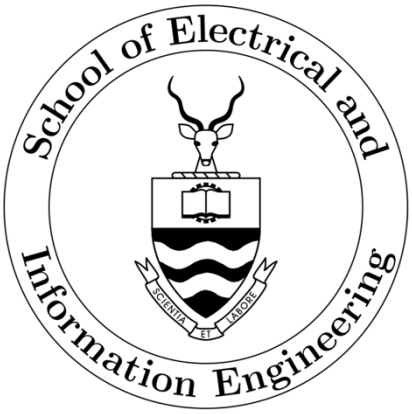


School of Electrical and Information Engineering
University of the Witwatersrand, Johannesburg
ELEN2020 - Software Development I

Author: Taahir Kolia

Student Number: 2423748

Branch: Electrical Engineering

Date: 30 May 2022

Implementing an intelligent and a random C++ algorithm to play against each other in *Gomoku*.

Abstract: The aim of the project is to design and implement two algorithms that play each other in a game of Gomoku. Algorithm one is developed to play random moves whilst algorithm two plays in positions that increases its chances of winning. After conducting multiple tests, algorithm two is shown to be a better designed algorithm.

1. INTRODUCTION

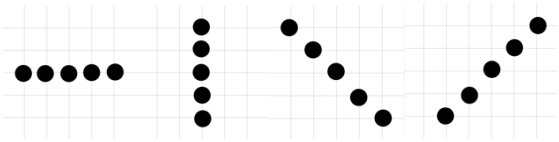
The aim of this project is to develop and implement two C++ algorithms that play against each other in a game called *Gomoku*. *Gomoku* is a Japanese strategy board game which involves two players that take alternating turns to place their pieces (either black or white pieces) in an empty space with the goal of placing five of their respective pieces in a continuous line [1]. Generally, the game is played on a 15x15 grid board, however there are other acceptable boards sizes that the game can be played on [1]. These rules are thoroughly considered during the design and implementation process to ensure that the game simulation is coherent with how the game is meant to be played. A further insight into the design, implementation, and testing process of the game and two different algorithms are discussed in the sections to follow.

2. DESIGN CONSIDERATIONS

2.1 Assumptions

Certain assumptions are made to clarify how the programme and algorithms should be designed. The first assumption made is that the two algorithms can play anywhere on the board provided that the position is empty. If the position that an algorithm is going to make a move to is occupied by either the opposition piece or its own piece, then it cannot play there or replace any of the previous moves made.

A second assumption that is made is that there are only four possible ways that an algorithm win. An algorithm can win if it has five pieces in a line horizontally, vertically, diagonally (leading diagonal) or diagonally in the other direction (counter diagonal).



Lastly, if the board is fully occupied and no winner is declared, the game ends as a draw and should be replayed.

Figure 1: Win outcomes when there are 5 in a line horizontally, vertically, diagonally, and counter diagonally

2.2 Constraints

Various constraints are given regarding the game board and user input. The programme is required to read in multiple different board sizes from a text file provided that the board size is within a certain range [2]. The smallest board size allowed is six whilst the largest is fifteen [2]. If a board size falls outside of this range, it should be disregarded. Thereafter the programme is required to set a board size that has equal an equal number of rows and columns [2]. The algorithms then begin playing the game until a winner is declared. The programme should not prompt any input from the user and thus there should be no external influence on the algorithms once the programme is run and the game begins [2].

3. DESIGN IMPLEMENTATION

Using classes and functions are an integral part of the project and are needed to get to the required outcome. Therefore, an object-orientated approach is used whereby all the functions that are deployed are defined within the class “gomoku” and its header file. Operations relating to reading from the file and commencing the game are found within the main.cpp file.

3.1 Gomoku Class

The “gomoku” class consists numerous functions that work together to make the game operational.

The *Gomoku* board consists of rows and columns. Therefore, the board needs to be initialised as a 2D array. A function called

"printBoard" is created to traverse through the rows and columns of the game board array for a given board size and fill them with empty spaces which are used to identify if an algorithm can make a move in that position.

The "isMoveValid" function makes use of the empty spaces and the algorithm pieces ('X' for algorithm one and 'O' for algorithm two) to determine whether the algorithms are allowed to play in a specific position.

The "checkForWinner" function is a recursive that traverses through the game board array and looks for five of the same markers in a line for the four different cases that are previously mentioned. Once it has identified five pieces in a row, it returns who the winner of the game is or in the unlikely case it returns that game ends in a draw.

