
173             color (int): The color of the stones.
174             length (int): The length of the query.
175
176         Raises:
177             IndexError: If the point (direction, a, b + length) is out of bounds.
178
179         Returns:
180             int: The number of stones of the given color.
181         """
182         if self.partial_count == None:
183             self._get_partial_count()
184         if b + length > self.get_max_b(direction, a):
185             raise IndexError
186         return self.partial_count[direction][color][a][b + length] - self.partial_count[direction][
    color][a][b]
187
188     def range_query_cartesian(self, direction: int, row: int, col: int, color: int, length: int) ->
    int:
189         """Queries how many stones of a color starting from a point given in cartesian coordinates
    with a given length on a given line.
190
191         Args:
192             direction (int): The direction of the line.
193             row (int): The row of the cell.
194             col (int): The column of the cell.
195             color (int): The color of the stones.
196             length (int): The length of the query.
```

```
197
198         Returns:
199             int: The number of stones of the given color.
200         """
201         a, b = self.cartesian_to_line(direction, row, col)
202         return self.range_query(direction, a, b, color, length)
203
204     """Methods for checking if a move is legal:"""
205
206     def is_overline(self) -> bool:
207         """Checks if the current move creates an overline.
208
209         Returns:
210             bool: Whether the current move creates an overline.
211         """
212         overline = False
213         for direction in range(4):
214             a, b = self.cartesian_to_line(direction, self.row, self.col)
215             start_b = max(b-5, 0)
216             end_b = min(b+1, self.get_max_b(direction, a) - 5)
217             for ib in range(start_b, end_b):
218                 if self.range_query(direction, a, ib, self.color, 6) == 6:
219                     overline = True
220         return overline
221
222     def is_five(self) -> bool:
223         """Checks if the current move creates a five.
224
225         Returns:
226             bool: Whether the current move creates a five.
227         """
228         five = False
229         for direction in range(4):
230             a, b = self.cartesian_to_line(direction, self.row, self.col)
231             start_b = max(b-4, 0)
232             end_b = min(b+1, self.get_max_b(direction, a) - 4)
233             for ib in range(start_b, end_b):
234                 if self.range_query(direction, a, ib, self.color, 5) == 5:
235                     five = True
236         return five
237
238     def count_current_open_three(self) -> int:
239         """Counts the number of open threes created by the current move.
240
241         Returns:
242             int: The number of open threes created by the current move.
243         """
244         num_threes = 0
245         for direction in range(4):
246             a, b = self.cartesian_to_line(direction, self.row, self.col)
247             # iterate through all possible positions of the an open three (of length 6)
248             start_b = max(b-4, 0) # start at 4 spaces before, or 0, whichever is larger
249             end_b = min(b, self.get_max_b(direction, a) - 5) # end at a space before b, or the
    maximum b, whichever is smaller
250             for ib in (iterator := iter(range(start_b, end_b))):
251                 left_0 = self.range_query(direction, a, ib, 0, 1)
252                 right_0 = self.range_query(direction, a, ib + 5, 0, 1)
253                 middle_color = self.range_query(direction, a, ib + 1, self.color, 4)
254                 middle_0 = self.range_query(direction, a, ib + 1, 0, 4)
255                 if left_0 == 1 and right_0 == 1 and middle_color == 3 and middle_0 == 1:
256                     # found a potential open three
257                     # find b_empty, the b of the cell of the potential straight four that is empty
258                     for b2 in range(ib + 1, ib + 5):
259                         result = self.range_query(direction, a, b2, 0, 1)
260                         if result == 1:
261                             b_empty = b2
262                             break
263                     # place the empty cell and check if it is legal
264                     board_evaluator = BoardEvaluator(
265                         self.board, self.color, *self.line_to_cartesian(direction, a, b_empty), self
    .renju)
266                     if not board_evaluator.is_legal():
267                         continue
268                     # a legal move can be placed to create a potential straight four
269                     ends = board_evaluator.count_ends_four(direction, a, ib)
270                     if ends == 2: # new shape is a straight four, so it is an open three
```

```
                        num_threes += 1
                        next_k(iterator, 1) # advance iterator to avoid double counting
        return num_threes

    def count_current_four(self) -> int:
        """Counts the number of fours created by the current move.

        Returns:
            int: The number of fours created by the current move.
        """
        num_fours = 0
        for direction in range(4):
            a, b = self.cartesian_to_line(direction, self.row, self.col)
            start_b = max(b-4, 0) # start at 4 spaces before, or 0, whichever is larger
            end_b = min(b+1, self.get_max_b(direction, a) - 4) # end at b, or the maximum b, whichever is smaller
            for ib in (iterator := iter(range(start_b, end_b))):
                count_color = self.range_query(direction, a, ib, self.color, 5)
                count_0 = self.range_query(direction, a, ib, 0, 5)
                if count_color == 4 and count_0 == 1:
                    # find b_empty, the b of the cell of the potential five that is empty
                    for b2 in range(ib, ib + 5):
                        result = self.range_query(direction, a, b2, 0, 1)
                        if result == 1:
                            b_empty = b2
                            break
                    # place the empty cell and check if it is legal
                    board_evaluator = BoardEvaluator(
                        self.board, self.color, *self.line_to_cartesian(direction, a, b_empty), self.renju)
                    if not board_evaluator.is_legal():
                        continue
                    # it is legal, so the shape is a four
                    num_fours += 1
                    next_k(iterator, 1) # advance iterator to avoid double counting
        return num_fours

    def count_ends_four(self, direction: int, a: int, b: int) -> int:
        """Counts the number of open ends of a four defined by (direction, a, b+1) to (direction, a, b+4).

        Args:
            direction (int): The direction of the line.
            a (int): The a coordinate of the cell.
            b (int): The b coordinate of the cell.

        Returns:
            int: The number of open ends of the four.
        """
        if not self.renju:
            return 2
        # It is guaranteed that (direction, a, b) and (direction, a, b+5) is empty
        board_evaluator1 = BoardEvaluator(
            self.board, self.color, *self.line_to_cartesian(direction, a, b), self.renju)
        board_evaluator2 = BoardEvaluator(
            self.board, self.color, *self.line_to_cartesian(direction, a, b+5), self.renju)
        count = 0
        count += 1 if board_evaluator1.is_legal() else 0
        count += 1 if board_evaluator2.is_legal() else 0
        return count

    def is_legal_detailed(self) -> Tuple[bool, int, int, bool, bool]:
        """Checks if the current move is legal, and returns with details.

        Returns:
            Tuple[bool, int, int, bool, bool]: A tuple containing:
                whether the move is legal,
                the number of open threes created by this move (if no fives or overlines are created),
                the number of fours created by this move (if no fives or overlines are created),
                whether the move creates a five,
                and whether the move creates an overline.
        """
        overline = self.is_overline()
        if self.renju and (self.color == 1 and overline): # black creates an overline
            return (False, 0, 0, False, True)
```

```
343         if self.is_five(): # winning is always legal if no overlines are created
344             return (True, 0, 0, True, overline)
345         open_three = self.count_current_open_three()
346         four = self.count_current_four()
347         if self.renju and self.color == 1 and (open_three >= 2 or four >= 2): # black plays a 3x3 or
     4x4 fork
348             return (False, open_three, four, False, overline)
349         return (True, open_three, four, False, overline)
350 
351     def is_legal(self) -> bool:
352         """Checks if the current move is legal.
353 
354         Returns:
355             bool: Whether the current move is legal.
356         """
357         if not self.renju or self.color == 2: # In Gomoku, all moves are legal. In Renju, white
    moves are always legal.
358             return True
359         return self.is_legal_detailed()[0]
360 
361     """Methods for finding threats and board evaluation:"""
362 
363     def find_five(self, color: int) -> bool:
364         """Checks if there is a five of a given color.
365 
366         Args:
367             color (int): The color of the five.
368 
369         Returns:
370             bool: Whether there is a five of the given color.
371         """
372         for direction in range(4):
373             for a in range(self.get_max_a(direction)):
374                 for b in range(self.get_max_b(direction, a) - 4):
375                     result = self.range_query(direction, a, b, color, 5)
376                     if result == 5:
377                         return True
378         return False
379 
380     def count_straight_four(self, color: int) -> bool:
381         """Counts the number of straight fours of a given color.
382 
383         Args:
384             color (int): The color of the straight fours.
385 
386         Returns:
387             bool: The number of straight fours of the given color.
388         """
389         # Straight fours are a line of 6 cells with 4 consecutive stones of the given color with
    empty cells at both ends
390         straight_four = 0
391         for direction in range(4):
392             for a in range(self.get_max_a(direction)):
393                 for b in (iterator := iter(range(self.get_max_b(direction, a) - 5))):
394                     count_color = self.range_query(direction, a, b+1, color, 4)
395                     count_0 = self.range_query(direction, a, b, 0, 6)
396                     if count_color == 4 and count_0 == 2:
397                         straight_four += 1
398                         next_k(iterator, 4) # minor optimisation
399         return straight_four
400 
401     def find_four(self, color: int) -> List[BoardEvaluator]:
402         """Find fours of a given color.
403 
404         Args:
405             color (int): The color of the fours.
406 
407         Returns:
408             List[BoardEvaluator]: A list of BoardEvaluators that represent the fours with the
    remaining empty cell played.
409         """
410         # a line of 5 spaces with 4 stone with the given color and 1 empty space
411         fours = []
412         for direction in range(4):
413             for a in range(self.get_max_a(direction)):
414                 for b in range(self.get_max_b(direction, a) - 4):
```

