ELEN2009A: Suicide Checkers in C++

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa

Simphiwe Shezi – 2462667

Abstract – This document presents the implementation of two algorithms that play against each other in the game of suicide checkers in C++. The board size ranges from 6 to 12 and it must be an even number. Algorithm 1 plays randomly whereas algorithm 2 plays at the corners of the board. Algorithm 1 uses the players 'o' and algorithm 2 uses 'x'. The king feature is implemented with characters 'O' and 'X' used for crowned pieces for both algorithm 1 and 2, respectively. Results show that Algorithm 1 performs better than Algorithm 2.

Keywords: *suicide checkers, king*

1. INTRODUCTION

Implementing an algorithm that plays to lose encourages players who are not good at that particular game to continue playing [1]. *Suicide checkers* is one example of a game whereby the aim is to lose. *Suicide checkers*, also known as *anti-checkers*, *giveaway checkers*, and *losing draugh*ts, is identical to conventional checkers but the only difference is that the aim here is to lose [1]. This document presents the implementation and analyses of Suicide checkers in C++. The project specification, design, and implementation of the game are presented in section 2 below, section 3 shows the results, and then the analysis and discussion are done in section 4.

2. SPECIFICATIONS AND DESIGN

2.1. Specifications

This section outlines the project requirements, constraints, and assumptions.

2.1.1. Requirements

It is required that two algorithms be implemented, in C++, to compete against each other for the *Suicide checkers* game. The game accepts an input file with board sizes and records every move made by each algorithm in an output file. The output file contains the size of the current game in the first line, the alternating player moves on the following lines, the number of pieces left for each algorithm, then the winner at the end. A suitable time plan for the project must be formulated.

2.1.2. Constraints

The game, together with the algorithms, must be implemented in C++. The board size must be an even number ranging from 6 to 12. Global variables are not allowed in the code. The board must use the labeling system as per the brief. Console output is not allowed. Pieces only move forward diagonally, towards the opponent. Playing is only allowed on the black part of the board. The king feature is allowed when one player gets to the end of the opponent's side and can move both forward and backward diagonally. A piece making a capturing move jumps the opponent's piece and lands on the following, empty, diagonal position. A piece is only allowed one move up if it is not capturing anything. A piece must be removed from the board after it has been captured. A capturing move is compulsory, an algorithm must capture whenever there is capturing opportunity.

2.1.3. Assumptions

Algorithm 1 will use the character 'o' to play and algorithm 2 will use the character 'x' to play. Algorithm 1 uses 'O' to denote king and algorithm 2 uses 'X' to denote king. Pieces are not allowed to move beyond board boundaries. Multiple jumps are allowed as long as there is capturing opportunity after a jump. An algorithm can choose one out of multiple capturing opportunities. Assume the '.' characters represent part of the board that playing is not allowed and '#' Characters represent the black part of the board.

2.2. Design and Implementation

2.2.1. Board

The board is designed with a 2-D vector that is inside the main function to allow manipulations by other classes and functions to manipulate it. A vector allows for flexibility in size. The vector is initiated with '.' characters and '#' characters in alternating turns in a class called Board. Figure E1 in appendix E below shows an example output of this, using a board of size 12. Figure 1 below shows the UML diagram for the 'Board' class.

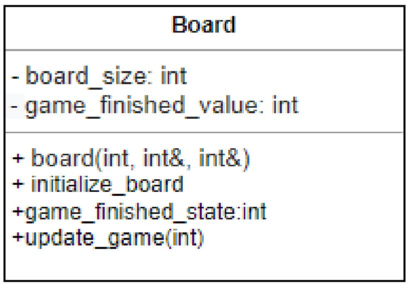


Figure 1: UML diagram for the class Board

The 'Board' class is used to size and initialize components of the game. A constructor is used to validate and set the board size, making sure it is even and ranges from 6 to 12, and to calculate the number of initial players according to equation 1 below.

$$player = \left(\frac{size\_of\_board}{2}\right) * \left(\frac{size\_of\_board - 1}{2}\right) \qquad (1)$$

This equation makes sure that only the middle two rows are available for play, and the board is shared evenly by both players. The 'initiate_board ()' function fills the board with the characters as per discussed above, algorithm1 as the character 'o' and algorithm2 as the character 'x'. The "game_finished_value" can have three values to determine and manipulate the "game_finished_state" function which returns this value; 0 means that the game is playing, 1 means player1 has won, 2 means player2 has won, and 3 means there was a draw. The value is private, and therefore is manipulated using the "update_game()" function.