3.2 Algorithm Design

Two different approaches are taken on the design of the algorithms. Algorithm one is designed to generate random numbers within the size of the given game board. These randomly generated numbers are assigned to a position on the game board for algorithm one to make a move. The algorithm then uses the "isMoveValid" function to verify that the randomly generated move can be made. If a move is not valid, then a new random number is generated until a valid move can be made. To ensure that a different sequence of numbers are generated each time a random number is needed, "srand()" has an argument of "time(NULL)" [3]. A flowchart of the design of algorithm one can be found alongside.

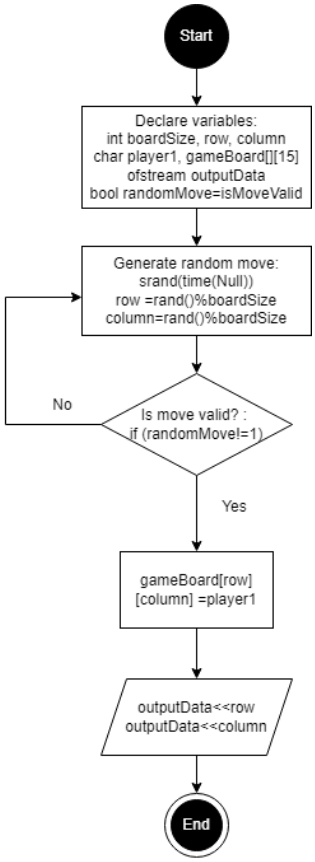


Figure 2: Flowchart of the design of algorithm one

Initially algorithm two was designed to play random moves in the same way as algorithm one. However, when the two algorithms were made to play against each other the programme would run for prolonged periods of time before the game would end or a winner was found. This was due to all the moves being random and was a problem common specifically for the smaller board sizes. Therefore, the need for a more efficient algorithm arose.

Algorithm two has been redesigned to make more intelligent moves once the board starts to fill up. In the early stages of the game, algorithm two makes random moves as it previously did but in the later stages it makes moves that increases its chances of winning. The algorithm traverses through the game board and uses recursive functions to find the optimal move to make. The algorithm looks for a certain number of its own pieces (represented by 'O') in line horizontally, vertically, diagonally, and counter diagonally and then plays in a position next to those pieces if the position is available.

If an intelligent move cannot be made, the algorithm will play a random move.
The pseudocode describing algorithm two is found in the figure below.

If there are four of algorithm two's pieces in a line horizontally and an empty space on either side of the pieces.
Play the winning move in the empty space.
Break.
Else if there are four of algorithm two's pieces in a line vertically and an empty space at the top or bottom of the pieces.
Play the winning move in the empty space.
Break.
Else if there are four of algorithm two's pieces in a line diagonally and an empty space at the top or bottom of the diagonal.
Play the winning move in the empty space.
Break.
Else if there are four of algorithm two's pieces in a line counter diagonally and an empty space at the top or bottom of the counter diagonal.
Play the winning move in the empty space.
Break.
Else if there are three of algorithm two's pieces in a line horizontally and an empty space on either side of the pieces.
Play the move in the empty space.
Break.
Else if there are three of algorithm two's pieces in a line vertically and an empty space at the top or bottom of the marker pieces.
Play the move in the empty space.
Break.
Else if there are three of algorithm two's pieces in a line diagonally and an empty space at the top or bottom of the diagonal.
Play the move in the empty space.
Break.
Else if there are three of algorithm two's pieces in a line counter diagonally and an empty space at the top or bottom of the counter diagonal.
Play the move in the empty space.
Break.
Else if there are two of algorithm two's pieces in a line horizontally and an empty space on either side of the pieces.
Play the move in the empty space.
Break.
Else if there are two of algorithm two's pieces in a line vertically and an empty space at the top or bottom of the pieces.
Play the move in the empty space.
Break.
Else if there are two of algorithm two's pieces in a line diagonally and an empty space at the top or bottom of the diagonal.
Play the move in the empty space.
Break.
Else if there are two of algorithm two's pieces in a line counter diagonally and an empty space at the top or bottom of the counter diagonal.
Play the move in the empty space.
Break.
Else play a random move
Break.

Figure 3: Pseudocode describing algorithm two's moves.

As a result of the improvements made to algorithm two, the chances of algorithm two beating algorithm one is substantially high since all its moves are no longer completely random.

3.2 Operations the main file

The main file consists of the simpler operations of the programme such as reading in the board sizes, commencing the game, determining the winner, and outputting the necessary information in the required format.

To use the functions that are defined in the class inside the main file, an object must be created. A single object called "play" is created to access all the functions.