```
                    count_color = self.range_query(direction, a, b, color, 5)
                    count_0 = self.range_query(direction, a, b, 0, 5)
                    if count_color == 4 and count_0 == 1:
                        # a four is found
                        # find b_empty, the b of the cell of the potential five that is empty
                        for b2 in range(b, b + 5):
                            result = self.range_query(
                                direction, a, b2, 0, 1)
                            if result == 1:
                                b_empty = b2
                                break
                        # place the empty cell and check if it is legal
                        board_evaluator = BoardEvaluator(
                            self.board, self.color, *self.line_to_cartesian(direction, a, b_empty), self.renju)
                        if board_evaluator.is_legal():
                            fours.append(board_evaluator)
        return fours

    def find_open_three(self, color: int) -> List[BoardEvaluator]:
        """Find open threes of a given color. They are a type of threat since placing one more stone would create a straight four.

        Args:
            color (int): The color of the threes.

        Returns:
            List[BoardEvaluator]: A list of BoardEvaluators that represent the threes with the remaining empty cell played.
        """
        # a line of four spaces with 3 stones of the given color and 1 empty space, surrounded by empty spaces
        # e.g. .OOO.. or .O.OO.
        threes = []
        for direction in range(4):
            for a in range(self.get_max_a(direction)):
                for b in range(self.get_max_b(direction, a) - 5):
                    left_0 = self.range_query(direction, a, b, 0, 1)
                    right_0 = self.range_query(direction, a, b+5, 0, 1)
                    middle_color = self.range_query(direction, a, b+1, color, 4)
                    middle_0 = self.range_query(direction, a, b+1, 0, 4)
                    if left_0 == 1 and right_0 == 1 and middle_color == 3 and middle_0 == 1:
                        # potential open three found
                        # find b_empty, the b of the cell of the potential straight four that is empty
                        for b2 in range(b + 1, b + 5):
                            result = self.range_query(direction, a, b2, 0, 1)
                            if result == 1:
                                b_empty = b2
                                break
                        # place the empty cell and check if it is legal
                        board_evaluator = BoardEvaluator(
                            self.board, self.color, *self.line_to_cartesian(direction, a, b_empty), self.renju)
                        if not board_evaluator.is_legal():
                            continue
                        # check if the four created is a straight four or not
                        ends = board_evaluator.count_ends_four(direction, a, b)
                        if ends == 2:
                            threes.append(board_evaluator)
        return threes

    def find_open_two(self, color: int) -> List[BoardEvaluator]:
        """Find open twos of a given color. They are a type of threat since placing one more stone would create an open three.

        Args:
            color (int): The color of the twos.

        Returns:
            List[BoardEvaluator]: A list of BoardEvaluators that represent the twos with one of the empty cells played.
        """
        # a line of four spaces with 2 stones of the given color and 2 empty spaces, surrounded by empty spaces
        open_two_boards = []
```

```
        for direction in range(4):
            for a in range(self.get_max_a(direction)):
                for b in (iterator := iter(range(self.get_max_b(direction, a) - 5))):
                    left_0 = self.range_query(direction, a, b, 0, 1)
                    right_0 = self.range_query(direction, a, b+5, 0, 1)
                    middle_color = self.range_query(direction, a, b+1, color, 4)
                    middle_0 = self.range_query(direction, a, b+1, 0, 4)
                    if left_0 == 1 and right_0 == 1 and middle_color == 2 and middle_0 == 2:
                        # potential open two found
                        for b2 in range(b + 1, b + 5):
                            is_0 = self.range_query(direction, a, b2, 0, 1)
                            if is_0 == 1:
                                # current cell is empty, place a stone and check if it is legal
                                board_evaluator = BoardEvaluator(
                                    self.board, self.color, *self.line_to_cartesian(direction, a, b2
    ), self.renju)
                                if not board_evaluator.is_legal():
                                    continue
                                open_two_boards.append(board_evaluator)
                        next_k(iterator, 1) # advance iterator to avoid double counting
        return open_two_boards

    def find_three(self, color: int) -> List[BoardEvaluator]:
        """Find any threes of a given color.

        Args:
            color (int): The color of the threes.

        Returns:
            List[BoardEvaluator]: A list of BoardEvaluators that represent the threes with one of
    the remaining empty cells played.
        """
        # A line of 5 spaces with 3 stones of the given color and 2 empty spaces
        three_boards = []
        for direction in range(4):
            for a in range(self.get_max_a(direction)):
                for b in range(self.get_max_b(direction, a) - 4):
                    count_color = self.range_query(direction, a, b, color, 5)
                    count_0 = self.range_query(direction, a, b, 0, 5)
                    if count_color == 3 and count_0 == 2:
                        for b2 in range(b, b + 5):
                            if self.range_query(direction, a, b2, 0, 1) == 1: # current cell is
    empty
                                board_evaluator = BoardEvaluator(
                                    self.board, self.color, *self.line_to_cartesian(direction, a, b2
    ), self.renju)
                                if board_evaluator.is_legal():
                                    three_boards.append(board_evaluator)
        return three_boards

    def count_three(self, color: int) -> int:
        """Counts the number of threes of a given color.

        Args:
            color (int): The color of the threes.

        Returns:
            int: The number of threes of the given color.
        """
        # A line of 5 spaces with 3 stones of the given color and 2 empty spaces
        three = 0
        for direction in range(4):
            for a in range(self.get_max_a(direction)):
                for b in (iterator := iter(range(self.get_max_b(direction, a) - 4))):
                    count_color = self.range_query(direction, a, b, color, 5)
                    count_0 = self.range_query(direction, a, b, 0, 5)
                    if count_color == 3 and count_0 == 2:
                        three += 1
                        next_k(iterator, 2) # advance iterator to avoid double counting
        return three

    def count_two(self, color: int) -> int:
        """Counts the number of twos of a given color.