2.2.2. Game Rules

The rules of this game were all implemented boolean functions inside another class called "Rules". Figure 2 below shows the members of this class. The "is_forbidden()" makes sure that play does not happen beyond board boundaries. It is private because it is used by member functions only, to perform high-level operations. "is_king" function validates if a player is a king or not, it is also used by member functions. A piece that is crowned king can move any direction diagonally and can capture pieces at any direction as long as it is diagonal and valid the king uses 'O' for player 1 and 'X' for player 2. The "Rules" class makes sure that moving up and down only happens diagonally based on the players and, their restricted movements and their positions, and this is achieved by the use of "move_left_up_diagonally, move_left_down_diagonally, etc". The class also enables jumping depending on whether player1, player2, or a king is playing using the "is_jumping()" function which returns true after a jump and false otherwise. The class also enables a piece to be king if it ever gets to the boundaries of the opponent algorithm.

2.2.3. Algorithm1

This is the random algorithm as per the brief. Both algorithms start off by checking possible jumps. The jumps might be multiple, and after the jump/s, the algorithm hands over the turn to the opposite algorithm. If no jump was possible, this algorithm declares six empty vectors which will store start positions and end positions. One vector stores the start positions rows, another the start positions columns, the other holds destination position rows, and another, the destination columns. These four vectors are used to convert to two other vectors that store the start position as a number, then the end position as a number as well. Figure B1 in Appendixelow shows the flow chart of this algorithm. Figure 2 below makes an emphasis how important the vectors are.

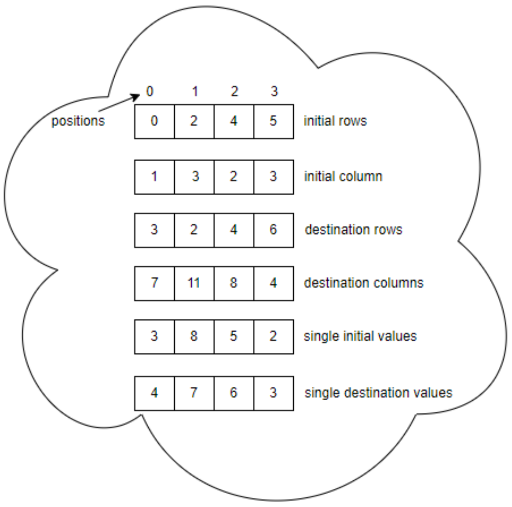

positions
0
1
2
3
0 2 4 5 initial rows
1 3 2 3 initial column
3 2 4 6 destination rows
7 11 8 4 destination columns
3 8 5 2 single initial values
4 7 6 3 single destination values

Figure 2: Possible moves vectors

The common positions of the vectors make it easy to convert and access the board, for instance, at position 0, moving from 3 to 4 is the same as moving from (0,1) to (3,7). Random numbers are used just for illustration. The single-valued coordinates are obtained using equation 2 below.

$$coordinate = row * \frac{board\_size}{2} + \frac{column}{2} + 1 \quad (2)$$

The $\frac{column}{2}$ part is rounded off such that 0.5 = 0, 1.5 =1, etc, using the C++ 'floor' function. This formula illustrates that the board is similar to a 1D array of numbers which are divided by $\frac{size}{2}$ pairs and every new pair adds a sequence of numbers (0,1,2,3,…) depending on the size. This is all illustrated in figure 3 below with a size 6 board.

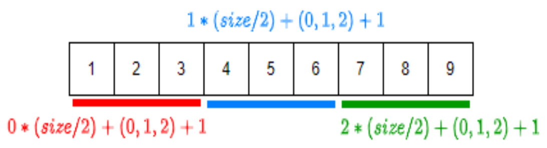


Figure 3: 1D coordinates

This vector represents all positions of the 2D board.

2.2.4. Algorithm2

This algorithm also assigns those six vectors the same way as the previous algorithm. This algorithm checks for possible jumps as well. This algorithm plays by checking for positions nearest to the sides (columns). This will then enable the player to target (seek) for jumps, which might trap the other algorithm. Figure B2 in Appendix B below shows this.

