2022

UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG 100 1922 2022

C++ GOMOKU COMPUTER GAME

22 MAY 2022

MUAAWIYAH DADABHAY - STUDENT NUMBER: 2426234

UNIVERSITY OF THE WITWATERSRAND
SCHOOL OF ENGINEERING AND THE BUILT ENVIRONMENT
ELECTRICAL AND INFORMATION ENGINEERING
SOFTWARE DEVELOPMENT I

Abstract:
Gomoku is a two-player traditional Japanese board game in which players take turns arranging pieces on a 19x19 board. The game's goal is to create a continuous line of 5 placements. [1] The board game has been modified into a computer program through C++ implementation. The computer program dissimilar from its traditional counterpart operates without human input, hence two algorithms are set up to compete against one another. Although the game is commonly played on a 19x19 size board [1], provisions have been developed to play games in sizes ranging from 6x6 to 15x15. The program's design is discussed thoroughly within this report with substantial motivation. It is presumed that the reader knows aspects of software development and the C++ language.

Keywords: Gomoku, C++, Algorithms, Input

1. Introduction

The aim was to create and implement a computer version of the classic Gomoku game. Gomoku (also known as five-in-a-row) is an ancient strategic game recognized by several names globally. The goal of Gomoku is to establish a row of five consecutive stones of two different colours, one for each of the two players. The qualifying row could be either horizontally, vertically, or diagonally. Both players would take turns until a winner is determined. Gomoku is generally played on a 19x19 size board known as a GO-Board [1]. The software had to be programmed such that no user inputs were to be processed, instead, two algorithms were to be established, with component placement dependent on their separate algorithms. Since user input is restricted, the input files used during setup comprise various board game sizes. Restrictions are discussed further in section 2.2. Implementation of the game is discussed further in section 3 followed by the design of the game subbed into section 3.

2. Problem Definition

2.1 Game-play

Due to the restriction of user input throughout the game's runtime, an input text file feeds through the Go-Board sizes of each game. It needs mention that the program's design allows for a dynamic amount of games played and Go-Board sizes ranging from 6x6 to 15x15. Upon the start of each game, two algorithms that use chance and mathematics alternate turns to play until a winner is determined. As the algorithms place hidden pieces, the corresponding placement denoted in an informal matrix notation is transferred to an output text file. An algorithm can only win when it can achieve a row of five consecutive pieces on the Go-Board, either in a vertical, horizontal, or diagonal setting. Once the game has ended, the output text file contains the game's results and displays the winning algorithm.

2.2 Specifications, Constraints and Assumptions

Specifications of the implementation of the game include:

- Dynamically receive Go-Board sizes
- Dynamically play a received amount of games
- Create two Algorithms that play against one another
- Make use of a user-defined class
- Make use of at least one user-defined function

Constraints include:

- The use of global variables is not allowed
- Other than input from a text file, no other user inputs are allowed
- There should be no console output

Assumptions:

- A game ends in two instances, algorithm 1 or 2 wins or a draw occurs
- Algorithms are unable to play pieces over an already used up space
- Each algorithm takes alternating turns to place a piece
- A game always starts with Algorithm 1 making the first placement

- A winner of an individual game played is displayed in an output text file
- The last few lines of the generated output text file contain total summaries on the number of games each Algorithm won

2.3 Success Criteria

The program can compile and run successfully displaying the desired outputs and ticking the boxes about the specifications and assumptions listed. Section 4 discusses possible improvements for a more artificially intelligent approach and the drawbacks of the current implementation.

3. Design and Implementation

When planning out the design of the code, the best approach to a neat and constructive program was to make use of classes. Hence the design would consist of three files namely: main.cpp, gomoku.h and gomoku.cpp. A flowchart visualizing the process is presented in *Appendix B Figure 1*.

3.1 Class Design

Figure 1 indicates the name of the class created and includes the private and public associated variables/functions.

Gomoku
- board_size; - player1_loop - player2_loop - temp_row - temp_col
+ setBoardSize() + printBoard() + playPlayer2() + generateRandomNumber() + determineWinner() + emptyPlacement() + fullBoard() + checkHorizontalFiveLeft() + checkVerticalFiveUp() + checkVerticalFiveDown() + checkDiagonalRightUp() + checkDiagonalRightDown() + checkDiagonalLeftUp() + checkDiagonalLeftDown()

Figure 1: A class diagram of the user-defined class used within this implementation

3.2 Implementation

The design implementation is discussed under three subtopics respective to the three file names.

3.2.1 main.cpp

The code in this file is minimal due to the usage of classes. Following the fundamentals of engineering design, the code follows an Input – Process – Output fashion.