        Args:
            color (int): The color of the twos.
```

```
554
555         Returns:
556             int: The number of twos of the given color.
557         """
558         # A line of 5 spaces with 2 stones of the given color and 3 empty spaces
559         two = 0
560         for direction in range(4):
561             for a in range(self.get_max_a(direction)):
562                 for b in (iterator := iter(range(self.get_max_b(direction, a) - 4))):
563                     count_color = self.range_query(direction, a, b, color, 5)
564                     count_0 = self.range_query(direction, a, b, 0, 5)
565                     if count_color == 2 and count_0 == 3:
566                         two += 1
567                         next_k(iterator, 3) # advance iterator to avoid double counting
568         return two
569 
570     def count_one(self, color: int) -> int:
571         """Counts the number of ones of a given color.
572 
573         Args:
574             color (int): The color of the ones.
575 
576         Returns:
577             int: The number of ones of the given color.
578         """
579         # A line of 5 spaces with 1 stone of the given color and 4 empty spaces
580         one = 0
581         for direction in range(4):
582             for a in range(self.get_max_a(direction)):
583                 for b in (iterator := iter(range(self.get_max_b(direction, a) - 4))):
584                     count_color = self.range_query(direction, a, b, color, 5)
585                     count_0 = self.range_query(direction, a, b, 0, 5)
586                     if count_color == 1 and count_0 == 4:
587                         one += 1
588                         next_k(iterator, 4) # advance iterator to avoid double counting
589         return one
590 
591     def get_search_space(self, r: int) -> List[BoardEvaluator]:
592         """Gets the search space of the board. Refer to Section 2.2.2 of the report for more details
    .
593 
594         Args:
595             r (int): The radius of the search space.
596 
597         Returns:
598             List[BoardEvaluator]: A list of BoardEvaluators that represent the boards in the search
    space.
599         """
600         search_space = [[False for _ in range(
601             BOARD_SIZE)] for _ in range(BOARD_SIZE)]
602         for row in range(BOARD_SIZE):
603             for col in range(BOARD_SIZE):
604                 if self.board[row][col] != 0:
605                     for dx in range(-r, r+1):
606                         for dy in range(-r, r+1):
607                             new_row = row + dx
608                             new_col = col + dy
609                             if not (0 <= new_row < BOARD_SIZE and 0 <= new_col < BOARD_SIZE) or self
    .board[new_row][new_col] != 0:
610                                 continue
611                             search_space[new_row][new_col] = True
612         candidate_boards: List[BoardEvaluator] = []
613         for row in range(BOARD_SIZE):
614             for col in range(BOARD_SIZE):
615                 if not search_space[row][col] or self.board[row][col] != 0:
616                     continue
617                 new_board = BoardEvaluator(
618                     self.board, self.color, row, col, self.renju)
619                 if new_board.is_legal():
620                     candidate_boards.append(new_board)
621         return candidate_boards
622 
623     def is_empty(self) -> bool:
624         """Checks if the board is empty.
625 
626         Returns:
```

```
            bool: Whether the board is empty.
        """
        return all(all(cell == 0 for cell in row) for row in self.board)

    def is_full(self) -> bool:
        """Checks if the board is full.

        Returns:
            bool: Whether the board is full.
        """
        return all(0 not in row for row in self.board)

if __name__ == '__main__':
    board = [[0 for _ in range(BOARD_SIZE)] for _ in range(BOARD_SIZE)]
    board[7][8] = board[7][9] = 1
    board[8][7] = board[9][7] = 1
    board_evaluator = BoardEvaluator(board, 1, 7, 7, True) # play move (7, 7) for black
    print(board_evaluator.is_legal_detailed()) # (False, 2, 0, False, False), illegal, 3x3 fork
    board[7][10] = board[10][7] = 1
    board_evaluator = BoardEvaluator(board, 1, 7, 7, True)
    print(board_evaluator.is_legal_detailed()) # (True, 0, 2, False, False), illegal, 4x4 fork
    board[7][10] = 0
    board_evaluator = BoardEvaluator(board, 1, 7, 7, True)
    print(board_evaluator.is_legal_detailed()) # (True, 1, 1, False, False), legal, 3x4 fork
    board[11][7] = 1
    board_evaluator = BoardEvaluator(board, 1, 7, 7, True)
    print(board_evaluator.is_legal_detailed()) # (True, 0, 0, True, False), legal, five
    board[12][7] = 1
    board_evaluator = BoardEvaluator(board, 1, 7, 7, True)
    print(board_evaluator.is_legal_detailed()) # (False, 0, 0, False, True), illegal, overline
```

3.11 Worker

worker.py

```
# Modified from https://www.pythonguis.com/tutorials/multithreading-pyqt6-applications-qthreadpool/
# This file is used solely for the purpose of multithreading in the application.

from PyQt6.QtCore import QObject, QRunnable, pyqtSignal, pyqtSlot

import traceback, sys

class WorkerSignals(QObject):
    '''Defines the signals available from a running worker thread.

    Supported signals are:
        finished: No data
        error: tuple (exctype, value, traceback.format_exc())
        result: object data returned from processing, can be anything
    '''
    finished = pyqtSignal()
    error = pyqtSignal(tuple)
    result = pyqtSignal(object)

class Worker(QRunnable):
    '''The worker thread. Inherits from QRunnable to handler worker thread setup, signals and wrap-up.'''
    def __init__(self, fn, *args, **kwargs):
        super(Worker, self).__init__()

        # Store constructor arguments (reused for processing)
        self.fn = fn
        self.args = args
        self.kwargs = kwargs
        self.signals = WorkerSignals()

    @pyqtSlot()
    def run(self):
        '''Initialise the runner function with passed args, kwargs.'''
        try:
            result = self.fn(*self.args, **self.kwargs)
        except:
            traceback.print_exc()
            exctype, value = sys.exc_info()[:2]
            self.signals.error.emit((exctype, value, traceback.format_exc()))
```

```
40          else:
41              self.signals.result.emit(result)  # Return the result of the processing
42          finally:
43              self.signals.finished.emit()  # Done
```

3.12 Players

3.12.1 Player, HumanPlayer, and ComputerPlayer

players.py

```
1  from typing import Tuple, List
2
3  class Player:
4      """The abstract class for all players.
5
6      Attributes:
7          name (str): The name of the player.
8      """
9      def __init__(self) -> None:
10         self.name = 'Player' # Must be changed when subclassed.
11
12 class ComputerPlayer(Player):
13     """The abstract class for all computer players.
14
15     Methods:
16         get_move: Returns the best move to make on the board.
17         get_continuation: Returns the best continuation to make on the board.
18         get_evaluation: Returns the evaluation of the board.
19     """
20     def __init__(self) -> None:
21         self.name = 'Computer Player' # Must be changed when subclassed.
22         pass
23
24     def get_move(self, board: List[List[int]], color: int, renju: bool) -> Tuple[int, int]:
25         """Returns the best move to make on the board."""
26         pass
27
28     def get_continuation(self, board: List[List[int]], color: int, renju: bool) -> List[Tuple[int,
       int]]:
29         """Returns the best continuation to make on the board."""
30         pass
31
32     def get_evaluation(self, board: List[List[int]], color: int, renju: bool) -> int:
33         """Returns the evaluation of the board."""
34         pass
35
36 class HumanPlayer(Player):
37     """The human player."""
38     def __init__(self) -> None:
39         self.name = 'Human Player'
```

3.12.2 MinimaxPlayer

computer_players/minimax_player.py

```
1  from typing import List, Tuple
2  from random import randint
3
4  from board_evaluator import BoardEvaluator
5  from constants import *
6  from players import ComputerPlayer
7  from utils import previous_of, bubble_sort_k
8
9  class MinimaxPlayer(ComputerPlayer):
10     """A computer player that uses the minimax algorithm to make moves. Refer to Section 2.2.2 of
       the report for more details.
11
12     Attributes:
13         name (str): The name of the player.
14         depth (int): The depth of the search tree.
15         max_branches (int): The maximum number of branches to search.
16
```

```
    Methods:
        local_evaluation: Evaluates the board locally.
        minimax: Applies the minimax algorithm to the board.
        get_evaluation: Gets the evaluation of the board.
        get_move: Gets the best move to make on the board.
        get_continuation: Gets the best continuation to make on the board.
    """
    def __init__(self, depth: int, max_branches: int) -> None:
        """Initialises the MinimaxPlayer.

        Args:
            depth (int): The depth of the search tree.
            max_branches (int): The maximum number of branches to search.
        """
        self.depth = depth
        self.max_branches = max_branches
        self.name = f"Minimax {self.depth}"

    def _local_evaluation(self, board: BoardEvaluator) -> int:
        if board.find_five(1):
            return WIN
        if board.find_five(2):
            return -WIN
        evaluation = 0
        evaluation += board.count_straight_four(1) * STRAIGHT_FOUR
        evaluation += len(board.find_open_three(1)) * OPEN_THREE
        evaluation += len(board.find_four(1)) * FOUR
        evaluation += board.count_three(1) * THREE
        evaluation += board.count_two(1) * TWO
        evaluation += board.count_one(1) * ONE

        evaluation += board.count_straight_four(2) * -STRAIGHT_FOUR
        evaluation += len(board.find_open_three(2)) * -OPEN_THREE
        evaluation += len(board.find_four(2)) * -FOUR
        evaluation += board.count_three(2) * -THREE
        evaluation += board.count_two(2) * -TWO
        evaluation += board.count_one(2) * -ONE

        evaluation += randint(-ONE//2, ONE//2) # randomize evaluation
        return evaluation

    def _minimax(self, board: BoardEvaluator, color: int, alpha: int, beta: int, depth: int) ->
    Tuple[int, List[Tuple[int, int, int]]]:
        """Applies the minimax algorithm to the board.