2.2.5. Game

An even board size which ranges from 6 to 12 is read from a file. A counter is then used to facilitate whichever algorithm starts. If the counter is 1, algorithm 1 starts and the opposite is true. Each algorithm plays while recording its move to an output file. Validations checks, such as, checking for boundaries and checking if a player must be crowned king are made as each algorithm runs. The game plays while checking and updating if the game is over. The game is over under 3 conditions, player 1 winning, player 2 winning, or a draw. The conditions for a draw are if the algorithm that is playing cannot move or both algorithms cannot move. Figure B3 in Appendix B below shows the flow diagram to demonstrate this.

2.2.6. Time management

It is estimated that due to the project worthing 30%, a minimum of 1.5 hours per day over a period of 30 days would be required making the overall project to be completed in 45 hours. However, during the course of the project, it was discovered that the project needed more time, especially the implementation and documentation. Therefore, the project ended up taking 111 hours, and Table 1 in appendix A shows the time spent on each task. It can be observed that the actual time was more than double the estimated time for each task.

3. RESULTS

The code was implemented in C++ on Codeblocks 20.03 IDE . The computer at which it was run had the following properties:

- RAM: 4.00GB
- Processor: Intel (R)
- System type: 64-bit operating system
- Frequency: 1.10 GHz

The time taken to run each code was highly dependent on the board size and the number of tests performed. To run 50 tests, it took 9.831s and 239.2165s for board sizes 6 and 12, respectively. 50 tests were performed for all possible board sizes and the results are presented in figure 4 below.

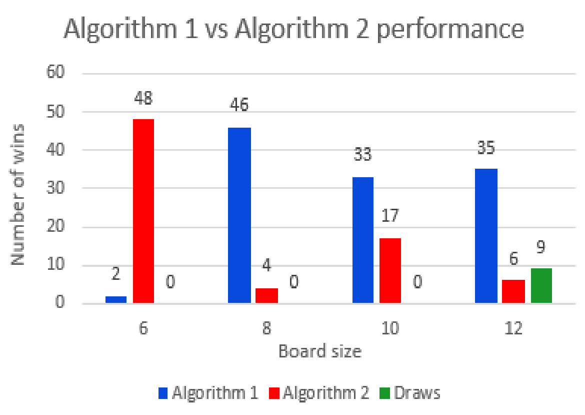


Figure 4: Results of the algorithms' performance when algorithm 1 starts.

The results show that algorithm 2 is better than algorithm 1 only on board size 6. Algorithm 1 performs much better than algorithm 2 on a size 8 board. Algorithm 1 continues to be better than algorithm 2 as the sizes increase but there is an improvement in performance. Algorithm 2 starts performing just fairly when the sizes are too bigger (sizes 10 and 12), algorithm 1 however, wins more than algorithm 2 even with bigger sizes. This is because algorithm 2 always chooses positions closer to the corners, making it difficult for algorithm 2 to lose its pieces. Algorithm 1 on the other hand, is random and there are very high chances of it being all over the board, increasing its chances of losing pieces. Multiple jumps are highly likely for the random algorithm since it is all over the place and can lose its pieces more often than algorithm 1. The previous results are when player 1 was starting. Figure 5 below shows results when player 2 starts.

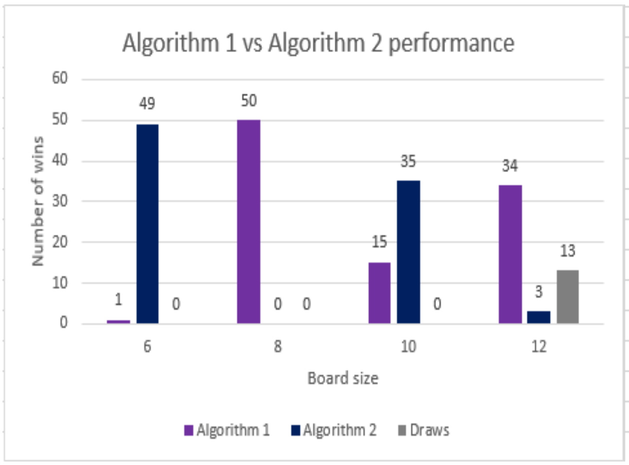


Figure 5: Results of the algorithms' performance when algorithm 2 starts.