A detailed description of the how the functions and other operations work in unison to make the game operable is shown in the flow chart found in appendix A.

4. RESULTS

Since the programme is automated and multiple board sizes can be read in from a text file at a time, 100 iterations of each board size were tested to determine the overall performance of the algorithms against each other. The total number of wins of each player was displayed at the end of the output file once all 100 games were played for each board size.

Table 3: Summarised outcomes of 100 iterations of testing for each board size.

Board Size	Algorithm 1 Wins	Algorithm 2 Wins	Draws
6	8	92	0
7	2	98	0
8	4	96	0
9	6	94	0
10	4	96	0
11	0	100	0
12	0	100	0
13	0	100	0
14	0	100	0
15	0	100	0

The average number of moves taken for each algorithm to win were also determined. The data obtained is summarised in the bar graph below.

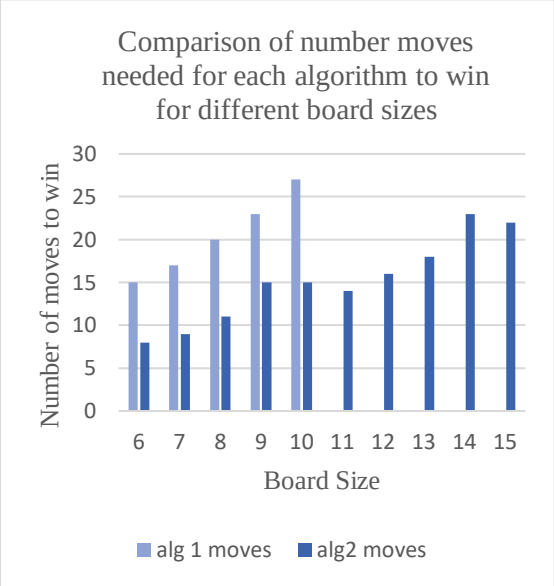


Figure 4: Graph showing the average number of moves needed for the algorithms to win

5.DISCUSSION

5.1 Observations from testing

Since 100 iterations of each board size were tested, the results can be deemed fairly accurate. A pattern is observed in the number of wins obtained for each algorithm. The winner in most games is algorithm two. Algorithm two plays as expected as it plays intelligent moves that increase the chance of it winning. However, it is not deemed unbeatable as algorithm one was able to win a few games on the smaller board sizes. The reason that algorithm was able to win on the smaller boards is because the pool of possible moves that the algorithm can make is smaller and therefore there is a greater chance of algorithm one generating five random moves in a line. Similarly, there is a greater chance that algorithm one plays a random move that prevents algorithm two from winning.

The data displayed on the graph shows that very few moves, in comparison to the board size, are required for algorithm two to win since it recursively searches for the best moves, thus making it more effective. When algorithm one was able to win, many moves were required to reach that outcome and thus playing random moves is not an effective strategy to obtain a favourable outcome.

Overall, the programme is functional and carries out the required tasks. Further improvements can be made to the algorithms to make them more competitive.

5.2 Further improvement

The first algorithm can be improved by allowing it to also search for the best possible move that it can make, similar to how algorithm two functions. Algorithm two can be further improved be allowing it to block the winning opportunities that the opposition has.
This can be implemented using recursive operations.

6. CONCLUSION

The aim of the project was to design and implement two C++ algorithms that play against each other in a strategy board game called *Gomoku*. Numerous rules and considerations were considered during the design process. One of the algorithms designed played completely random moves whilst the other played random moves in the early stages of the game and thereafter played intelligent moves that increased the chances of it winning. After conducting multiple tests, algorithm two proved to be the better algorithm as it won majority of the games in very few moves.

References

[1] wikiHow., How to Play Gomoku.

https://www.wikihow.com/Play-Gomoku, Last accessed 28 May 2022.

[2] ELEN2020-Software Development l, Project Brief 2022, University of the Witwatersrand, School of Electrical and Information Engineering.

[3] IBM., IBM Docs.

https://www.ibm.com/docs/en/zos/2.2.0?topic=functions-srand-set-seed-rand-function, Last accessed 28 May 2022.