        Args:
            board (BoardEvaluator): The board to be evaluated.
            color (int): The color of the player whose turn it is. 1 for black, 2 for white.
            alpha (int): The alpha value.
            beta (int): The beta value.
            depth (int): The depth of the search tree.
            renju (bool): Whether renju rules are enabled.

        Returns:
            Tuple[int, List[Tuple[int, int, int]]]: A tuple of the evaluation of the board and the
    moves to make.
        """
        # apply the minimax alphabeta algorithm with the given candidate boards
        # black is the maximising player, white is the minimising player
        if depth == 0 or board.find_five(1) or board.find_five(2):
            return (self._local_evaluation(board), [])
        board.color = color
        candidate_boards = board.get_search_space(1)
        # sort the candidate boards by their evaluation, descending if black, ascending if white
        bubble_sort_k(candidate_boards, self.max_branches, self._local_evaluation, color == 1)
        candidate_boards = candidate_boards[:self.max_branches]
        if DEBUG and depth == self.depth:
            for cb in candidate_boards: print((cb.row, cb.col))
        best_eval = -WIN if color == 1 else WIN
        best_move = None
        best_moves = []
        for candidate_board in candidate_boards:
            new_eval, moves = self._minimax(candidate_board, 3-color, alpha, beta, depth-1)
            new_eval = previous_of(new_eval)
            if color == 1:
                if new_eval > best_eval:
```

```
                    best_eval = new_eval
                    alpha = new_eval
                    best_move = (candidate_board.row, candidate_board.col, color)
                    best_moves = [best_move] + moves
            elif color == 2:
                if new_eval < best_eval:
                    best_eval = new_eval
                    beta = new_eval
                    best_move = (candidate_board.row, candidate_board.col, color)
                    best_moves = [best_move] + moves
            if alpha >= beta:
                break
        return (best_eval, best_moves)

    def get_move(self, board: List[List[int]], color: int, renju: bool) -> Tuple[int, int]:
        """Makes a move on the board.

        Args:
            board (List[List[int]]): The board to be evaluated.
            color (int): The color of the player whose turn it is. 1 for black, 2 for white.
            renju (bool): Whether renju rules are enabled.

        Returns:
            Tuple[int, int]: The move to make.
        """
        return self.get_continuation(board, color, renju)[0][:2]

    def get_continuation(self, board: List[List[int]], color: int, renju: bool) -> List[Tuple[int,
    int]]:
        board_evaluator = BoardEvaluator(board, color, renju=renju)
        if board_evaluator.is_empty():
            return [(BOARD_SIZE // 2, BOARD_SIZE // 2, 1)]
        evaluation, moves = self._minimax(board_evaluator, color, -WIN, WIN, self.depth)
        if DEBUG:
            print(evaluation, moves)
        return moves

    def get_evaluation(self, board: List[List[int]], color: int, renju: bool) -> int:
        """Gets the evaluation of the board.

        Args:
            board (List[List[int]]): The board to be evaluated.
            color (int): The color of the player whose turn it is. 1 for black, 2 for white.
            renju (bool): Whether renju rules are enabled.

        Returns:
            int: The evaluation of the board.
        """
        board_evaluator = BoardEvaluator(board, color, renju=renju)
        if board_evaluator.is_empty():
            return 0
        evaluation, _ = self._minimax(board_evaluator, color, -WIN, WIN, self.depth)
        return evaluation
```

3.12.3 AggressivePlayer

computer_players/aggressive_player.py

```
from typing import List, Tuple
from random import randint

from board_evaluator import BoardEvaluator
from constants import *
from players import ComputerPlayer
from utils import previous_of, bubble_sort_k

class AggressivePlayer(ComputerPlayer):
    """A computer player that uses the minimax algorithm to make moves, with an aggressive
    evaluation function.
    The only difference between this and MinimaxPlayer is the evaluation function.

    Attributes:
        name (str): The name of the player.
        depth (int): The depth of the search tree.
        max_branches (int): The maximum number of branches to search.

```

```
    Methods:
        local_evaluation: Evaluates the board locally.
        minimax: Applies the minimax algorithm to the board.
        get_evaluation: Gets the evaluation of the board.
        get_move: Gets the best move to make on the board.
        get_continuation: Gets the best continuation to make on the board.
    """
    def __init__(self, depth: int, max_branches: int) -> None:
        """Initialises the AggressivePlayer.

        Args:
            depth (int): The depth of the search tree.
            max_branches (int): The maximum number of branches to search.
        """
        self.depth = depth
        self.max_branches = max_branches
        self.name = f"Aggressive {self.depth}"

    def _local_evaluation(self, board: BoardEvaluator, color: int) -> int:
        if board.find_five(1):
            return WIN
        if board.find_five(2):
            return -WIN
        black_evaluation = 0
        black_evaluation += board.count_straight_four(1) * STRAIGHT_FOUR
        black_evaluation += len(board.find_open_three(1)) * OPEN_THREE
        black_evaluation += len(board.find_four(1)) * FOUR
        black_evaluation += board.count_three(1) * THREE
        black_evaluation += board.count_two(1) * TWO
        black_evaluation += board.count_one(1) * ONE
        white_evaluation = 0
        white_evaluation += board.count_straight_four(2) * -STRAIGHT_FOUR
        white_evaluation += len(board.find_open_three(2)) * -OPEN_THREE
        white_evaluation += len(board.find_four(2)) * -FOUR
        white_evaluation += board.count_three(2) * -THREE
        white_evaluation += board.count_two(2) * -TWO
        white_evaluation += board.count_one(2) * -ONE

        # multiply the evaluation by the aggressive factor for own color
        if color == 1:
            black_evaluation = round(black_evaluation * AGGRESSIVE_FACTOR)
        else:
            white_evaluation = round(white_evaluation * AGGRESSIVE_FACTOR)
        evaluation = black_evaluation + white_evaluation

        evaluation += randint(-ONE//2, ONE//2) # randomize evaluation
        return evaluation

    def _minimax(self, board: BoardEvaluator, color: int, alpha: int, beta: int, depth: int) ->
    Tuple[int, List[Tuple[int, int, int]]]:
        """Applies the minimax algorithm to the board.

        Args:
            board (BoardEvaluator): The board to be evaluated.
            color (int): The color of the player whose turn it is. 1 for black, 2 for white.
            alpha (int): The alpha value.
            beta (int): The beta value.
            depth (int): The depth of the search tree.
            renju (bool): Whether renju rules are enabled.

        Returns:
            Tuple[int, List[Tuple[int, int, int]]]: A tuple of the evaluation of the board and the
    moves to make.
        """
        # apply the minimax alphabeta algorithm with the given candidate boards
        # black is the maximising player, white is the minimising player
        if depth == 0 or board.find_five(1) or board.find_five(2):
            return (self._local_evaluation(board, color), [])
        board.color = color
        candidate_boards = board.get_search_space(1)
        # sort the candidate boards by their evaluation, descending if black, ascending if white
        bubble_sort_k(candidate_boards, self.max_branches, lambda b: self._local_evaluation(b, color
    ), color == 1)
        candidate_boards = candidate_boards[:self.max_branches]
        if DEBUG and depth == self.depth:
            for cb in candidate_boards: print((cb.row, cb.col))
```

```
        best_eval = -WIN if color == 1 else WIN
        best_move = None
        best_moves = []
        for candidate_board in candidate_boards:
            new_eval, moves = self._minimax(candidate_board, 3-color, alpha, beta, depth-1)
            new_eval = previous_of(new_eval)
            if color == 1:
                if new_eval > best_eval:
                    best_eval = new_eval
                    alpha = new_eval
                    best_move = (candidate_board.row, candidate_board.col, color)
                    best_moves = [best_move] + moves
            elif color == 2:
                if new_eval < best_eval:
                    best_eval = new_eval
                    beta = new_eval
                    best_move = (candidate_board.row, candidate_board.col, color)
                    best_moves = [best_move] + moves
            if alpha >= beta:
                break
        return (best_eval, best_moves)

    def get_move(self, board: List[List[int]], color: int, renju: bool) -> Tuple[int, int]:
        """Makes a move on the board.

        Args:
            board (List[List[int]]): The board to be evaluated.
            color (int): The color of the player whose turn it is. 1 for black, 2 for white.
            renju (bool): Whether renju rules are enabled.