The first thing that was observed is that when player 2 starts, the time it takes to complete the code is much shorter on lesser sizes. Size 6 board took 0.471 seconds. Size 12 executed in 175.081 seconds, which is better than when algorithm 1 was starting. The first results do not change, algorithm 2 still performs better at size 6. There is a flip of performance when the size went to 10 and, Algorithm 2 performed better than algorithm 1. Algorithm 1 continued dominating when the size was 12. Another thing to observe is that the draws start happenning when the board size increase. For both graphs, draws were only observed on board size 12.

4. DISCUSSION

From the previous section, it can be shown that an algorithm that scatters its players, like the random algorithm, has a high chance of winning more than one that stays in corners or one that has its pieces all in one place. The assumption when implementing algorithm 2 was that the pieces at the corner would trap the opposite algorithm by forcing multiple jumps, through sacrifices. It was found that algorithm 1 was hard to collect pieces from. This is because as the pieces are grouped through the corners, they defend jumps against opponents. Multiple jumps are highly likely when the pieces are scattered. An algorithm that scatters its pieces is more reliable. An algorithm that would first check its fellow pieces and try to fill the board would be more effective for this game than an algorithm that forms a group.

5. CONCLUSION

Two algorithms were implemented in C++ to play against each other in a game of suicide checkers. Both algorithms firstly check for possible jumps before making any move. Whenever a jump is possible, the algorithm loses its turn, assuming that it played the jump. Algorithm choses a random position whereas algorithm 2 choses positions near corners. Results shows that algorithm 1 performs better than algorithm 2 because algorithm 1 pieces are scartted all over the board. Algorithm 2 only performs better on the smaller size board. Discussions were made and recommendation made was to have an algorithm that would first check its fellow pieces and try to fill the board would be more effective for this game than an algorithm that forms a group. The implementation of the game of suicide checkers was a success.

REFERENCES

[1] Bosboom, J., Congero, S., Demaine, E.D., Demaine, M.L. and Lynch, J., 2019. Losing at Checkers is hard. *The Mathematics of Various Entertaining Subjects (MOVES 2017)*, *3*, pp.103-118.

APPENDIX A: Project time management

Table 1: Project estimated and actual time

Tasks	Estimate time(hrs)	Actual time(hrs)
Background(Research)	6.5	11
Analysis & Design	9	15
Implementation	9	45
Testing	9	10
Documentation	11.5	30