Appendix A

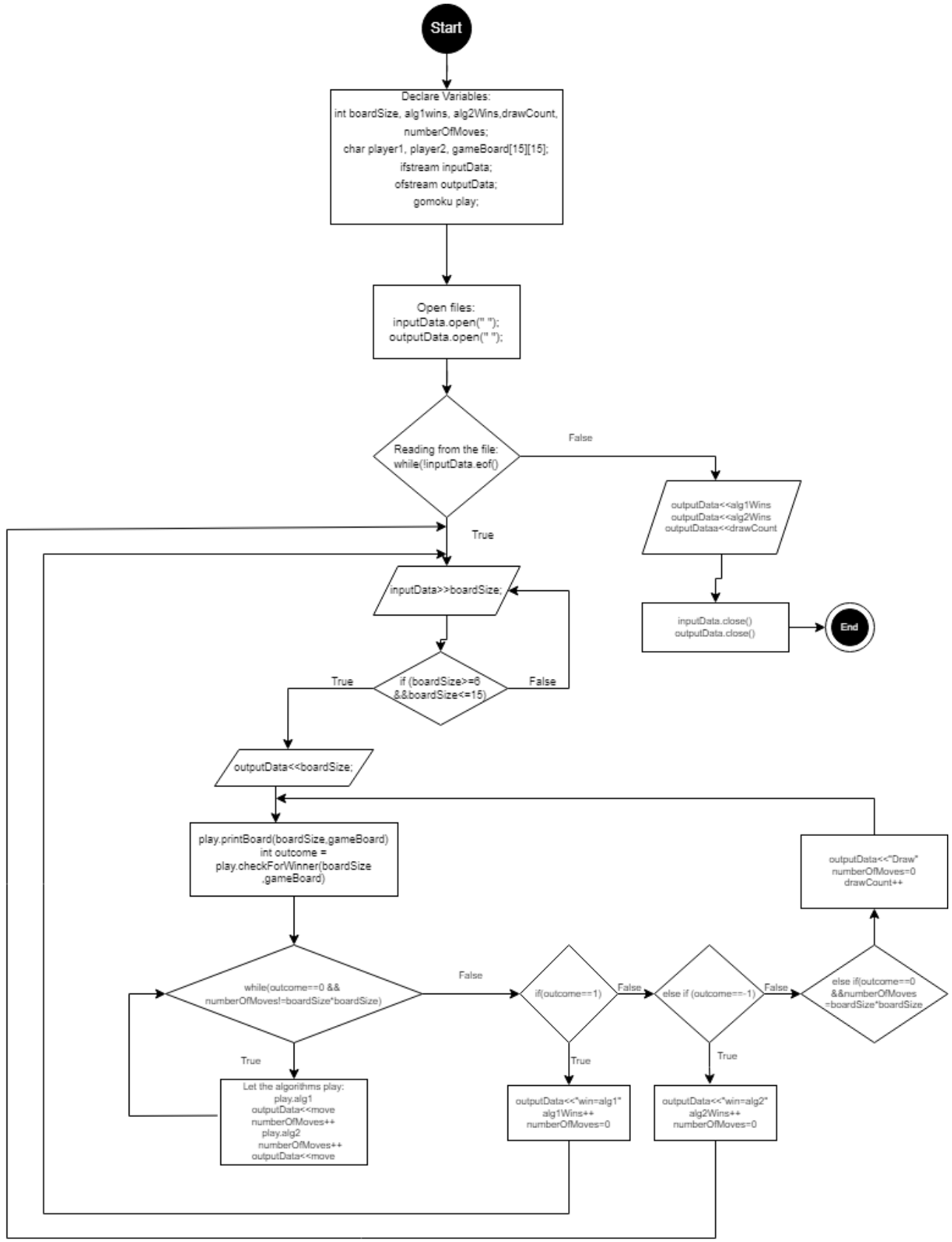


Figure 5: Flow chart of the describing the structure of the main file

Appendix B

The expected duration of the project according to the course outline is 30 hours.

Table 2: comparison and breakdown of the expected time spent on the project and the actual time spent

Task	**Expected time limit**	**Percentage breakdown of expected time limit**	**Actual Time Spent**	**Percentage breakdown of actual time spent**
Background	4.5 hours	15%	2 hours	6.25%
Analysis and Design	6 hours	20%	7 hours	21.875%
Implementation	6 hours	20%	8 hours	25%
Testing	6 hours	20%	5 hours	15.625%
Documentation	7.5 hours	25%	10 hours	31.25%

To understand how the game works, background research was done on the rules. This took 2.5 hours less than the expected time. The design and implementation process were over the expected time by 2 hours. This is due to algorithm two having needed to be redesigned. Due to the automated nature of the projected, the testing period was less than the expected time by an hour. Lastly the documentation process exceeded the expected time by 2.5 hours. Overall, the time was managed quite fairly as the actual time spent only exceeded the expected time by 2 hours.