        Returns:
            Tuple[int, int]: The move to make.
        """
        return self.get_continuation(board, color, renju)[0][:2]

    def get_continuation(self, board: List[List[int]], color: int, renju: bool) -> List[Tuple[int,
    int]]:
        board_evaluator = BoardEvaluator(board, color, renju=renju)
        if board_evaluator.is_empty():
            return [(BOARD_SIZE // 2, BOARD_SIZE // 2, 1)]
        evaluation, moves = self._minimax(board_evaluator, color, -WIN, WIN, self.depth)
        if DEBUG:
            print(evaluation, moves)
        return moves

    def get_evaluation(self, board: List[List[int]], color: int, renju: bool) -> int:
        """Gets the evaluation of the board.

        Args:
            board (List[List[int]]): The board to be evaluated.
            color (int): The color of the player whose turn it is. 1 for black, 2 for white.
            renju (bool): Whether renju rules are enabled.

        Returns:
            int: The evaluation of the board.
        """
        board_evaluator = BoardEvaluator(board, color, renju=renju)
        if board_evaluator.is_empty():
            return 0
        evaluation, _ = self._minimax(board_evaluator, color, -WIN, WIN, self.depth)
        return evaluation
```

3.12.4 DefensivePlayer

computer_players/defensive_player.py

```
from typing import List, Tuple
from random import randint

from board_evaluator import BoardEvaluator
from constants import *
from players import ComputerPlayer
from utils import previous_of, bubble_sort_k

class DefensivePlayer(ComputerPlayer):
```

```
    """A computer player that uses the minimax algorithm to make moves, with a defensive evaluation function.
    The only difference between this and MinimaxPlayer is the evaluation function.

    Attributes:
        name (str): The name of the player.
        depth (int): The depth of the search tree.
        max_branches (int): The maximum number of branches to search.

    Methods:
        local_evaluation: Evaluates the board locally.
        minimax: Applies the minimax algorithm to the board.
        get_evaluation: Gets the evaluation of the board.
        get_move: Gets the best move to make on the board.
        get_continuation: Gets the best continuation to make on the board.
    """
    def __init__(self, depth: int, max_branches: int) -> None:
        """Initialises the DefensivePlayer.

        Args:
            depth (int): The depth of the search tree.
            max_branches (int): The maximum number of branches to search.
        """
        self.depth = depth
        self.max_branches = max_branches
        self.name = f"Defensive {self.depth}"

    def _local_evaluation(self, board: BoardEvaluator, color: int) -> int:
        if board.find_five(1):
            return WIN
        if board.find_five(2):
            return -WIN
        black_evaluation = 0
        black_evaluation += board.count_straight_four(1) * STRAIGHT_FOUR
        black_evaluation += len(board.find_open_three(1)) * OPEN_THREE
        black_evaluation += len(board.find_four(1)) * FOUR
        black_evaluation += board.count_three(1) * THREE
        black_evaluation += board.count_two(1) * TWO
        black_evaluation += board.count_one(1) * ONE
        white_evaluation = 0
        white_evaluation += board.count_straight_four(2) * -STRAIGHT_FOUR
        white_evaluation += len(board.find_open_three(2)) * -OPEN_THREE
        white_evaluation += len(board.find_four(2)) * -FOUR
        white_evaluation += board.count_three(2) * -THREE
        white_evaluation += board.count_two(2) * -TWO
        white_evaluation += board.count_one(2) * -ONE

        # multiply the evaluation by the defensive factor for opponent's color
        if color == 2:
            black_evaluation = round(black_evaluation * DEFENSIVE_FACTOR)
        else:
            white_evaluation = round(white_evaluation * DEFENSIVE_FACTOR)
        evaluation = black_evaluation + white_evaluation

        evaluation += randint(-ONE//2, ONE//2) # randomize evaluation
        return evaluation

    def _minimax(self, board: BoardEvaluator, color: int, alpha: int, beta: int, depth: int) -> Tuple[int, List[Tuple[int, int, int]]]:
        """Applies the minimax algorithm to the board.

        Args:
            board (BoardEvaluator): The board to be evaluated.
            color (int): The color of the player whose turn it is. 1 for black, 2 for white.
            alpha (int): The alpha value.
            beta (int): The beta value.
            depth (int): The depth of the search tree.
            renju (bool): Whether renju rules are enabled.

        Returns:
            Tuple[int, List[Tuple[int, int, int]]]: A tuple of the evaluation of the board and the moves to make.
        """
        # apply the minimax alphabeta algorithm with the given candidate boards
        # black is the maximising player, white is the minimising player
        if depth == 0 or board.find_five(1) or board.find_five(2):
```

```
            return (self._local_evaluation(board, color), [])
        board.color = color
        candidate_boards = board.get_search_space(1)
        # sort the candidate boards by their evaluation, descending if black, ascending if white
        bubble_sort_k(candidate_boards, self.max_branches, lambda b: self._local_evaluation(b, color), color == 1)
        candidate_boards = candidate_boards[:self.max_branches]
        if DEBUG and depth == self.depth:
            for cb in candidate_boards: print((cb.row, cb.col))
        best_eval = -WIN if color == 1 else WIN
        best_move = None
        best_moves = []
        for candidate_board in candidate_boards:
            new_eval, moves = self._minimax(candidate_board, 3-color, alpha, beta, depth-1)
            new_eval = previous_of(new_eval)
            if color == 1:
                if new_eval > best_eval:
                    best_eval = new_eval
                    alpha = new_eval
                    best_move = (candidate_board.row, candidate_board.col, color)
                    best_moves = [best_move] + moves
            elif color == 2:
                if new_eval < best_eval:
                    best_eval = new_eval
                    beta = new_eval
                    best_move = (candidate_board.row, candidate_board.col, color)
                    best_moves = [best_move] + moves
            if alpha >= beta:
                break
        return (best_eval, best_moves)

    def get_move(self, board: List[List[int]], color: int, renju: bool) -> Tuple[int, int]:
        """Makes a move on the board.

        Args:
            board (List[List[int]]): The board to be evaluated.
            color (int): The color of the player whose turn it is. 1 for black, 2 for white.
            renju (bool): Whether renju rules are enabled.

        Returns:
            Tuple[int, int]: The move to make.
        """
        return self.get_continuation(board, color, renju)[0][:2]

    def get_continuation(self, board: List[List[int]], color: int, renju: bool) -> List[Tuple[int, int]]:
        board_evaluator = BoardEvaluator(board, color, renju=renju)
        if board_evaluator.is_empty():
            return [(BOARD_SIZE // 2, BOARD_SIZE // 2, 1)]
        evaluation, moves = self._minimax(board_evaluator, color, -WIN, WIN, self.depth)
        if DEBUG:
            print(evaluation, moves)
        return moves

    def get_evaluation(self, board: List[List[int]], color: int, renju: bool) -> int:
        """Gets the evaluation of the board.

        Args:
            board (List[List[int]]): The board to be evaluated.
            color (int): The color of the player whose turn it is. 1 for black, 2 for white.
            renju (bool): Whether renju rules are enabled.

        Returns:
            int: The evaluation of the board.
        """
        board_evaluator = BoardEvaluator(board, color, renju=renju)
        if board_evaluator.is_empty():
            return 0
        evaluation, _ = self._minimax(board_evaluator, color, -WIN, WIN, self.depth)
        return evaluation
```

3.12.5 RandomMovesPlayer

computer_players/random_moves_player.py

```
from random import choice
```

```python
from typing import List, Tuple

from constants import BOARD_SIZE
from board_evaluator import BoardEvaluator
from players import ComputerPlayer

class RandomMovesPlayer(ComputerPlayer):
    """A computer player that makes random moves.

    Attribute:
        name (str): The name of the player.
    """
    def __init__(self) -> None:
        self.name = 'Random Moves'

    def get_move(self, board: List[List[int]], color: int, renju: bool) -> Tuple[int, int]:
        """Returns a random move."""
        board_evaluator = BoardEvaluator(board, color, renju=renju)
        if board_evaluator.is_empty():
            return (BOARD_SIZE //2, BOARD_SIZE // 2)
        candidate_boards = board_evaluator.get_search_space(1)
        candidate = choice(candidate_boards)
        return (candidate.row, candidate.col)

    def get_continuation(self, board: List[List[int]], color: int, renju: bool) -> List[Tuple[int, int]]:
        return [self.get_move(board, color, renju)]

    def get_evaluation(self, board: List[List[int]], color: int, renju: bool) -> int:
        return 0
```

3.12.6 PTPlayer

computer_players/prioritise_threats_player.py

```python
from random import randint
from typing import List, Tuple

from players import ComputerPlayer
from typing import List
from board_evaluator import BoardEvaluator
from constants import *
from utils import previous_of, bubble_sort_k, merge_sort

class PTPlayer(ComputerPlayer):
    """A computer player that prioritises to play threats. Refer to Section 2.2.3 of the report for more details.