APPENDIX B: Flow charts

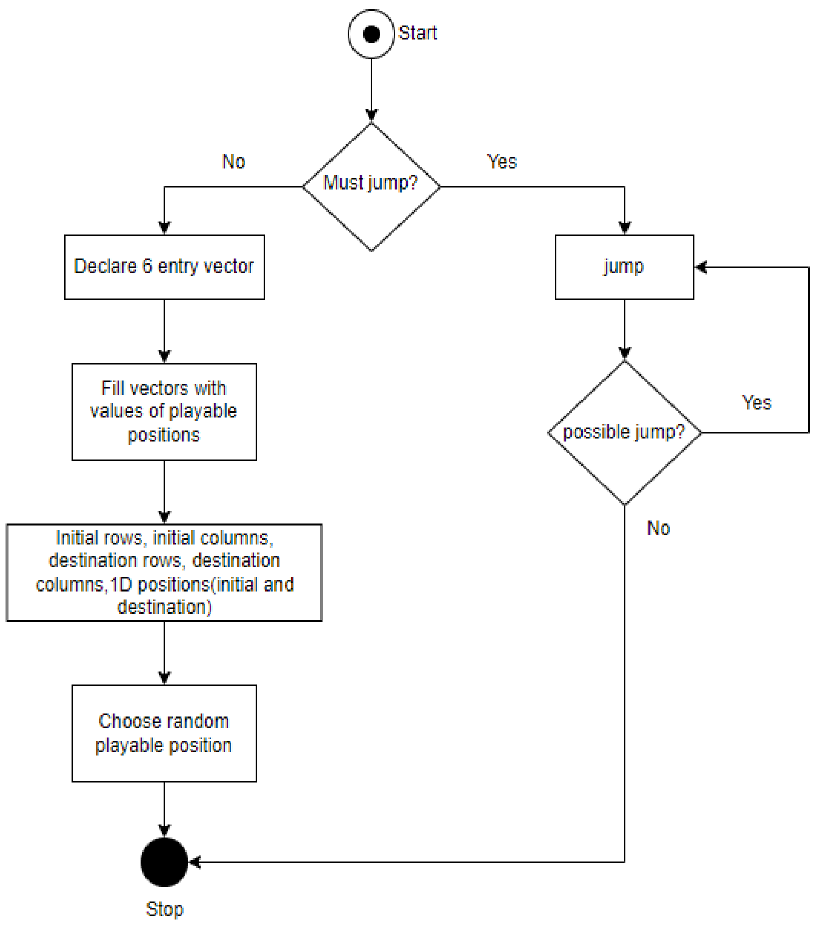

Start
Must jump?
No
Yes
Declare 6 entry vector
jump
Fill vectors with values of playable positions
possible jump?
Yes
No
Initial rows, initial columns, destination rows, destination columns, 1D positions(initial and destination)
Choose random playable position
Stop

Figure B1: flow chart for algorithm 1.

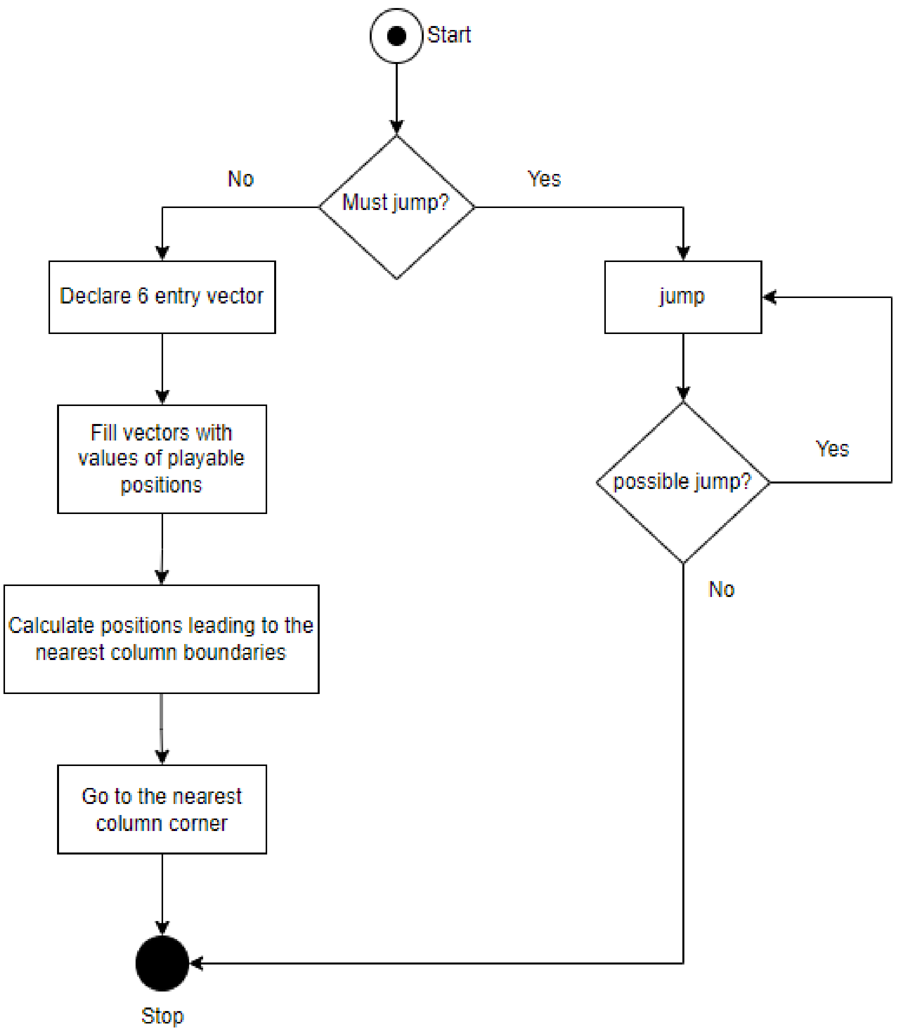

Start
Must jump?
No
Yes
Declare 6 entry vector
jump
Fill vectors with values of playable positions
possible jump?
Yes
No
Calculate positions leading to the nearest column boundaries
Go to the nearest column corner
Stop