The input container contains all code related to variable initialization and text file generation. File validation through the use of a conditional statement helps while reading in the input text file to avoid runtime errors and the possibility of crashing. For dynamic inputs, the text file contents are placed within a vector. The use of the vector header allows the use of vectors. The input text file contains the Go-Board sizes.

The process section utilizes two loops, one within the other. The outer loop is a for loop that runs through the number of games received through the input file, whereas an inner while loop controls a single game. So long as the game is active, the inner while loop will run. Within the for loop variables are initialized and temporary for that specific game. Once a new game is in motion, the variables are reset. The current board size is received from the outer for loop. Functions declared within the class files are also called within this outer loop to set up the game. The game is set up with ‘0’s in each placement and uses a 2D array to which the row and column size match that of the input received. Within the inner while loop, various functions from the Gomoku class are used for tasks such as the placement of pieces. However, code is still needed to determine a winner, and output the received outcomes. If a winner is found, a flag is set and the game ends, terminating the while loop. To determine which algorithm is the winner, an additional for loop is implemented that runs through each element within the 2D array. By use of functions from the Gomoku class, if a placement happens to follow the conditions to win, a flag is set and the game would end.

The output section of the main.cpp mainly contains the closing of text files and outputting the total number of wins each algorithm made.

3.2.2 gomoku.h

'gomoku.h' is a header file that includes the functions that control the general functionality of the game. The defined class is divided into two sections: private and public, with a few private variables used. The functions are declared within the public section. Explanations of the implementations of the functions are elaborated in section 3.2.3.

3.2.3 gomoku.cpp

The first few functions pertain to setting up the game board. The function "createBoard" runs through a 2D array and assigns 0's to each index, with the 0's representing vacant spots on the Go-board.

"printBoard" is an optional feature that aids in the depiction of the Gomoku game and is not necessary as a specification. As a result, calling it in the main function will be removed/commented on once testing is complete. Nonetheless, this function loops over the Go-boards 2D (two-dimensional) array and displays the contents of the array to the console.

"placePiece" receives parameters such as the desired row and column an algorithm would like to play and adjusts the game board accordingly. However, to make a placement, a function labeled "emptyPlacement" of type Boolean first validates if a move is possible by determining if no algorithm has already used that index.

"fullBoard" runs through the 2D array of the Go-board. By use of a conditional statement, if any index contains a '0', it would mean the Go-board is not yet full and return a false value.

There are 8 functions that check whether a horizontal, vertical, or diagonal win has occurred. 2 functions for horizontal, 2 for vertical, and 4 for diagonal, 2 diagonal right, and 2 diagonal left. The reason for so many functions is to avoid any checks being missed. These functions make use of multiple conditional statements that determine if a winner is found respective to the function's name. The list of these function names is listed in Figure 1's class diagram. An example of a handwritten working out used to help refine the conditions of these checks may be found in *Appendix B Figure 2*.

3.2.3.1 Algorithm 1 ("playPlayer1")

First and foremost, a function called "generateRandomNumber" was created to randomly generate a number between 0 and the current size of the Go-board. However, when implementing the function, a seed was required. Without a seed, the random numbers generated would follow a constant sequence [2]. The seed helps break up the numbers sequence and is associated with the current time of a computer's clock. Hence every time a game is run, a completely different set of numbers would be generated due to the constant change in time [2]. Since Algorithm 1 would like to place a piece, a for loop is run an odd amount of times generating random numbers until the loop ends. The values generated in the last loop cycle would be used as the pieces rows and columns respectively unless that specific placement already has a piece. In this situation, the for loop would re-execute and continue to do such until the conditions of a free random space arise. Once an empty placement is located, the "placePiece" function is utilized. The counter used for the odd loop is incremented by 2 always keep an odd condition. It was decided that an odd number be used since 1 for Algorithm 1 is an odd number.

3.2.3.2 Algorithm 2 ("playPlayer2")

Algorithm 2 is similar to Algorithm 1 however instead of the for loop running an odd amount of times, it runs an even amount since 2 is an even number. This seems like a fair situation to both algorithms as all moves are completely random.