    Attributes:
        name (str): The name of the player.
        depth (int): The depth of the search tree.
        max_branches (int): The maximum number of branches to search.

    Methods:
        local_evaluation: Evaluates the board locally.
        search: Searches for threats to play. If there are none, it will run minimax.
        minimax: Applies the minimax algorithm to the board.
        get_evaluation: Gets the evaluation of the board.
        get_move: Gets the best move to make on the board.
        get_continuation: Gets the best continuation to make on the board.

    """
    def __init__(self, depth, max_branches) -> None:
        self.depth = depth * 4
        self.max_branches = max_branches
        self.name = f"Prioritise Threats {depth}"

    def _local_evaluation(self, board: BoardEvaluator) -> int:
        """Returns the local evaluation of the board."""
        if board.find_five(1):
            return WIN
        if board.find_five(2):
            return -WIN
        evaluation = 0
        evaluation += board.count_straight_four(1) * STRAIGHT_FOUR
        evaluation += len(board.find_open_three(1)) * OPEN_THREE
```

```
        evaluation += len(board.find_four(1)) * FOUR
        evaluation += board.count_three(1) * THREE
        evaluation += board.count_two(1) * TWO
        evaluation += board.count_one(1) * ONE

        evaluation += board.count_straight_four(2) * -STRAIGHT_FOUR
        evaluation += len(board.find_open_three(2)) * -OPEN_THREE
        evaluation += len(board.find_four(2)) * -FOUR
        evaluation += board.count_three(2) * -THREE
        evaluation += board.count_two(2) * -TWO
        evaluation += board.count_one(2) * -ONE

        evaluation += randint(-ONE//2, ONE//2) # randomize evaluation
        return evaluation

    def _search(self, board: BoardEvaluator, color, alpha, beta, depth) -> Tuple[int, List[Tuple[int, int, int]]]:
        """Searches for threats recursively.

        Args:
            board (BoardEvaluator): The board to be evaluated.
            color (int): The color of the player whose turn it is. 1 for black, 2 for white.
            alpha (int): The alpha value.
            beta (int): The beta value.
            depth (int): The depth of the search tree.

        Returns:
            Tuple[int, List[Tuple[int, int, int]]]: The evaluation and the moves to make.
        """
        board.color = color
        opp_color = 3 - color
        # if depth >= 0:
        #     print(color,(board.row, board.col), alpha, beta, depth)

        # find fives
        if depth <= 0 or board.find_five(color) or board.find_five(opp_color):
            return (self._local_evaluation(board), [])

        # find fours, which are immediate wins or threats
        four_boards = board.find_four(color)
        if four_boards:
            # directly win, do not decrease depth
            evaluation, moves = self._search(four_boards[0], opp_color, alpha, beta, depth)
            return (previous_of(evaluation), [(four_boards[0].row, four_boards[0].col, color, 'four')] + moves)
        four_boards_opp = board.find_four(opp_color)
        if four_boards_opp:
            # forced to block, decrease depth by 1 since this search is short
            evaluation, moves = self._search(four_boards_opp[0], opp_color, alpha, beta, depth-1)
            return (previous_of(evaluation), [(four_boards_opp[0].row, four_boards_opp[0].col, color, 'opp four')] + moves)

        # find threes
        open_three_boards = board.find_open_three(color)
        if open_three_boards:
            # win in three moves by creating a straight four, so do not decrease depth
            evaluation, moves = self._search(open_three_boards[0], opp_color, alpha, beta, depth-1)
            return (previous_of(evaluation), [(open_three_boards[0].row, open_three_boards[0].col, color, 'open three')] + moves)

        # if opp have a three, it can also be refuted by creating a four
        three_boards_opp = board.find_open_three(opp_color)
        if three_boards_opp:
            candidate_boards: List[BoardEvaluator] = []
            candidate_boards.extend(three_boards_opp)
            any_threes = board.find_three(color)
            candidate_boards.extend(any_threes)
            bubble_sort_k(candidate_boards, self.max_branches, self._local_evaluation, color == 1)
            # call minimax on the candidate boards, decreasing depth by 2
            return self._minimax([(cb, 2) for cb in candidate_boards], color, alpha, beta, depth)

        # consider threats first, then fill up the rest with non-threats, decreasing depth
        three_boards = board.find_three(color)
        open_two_boards = board.find_open_two(color)
        non_threat_boards = board.get_search_space(1)

```

```
113         # sort the candidate boards by their evaluation, descending if black, ascending if white
114         # use library sort for better performance
115         three_boards.sort(key=self._local_evaluation, reverse=color == 1)
116         open_two_boards.sort(key=self._local_evaluation, reverse=color == 1)
117         non_threat_boards.sort(key=self._local_evaluation, reverse=color == 1)
118         # my merge sort
119         # merge_sort(three_boards, key=lambda b: self.local_evaluation(b), reverse=color == 1)
120         # merge_sort(open_two_boards, key=lambda b: self.local_evaluation(b), reverse=color == 1)
121         # merge_sort(non_threat_boards, key=lambda b: self.local_evaluation(b), reverse=color == 1)
122 
123         # threes are the most important, then open twos, then non-threats
124         # decrease depth by 2 for threes (since it creates a four - a threat)
125         # 3 for open twos, (since it creates an open three - a less compelling threat)
126         # 4 for non-threats, (does not create a threat)
127         candidate_boards = [(b, 2) for b in three_boards] + \
128                            [(b, 3) for b in open_two_boards] + \
129                            [(b, 4) for b in non_threat_boards]
130         candidate_boards = candidate_boards[:self.max_branches]
131         return self._minimax(candidate_boards, color, alpha, beta, depth)
132 
133     def _minimax(self, candidate_boards: List[Tuple[BoardEvaluator, int]], color, alpha, beta, depth
    ) -> Tuple[int, List[Tuple[int, int, int]]]:
134         """Applies the minimax alpha-beta algorithm to the board.
135 
136         Args:
137             candidate_boards (List[BoardEvaluator]): The boards to be evaluated.
138             color (int): The color of the player whose turn it is. 1 for black, 2 for white.
139             alpha (int): The alpha value.
140             beta (int): The beta value.
141             depth (int): The depth of the search tree.
142 
143         Returns:
144             Tuple[int, List[Tuple[int, int, int]]]: The evaluation and the moves to make.
145         """
146         best_eval = -WIN if color == 1 else WIN
147         best_eval = -WIN if color == 1 else WIN
148         best_moves = []
149         for (candidate_board, d) in candidate_boards:
150             new_eval, moves = self._search(candidate_board, 3-color, alpha, beta, depth - d)
151             new_eval = previous_of(new_eval)
152             if color == 1:
153                 if new_eval > best_eval:
154                     best_eval = new_eval
155                     alpha = new_eval
156                     best_moves = [(candidate_board.row, candidate_board.col, color)] + moves
157             elif color == 2:
158                 if new_eval < best_eval:
159                     best_eval = new_eval
160                     beta = new_eval
161                     best_moves = [(candidate_board.row, candidate_board.col, color)] + moves
162             if alpha >= beta:
163                 break
164         return (best_eval, best_moves)
165 
166     def get_move(self, board: List[List[int]], color: int, renju: bool) -> Tuple[int, int]:
167         return self.get_continuation(board, color, renju)[0][:2]
168 
169     def get_continuation(self, board: List[List[int]], color: int, renju: bool) -> Tuple[int, int]:
170         board_evaluator = BoardEvaluator(board, color, renju=renju)
171         if board_evaluator.is_empty():
172             return [(BOARD_SIZE //2, BOARD_SIZE // 2, 1)]
173         evaluation, moves = self._search(board_evaluator, color, -WIN, WIN, self.depth)
174         if DEBUG:
175             print(evaluation, moves)
176         return moves
177 
178     def get_evaluation(self, board, color, renju):
179         board_evaluator = BoardEvaluator(board, color, renju=renju)
180         if board_evaluator.is_empty():
181             return 0
182         evaluation, _ = self._search(board_evaluator, color, -WIN, WIN, self.depth)
183         return evaluation
```

3.12.7 RandomPlayer

computer_players/random_player.py

```
from random import choice
from typing import List, Tuple

from players import ComputerPlayer
from computer_players.minimax_player import MinimaxPlayer
from computer_players.prioritise_threats_player import PTPlayer
from computer_players.aggressive_player import AggressivePlayer
from computer_players.defensive_player import DefensivePlayer
from computer_players.random_moves_player import RandomMovesPlayer

COMPUTER_PLAYERS : List[ComputerPlayer] = [
    RandomMovesPlayer(),
    MinimaxPlayer(2, 10),
    MinimaxPlayer(3, 7),
    MinimaxPlayer(4, 5),
    AggressivePlayer(2, 10),
    AggressivePlayer(3, 7),
    AggressivePlayer(4, 5),
    DefensivePlayer(2, 10),
    DefensivePlayer(3, 7),
    DefensivePlayer(4, 5),
    PTPlayer(2, 10),
    PTPlayer(3, 5),
]

class RandomPlayer(ComputerPlayer):
    """A random computer player chosen from the List of computer players.