Figure B2: Algorithm 2 flow chart

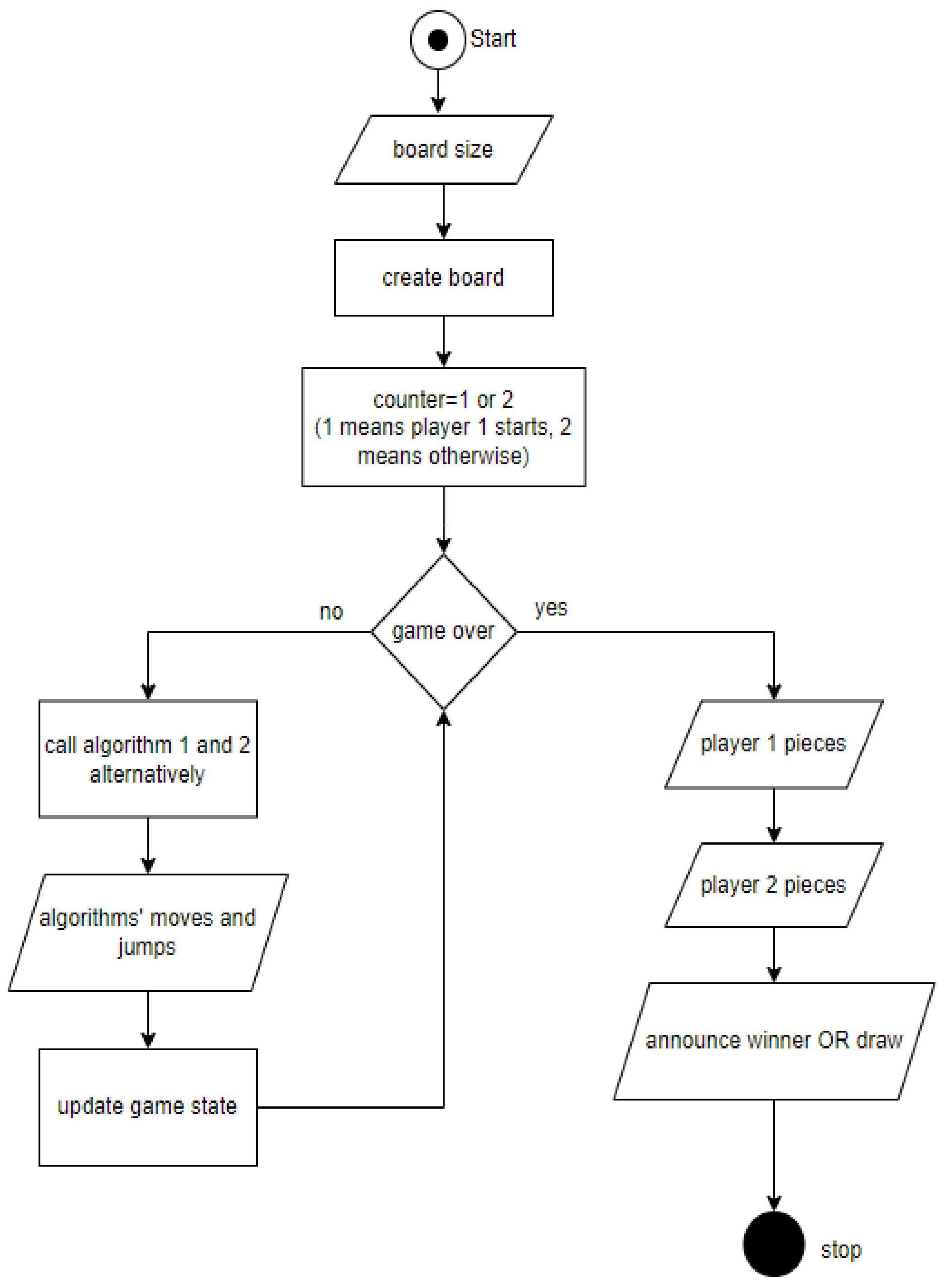