4. Results

The game runs and compiles as intended and displays the necessary output listed under the assumptions heading. Due to the dynamic specifications, in the project testing, an input file contained a single game of a Go-board size of 6x6. The file's contents are depicted in *Figure 2*. Upon runtime completion, the generated output is represented in *Figure 3*.

```
6
```

Figure 2: Contents of the input.txt file used for project testing

```
size=6
r5c2 alg1
r0c4 alg2
r4c1 alg1
r2c5 alg2
r5c5 alg1
r1c5 alg2
r1c2 alg1
r2c2 alg2
r3c0 alg1
r3c5 alg2
r1c0 alg1
r1c3 alg2
r4c5 alg1
r3c4 alg2
r4c2 alg1
r5c4 alg2
r0c0 alg1
r3c3 alg2
r4c0 alg1
r2c3 alg2
r0c3 alg1
r5c1 alg2
r0c1 alg1
r2c0 alg2
r2c4 alg1
r5c3 alg2
r3c2 alg1
r0c5 alg2
r1c1 alg1
r2c1 alg2
r5c0 alg1
r4c3 alg2
win=alg2

wins alg1=0
wins alg2=1
wins draw=0
```

Figure 3: Contents of the gomokuResults.txt text file (output) generated for project testing with the input from Figure 2

1 1 0 1 2 2
1 1 1 2 0 2
2 2 2 2 1 2
1 0 1 2 2 2
1 1 1 2 0 1
1 2 1 2 2 1

Figure 4: A visual representation of the above game that has been played on a Go-board of size 6x6

From the above three figures, the program works as desired. Algorithm 2 is confirmed to be the winner of the game by analysing *Figure 4* which depicts a five consecutive win vertically.
To test the effectiveness of the two algorithms, a hundred games were run for each Go-board size and graphed into *Figure 5*.

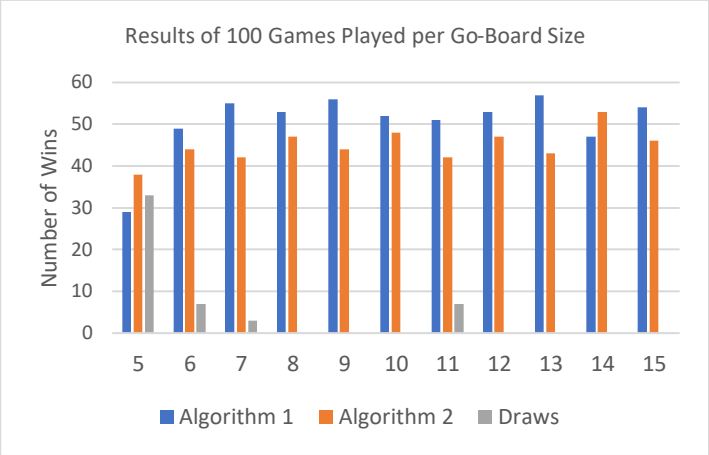

Figure 5

It was observed while conducting the above testing for *Figure 5*, that as the Go-Board size increased, the time taken to play a game also increased. Hence a proportional relationship between the two. This is due to the increased amount of available tiles.

5. Critical Evaluation of Solution

Through analysing *Figure 5*, it is observed that Algorithm 1 tends to win more games than Algorithm 2. This is because Algorithm 1 was designed to always play first whenever a game started giving the algorithm an advantage. Although Algorithm 1 tends to win a few more games than Algorithm 2, there seems to be around a 50% chance between each algorithm gaining a win with rarely a draw ever concluded. This is due to both Algorithms only varying slightly and relying on the generation of random numbers. The use of randomization may be seen as a flaw since there are no "smart" moves. Randomizing the placements also leads to longer games being played especially on bigger Go-board sizes since there is an increase in empty placements.

Another flaw lies within the main.cpp file particularly the code segment regarding a winner. A loop is run through the game board's two-dimensional array to determine a winner after every single placement. This increases the time overall to complete.

6. Recommendations

Recommendations include improving the function to locate a winner since the current implementation runs through the whole game's array every time a piece is placed. Another recommendation would be to remove the randomization used in the algorithms and adopt a smarter algorithm. Research and similar implementations on the net suggest making use of the Minimax function [3]. The Minimax method is a backtracking technique used in decision-making and game theory to identify the ideal move for an algorithm, provided the opposing algorithm plays optimally. Minimax is common in two-player turn-based games. [3]

7. Conclusion

The Gomoku game was designed and implemented effectively with an easy-to-read flow of code due to the usage of classes. The code compiles and runs without errors and correctly displays the required outputs. All specifications and assumptions listed have been checked and overall, the game is ready to tackle any input received hence the game is dynamic. The use of randomization allows for a working implementation however by making use of more complex algorithms, an artificially intelligent game may be implemented.

References

[1] Weinhardt, A., 2007. *Yucata - Rules for the game 'Gobang & Gomoku'*. [online] Yucata.de. Available at: <https://www.yucata.de/en/Rules/Gomoku> (Accessed 22 May 2022).