    Attributes:
        _instance (ComputerPlayer): The instance of the randomly selected computer player.
        name (str): The name of the player.

    Methods:
        get_move: Gets the best move to make on the board.
        get_evaluation: Gets the evaluation of the board.
        get_continuation: Gets the best continuation to make on the board.
    """
    def __init__(self) -> None:
        self._instance = choice(COMPUTER_PLAYERS)
        self.name = self._instance.name

    def get_move(self, board: List[List[int]], color: int, renju: bool) -> Tuple[int, int]:
        return self._instance.get_move(board, color, renju)

    def get_evaluation(self, board: List[List[int]], color: int, renju: bool) -> int:
        return self._instance.get_evaluation(board, color, renju)

    def get_continuation(self, board: List[List[int]], color: int, renju: bool) -> List[Tuple[int,
    int]]:
        return self._instance.get_continuation(board, color, renju)
```

Chapter 4

Testing

4.1 Unit Tests

4.1.1 EloRankings

Some unit test are carried out during the development process. The following tests are carried out to ensure the correctness of the `EloRankings` class.

- `rankings.update("A", "B")` is ran: {"A": 1516.0, "B": 1484.0} is shown in `rankings.json`.
- `rankings.update("A", "B")` is ran again: {"A": 1530.5..., "B": 1469.5...} is shown in `rankings.json`.
- `rankings.update("C", "D")` is ran again: {"C": 1500.0, "D": 1500.0} is shown in `rankings.json`.

The tests are passed. They can be carried out by directly running `elo_rankings.py`. They can be seen at the end of the file.

4.1.2 BoardEvaluator

The `is_legal_detailed` method of the `BoardEvaluator` class is tested during development to ensure that it can correctly determine the legality of a move. The following tests are carried out:

- Set up a board and play a move such that a 3x3 fork is created in the Renju ruleset.
- Set up a board and play a move such that a 4x4 fork is created in the Renju ruleset.
- Set up a board and play a move such that a 3x4 fork is created in the Renju ruleset.
- Set up a board and play a move such that a five is created.
- Set up a board and play an overline in the Renju ruleset.

The tests are passed. They can be carried out by directly running `board_evaluator.py`. They can be seen at the end of the file.

4.1.3 Other tests

Other classes are tested during development to ensure that they can correctly perform their functions. The tests are evident in the code usually in a `if DEBUG:` block, or commented out. They will not be described in detail here.

4.2 Final Tests

4.2.1 Video

A video is created to demonstrate the tests carried out. The video can be found at the following link:
https://youtu.be/v4UMOG3nAlo

4.2.2 Objective 1

The software should have a GUI that allows easy navigation to different parts of the software.

No.	Input	Output	Result
1	Click on the **Play** button in the title screen	The configuration window appears	Pass
2	Click on the **Analysis** button in the title screen	The analysis window appears	Pass
3	Click on the **Rankings** button in the title screen	The rankings window appears	Pass
4	In the play window, press the **Exit** button	The title window appears	Pass
5	In the analysis window, press the **Exit** button	The title window appears	Pass
6	In the rankings window, press the **Exit** button	The title window appears	Pass
7	Go to the play or analysis window	There is a sidebar on the right hand side of the screen	Pass
8	Make a valid move	A new button in the sidebar appears	Pass
9	Click on any button in the sidebar	The game board is replayed to the correpsonding move	Pass

4.2.3 Objective 2

The player should be able to play Gomoku and Renju.

No.	Input	Output	Result
10	Press **Play** in the configuration screen with any settings	The play window appears	Pass
11	With Gomoku ruleset, click on an empty cell on the game board	A piece appears	Pass
12	With Renju ruleset, click on an empty cell that is a valid move	A piece appears	Pass
13	With Renju ruleset, make a move that would then create a 3x3 fork	A message showing the move is a 3x3 fork and is invalid appears	Pass
14	With Renju ruleset, make a move that would then create a 4x4 fork	A message showing the move is a 4x4 fork and invalid appears	Pass
15	With Renju ruleset, make a move that would then create a overline	A message showing the move is an overline and is invalid appears	Pass
16	Place five pieces of the same colour in a row	A message showing who the winning player is appears	Pass
17	Hover on an empty cell on the game board	A transparent piece appears on the cell	Pass
18	Make some moves	The most recent move is always highlighted with the colour red	Pass

4.2.4 Objective 3

The player should be able to customise the settings for each game.

No.	Input	Output	Result
19	Select both the black and white players to be human players and press **Play**	Both players are controlled by the user	Pass
20	Select the black player to be **Minimax 2**	**Minimax 2** plays as the black player and plays first, the user controls the white player.	Pass
21	Select the white player to be **Minimax 2**, then make a move	The user controls the black player and plays first, while **Minimax 2** plays as the black player	Pass
22	Select the black player to be **Defensive 2** and the white player to be **Prioritise Threats 2**	**Defensive 2** as black plays a game with **Prioritise Threats 2** as white	Pass
23	Select the white player to be **Random Player** and repeat this several times	The white player is a computer player selected at random	Pass
24	Select both players to be **Random Player**, and set repeat to 5	Five games of pairs of randomly selected computer players are played	Pass

4.2.5 Objective 4

The player should be able to save a game after it is played.

No.	Input	Output	Result
25	In the play screen, press **File** > **Save Game** in the menu bar	A save game window appears	Pass
26	In the save game window, enter "Alice" and "Bob" as the white and black player names respectively, then press **Save**	The game is saved to a file, with the names "Alice" and "Bob" in the file name	Pass

4.2.6 Objective 5

The player should be able to analyse a game after it is played.

No.	Input	Output	Result
27	Click **File** > **Load Game** in the menu bar	The system's default file manager appears	Pass
28	In the file manager, click and open a saved game with `.json` format	The game is loaded in	Pass
29	In the analysis window, click on the **Black** button and click on cells	The cells are changed to black pieces	Pass
30	Click on the **White** button and click on cells	The cells are changed to white pieces	Pass
31	Click on the **Black** / **White** button and click on cells	Empty cells that are clicked are changed to pieces which the colour alternates in the order they are clicked	Pass
32	Setup the board in a way that black has a large advantage, then press **Get Evaluation**	A message showing black has a great advantage appears	Pass
33	Setup the board in a way that white has a reasonable advantage, select **Minimax 2** as the analysing computer player then press **Get Evaluation**	A message showing white has a slight advantage appears	Pass
34	Setup the board in a way that black has a forced winning sequence, select **Prioritise Threats 3** as the analysing computer player then press **Get Evaluation**	A message showing black has a forced winning sequence appears	Pass
35	In any board state, press **Get Best Continuation**	A message showing the continuation moves appears	Pass

4.2.7 Objective 6

There should be a variety of computer players.

No.	Input	Output	Result
36	In the configuration screen, open the select menu for players	11 different computer players are shown	Pass
37	Open the rankings screen	The computer players have a variety of ratings	Pass

4.2.8 Objective 7

The software should have a ranking system.

No.	Input	Output	Result
38	Go to the rankings screen	Rankings of computer players are shown in descending order of rating	Pass
39	Win a game against a computer player, then go to the rankings screen	The rating of the computer player is decreased, and the rating of the human player is increased	Pass
40	Lose a game against a computer player, then go to the rankings screen	The rating of the computer player is increased, and the rating of the human player is decreased	Pass

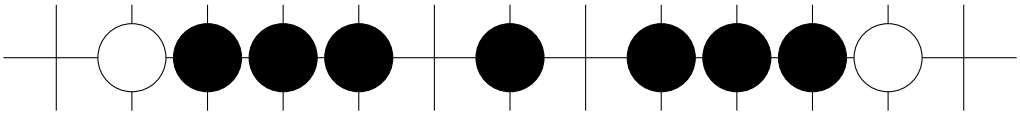


Figure 4.1: Position for 4x4 fork

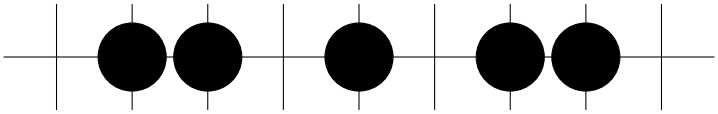


Figure 4.2: A valid move that looks like a 3x3 fork.

4.2.9 Boundary Tests

The following tests are carried out to ensure that the software can handle the edge cases. These situations are very unlikely to occur in a real game, but they are tested to ensure that the software can handle them.

No.	Input	Output	Result
41	In the Renju ruleset, input the position shown in Figure 4.1, placing the central stone last	A message showing the move is a 4x4 fork and is invalid appears	Pass