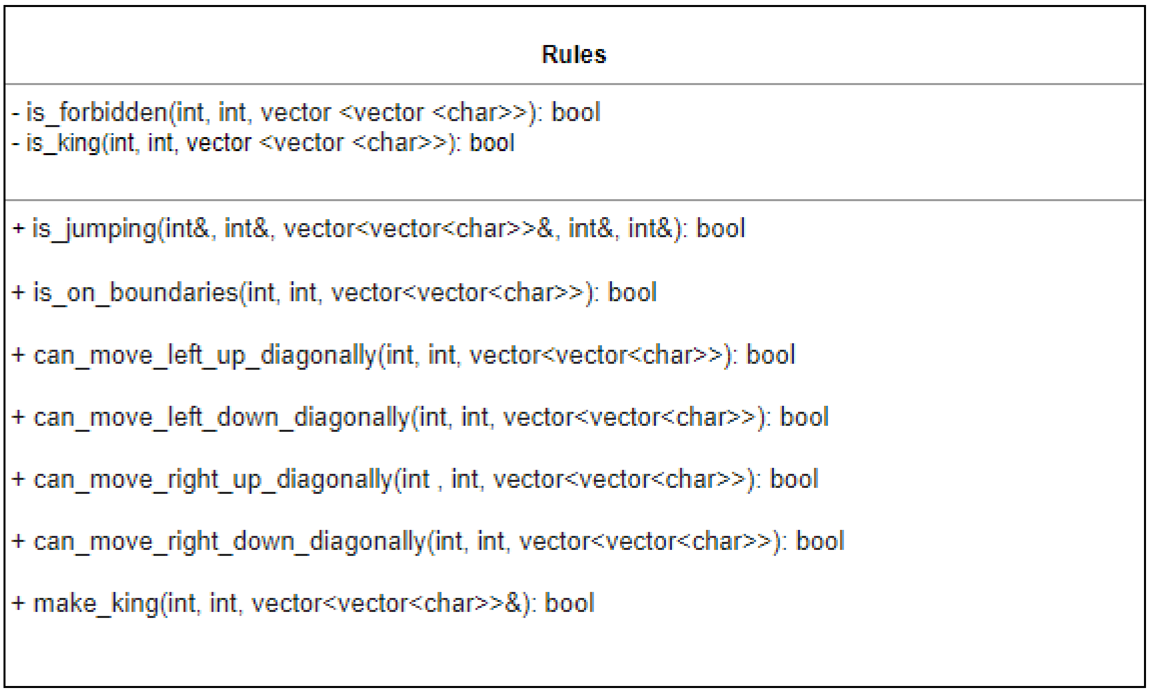