[2] GeeksforGeeks. 2022. *rand() and srand() in C/C++ - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/rand-and-srand-in-ccpp/> (Accessed 23 May 2022).

[3] GeeksforGeeks. n.d. *Minimax Algorithm in Game Theory | Set 1 (Introduction) - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/> [Accessed 25 May 2022].

Appendix A: Time Management

Table 1: Actual vs Estimated Time pertaining to time management of the project

Activity	Time Spent (Hours)	Estimated Time (Hours)
Background	0.48	3
Analysis and Design	0.88	4
Implementation	9.37	4
Testing	2.18	4
Documentation	7.17	5
Total	20.08	20

The project took nearly an hour longer to complete versus the estimated time frame calculated. Implementation and Documentation were over the estimated times compared to the remaining activities. Possible reasoning for increased time spent in implementation may be due to the lack of time spent on the background and the analysis and design. Perhaps if more time was focused on the analysis and design a more intelligent algorithm could have been implemented.

Appendix B: Visual Aid

Contained within this appendix lies a flowchart as well as some rough logical analysis used when writing functions for the Gomoku class.

Figure 1 - Flowchart:

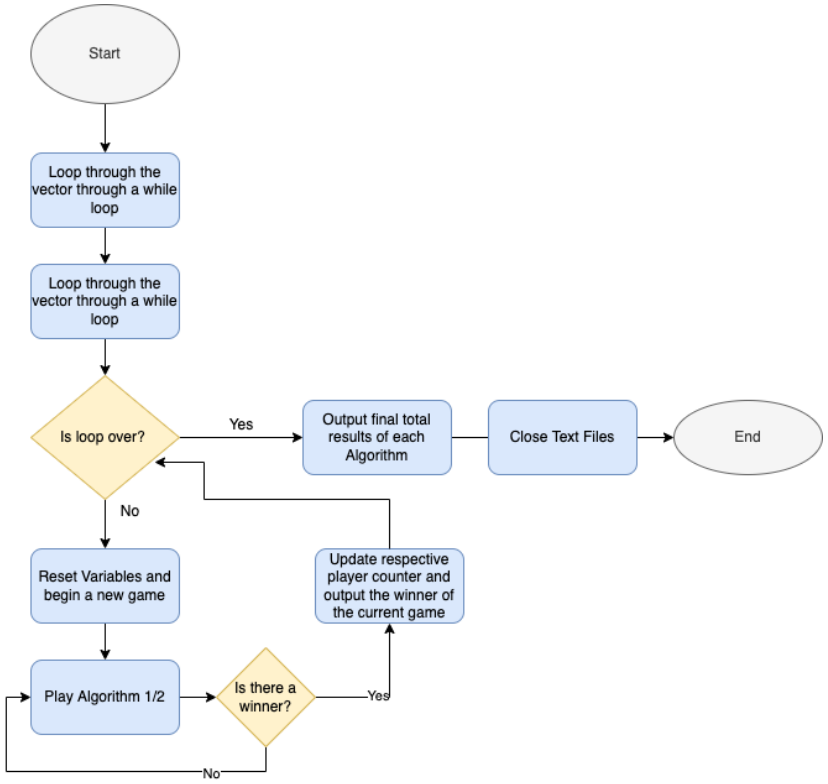

Figure 1: Flow chart of the main.cpp file

Figure 2 – Rough Working:

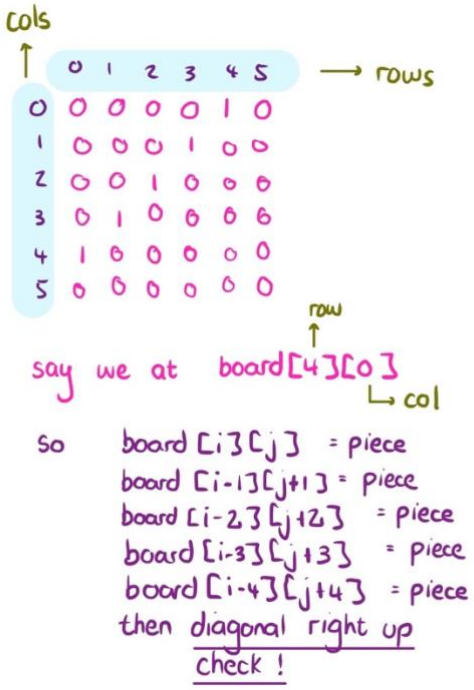

Figure 2: Showing handwritten designs on the winner checks