42	In the Renju ruleset, input the position shown in Figure 4.2, placing the central stone last	The move is placed	Pass
43	Fill in every cell on the board, without winning	A message showing the game is drawn appears	Pass

4.2.10 Erroneous Tests

Most invalid inputs in the software cannot be entered through the GUI. However, in the analysis window, the user may load in an invalid game file, so it must also be tested.

No.	Input	Output	Result
44	Press load game, and choose a JSON file that is not a game file	A message showing the file is invalid appears	Pass

4.3 Summary

All tests, including the boundary and erroneous tests, are passed. The software is able to handle all the inputs and outputs as expected. The tests completely cover the objectives of the software. The software is very robust, and is ready for gathering feedback from the client.

Chapter 5

Evaluation

5.1 Objective 1

The software should have a GUI that allows easy navigation to different parts of the software.

In the title screen, there are three working buttons, leading to the configuration, analysis, and ranking screen respectively. In those screens and the play screen, exit buttons can also be pressed, leading back to the title screen (objectives 1.1-2, tests 1-6). In the play screen and analysis screen, there is a sidebar which allows the user to navigate to any move in the game (objective 1.3, tests 7-9).

The objective is fully met. Every part of the software can be navigated to and from easily, only using the mouse. There is nothing more to be improved in this area when revisiting the software.

5.2 Objective 2

The player should be able to play Gomoku and Renju.

In the play screen and analysis screen, there is a board that allows the user to play Gomoku and Renju (objective 2.1, test 10). In both rulesets, the user can place valid moves but not invalid moves (objectives 2.2-3, tests 11-15). The user can claim a win (objective 2.4, test 16). Transparent pieces are appear when a cell is hovered, and the most recent move is highlighted with red (objectives 2.5-6, tests 17-18).

The objective is fully met.

5.3 Objective 3

The player should be able to customise the settings for each game.

When the play button is pressed, a configuration screen appears, allowing the user to change the settings for the game (objective 3.1, test 1). The user can choose the rulseset of the game (objective 3.2, tests 11-15). The user can play against another player or the AI, choosing which colour to play as (objectives 3.3, tests 19-21). The user can also choose to spectate a computer vs computer game (objective 3.4, test 22).

The objective is fully met. Additionaly, the user can also specify how many times a computer vs computer game should be played (test 24), if they wish to update the rankings. This is a feature that was not planned, but was added to the software to make it more user-friendly. There is nothing more to be improved in this area when revisiting the software.

5.4 Objective 4

The player should be able to save a game after it is played.

In the play screen, there is a save button that allows the user to save the game (objective 4.1, test 25). The user can also specify the name of both players. The game is saved in a file with the names of both players and the date when the game is played (objectives 4.2-3, test 26).

The objective is fully met. A possible improvement would be to also save all branches of the game while analysing the game. It would also be great if the setup of the board could be saved, so that the user can analyse from a specific position. This is not a priority, but it would be a great feature to have.

5.5 Objective 5

The player should be able to analyse a game after it is played.

In the analysis screen, the user is able to load in a previously saved game (objective 5.1, tests 27-28). The user can set up the board to any positions with the buttons (black, white, black / white) (objectives 5.2-3, test 29-31). The user is also able to see which side is winning (objective 5.4, tests 32-34). The user can also see the best continuation for the current position (objective 5.5, test 35).

The objective is fully met. In addition to objective 5.4, the user can also see whether which side has a forced winning sequence. This helps the user to understand the position better. There is nothing more to be improved in this area when revisiting the software.

5.6 Objective 6

There should be a variety of computer players.

In the configuration, analysis, or rankings screen, it can be seen that there are 11 different computer players with a variety of ratings (objective 6.1, tests 36-37). The user can play against any of these computer players (objective 6.2, tests 20-21). The computer players can play against each other (objective 6.3, test 22).

The objective is fully met. There is nothing more to be improved in this area when revisiting the software.

5.7 Objective 7

The software should have a ranking system.

In the rankings screen, the user can see the ranks and ratings of all computer players and the user (objectives 7.1-2, test 38). The ratings are updated after every game (objective 7.3, tests 39-40).

The objective is fully met. There is nothing more to be improved in this area when revisiting the software.

5.8 Client Feedback

An interview was conducted with the client to gather feedback on the software. The following is the manuscript of the interview.

Question: Did the software meet your expectations?

Mr Shafer: Definitely, it includes everything I wanted and more. The interface is user-friendly, the gameplay is smooth, and most importantly, the analysis tool is very helpful. I am very satisfied with the software.

Question: Are there any sections of the software that exceeded or fall below your expectations?

Mr Shafer: The collection of AI players exceeded my expectations. They have different skill levels and playing styles, which makes playing with them very interesting. There are no areas that fall below my expectations.

Question: Is there anything that could be improved in the software?

Mr Shafer: The software is great as it is, but being able to download Gomoku/Renju files online would enhance my analysis. I could analyse professional games, or play custom positions such as puzzles developed by other people.

Question: Is there anything that you would like to add to the software in the future?

Mr Shafer: An online multiplayer mode would be a great addition. Playing with computer players is fun, but playing with other people is even better. It would also be great if the software could also be used on mobile devices, so that I can play on the go.

5.9 Summary

It was a long 24 weeks of analysing the problem, designing, developing, testing, and evaluating the software. I have met all my objectives and developed a software that the client is satisfied with.

There are many lessons learnt through interacting with the client. I have always created software for myself, but this time, I had to create software for someone else. I learnt to prioritise his requirements, making sure that the software is usable for them. I have also learnt to communicate with the client effectively, making sure that I understand their requirements and they understand my progress.

In terms of development, I have learnt how to use the PyQt library, which is a very powerful library for creating professional-looking GUIs. I have also learnt proper coding conventions, such as using docstrings and comments to make the code more readable. I have also learnt to use version control with Git, which is very important when working on a large project.

5.10 Future Work

The client has suggested that the software could be improved by adding the ability to download Gomoku/Renju files online. This would allow the user to analyse professional games, or play custom positions such as puzzles developed by other people. This is a feature that I could add to the software in the future.

The client has also suggested that an online multiplayer mode would be good to have. This feature might be considered by web developers who are interested in creating a web version of the software.

The client has also suggested that the software could be used on mobile devices. This would require a complete rewrite of the software, which might be time-consuming. Mobile developers may be interested in this project.

Bibliography

[1] L.V. Allis, H.J. van den Herik, and M.P.H. Huntjens. *Go-Moku and threat-space search*. English. Technical reports in computer science CS 93-02. Pagination: 11. University of Limburg, Department of Computer Science, 1993.

[2] Jiun-Hung Chen and Adrienne X. Wang. *Five-in-row with local evaluation and Beam Search*. Aug. 2017. URL: https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/JA/report.pdf.

[3] Arpad E. Elo. *The Proposed USCF Rating System, Its Development, Theory, and Applications*. Aug. 1967. URL: https://uscf1-nyc1.aodhosting.com/CL-AND-CR-ALL/CL-ALL/1967/1967_08.pdf#page=26.