Figure B3: code flow diagram

APPENDIX C: UML diagram

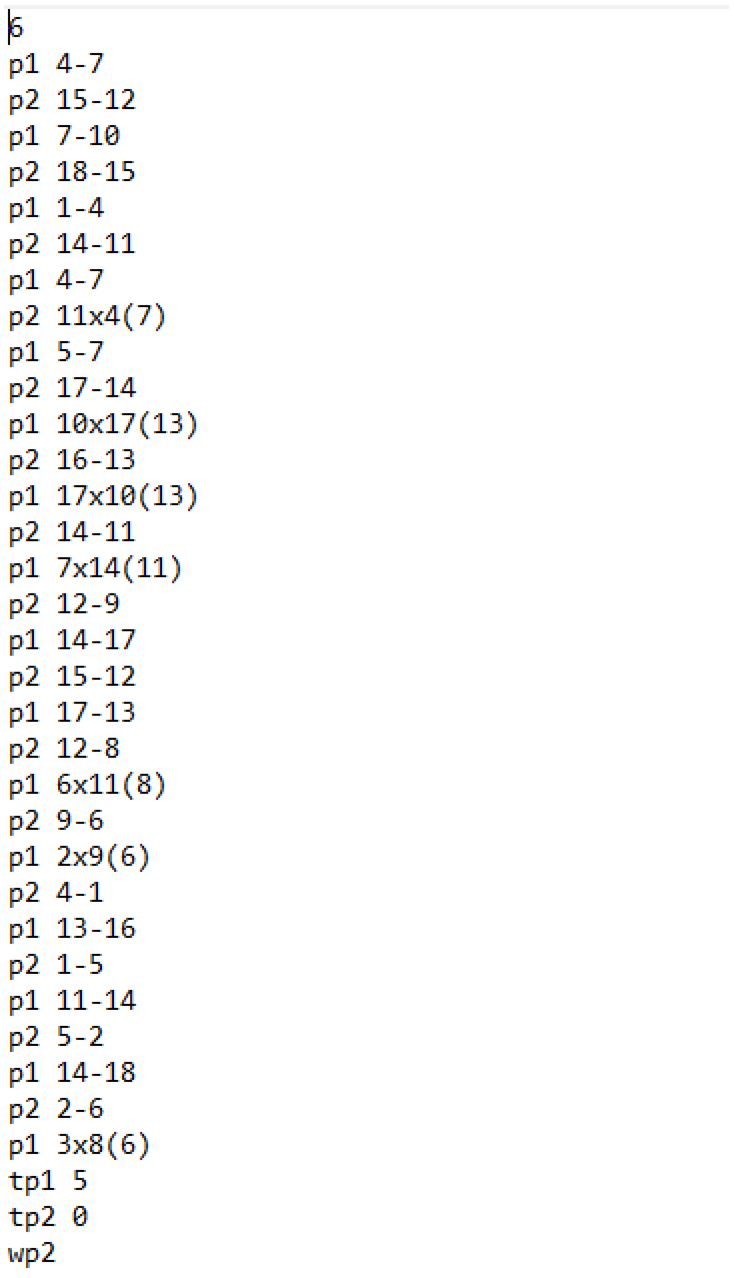


Rules

```
- is_forbidden(int, int, vector <vector <char>>): bool
- is_king(int, int, vector <vector <char>>): bool

+ is_jumping(int&, int&, vector<vector<char>>&, int&, int&): bool

+ is_on_boundaries(int, int, vector<vector<char>>): bool

+ can_move_left_up_diagonally(int, int, vector<vector<char>>): bool

+ can_move_left_down_diagonally(int, int, vector<vector<char>>): bool

+ can_move_right_up_diagonally(int , int, vector<vector<char>>): bool

+ can_move_right_down_diagonally(int, int, vector<vector<char>>): bool

+ make_king(int, int, vector<vector<char>>&): bool
```

Figure C1: UML diagram for the class Rules

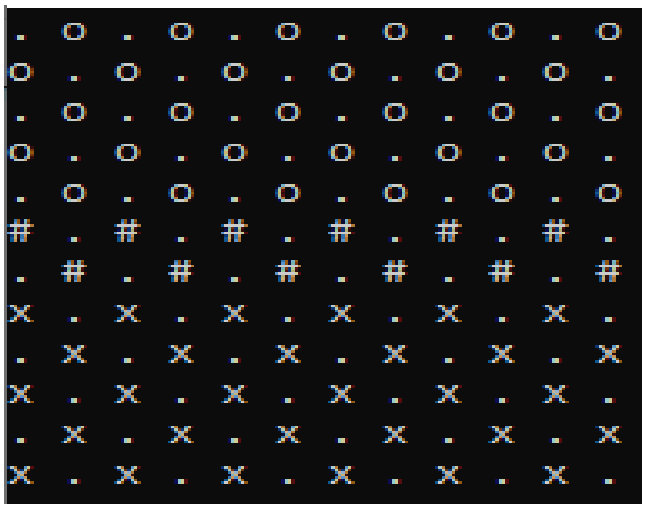
APPENDIX D: Text files



```
6
p1 4-7
p2 15-12
p1 7-10
p2 18-15
p1 1-4
p2 14-11
p1 4-7
p2 11x4(7)
p1 5-7
p2 17-14
p1 10x17(13)
p2 16-13
p1 17x10(13)
p2 14-11
p1 7x14(11)
p2 12-9
p1 14-17
p2 15-12
p1 17-13
p2 12-8
p1 6x11(8)
p2 9-6
p1 2x9(6)
p2 4-1
p1 13-16
p2 1-5
p1 11-14
p2 5-2
p1 14-18
p2 2-6
p1 3x8(6)
tp1 5
tp2 0
wp2
```

Figure D1: Example of the output file of a 6 × 6 board.

APPENDIX E: Console references



```
. O . O . O . O . O . O
O . O . O . O . O . O .
. O . O . O . O . O . O
O . O . O . O . O . O .
. O . O . O . O . O . O
# . # . # . # . # . # .
. # . # . # . # . # . #
X . X . X . X . X . X .
. X . X . X . X . X . X
X . X . X . X . X . X .
. X . X . X . X . X . X
X . X . X . X . X . X .
```

Figure E1 : 12x12 board in console