Nduvho E. Ramashia 1490804

School of Electrical and Information Engineering

University of the Witwatersrand, Johannesburg

ELEN2004/ELEN2020 - Software Development I

27 MAY 2020

C++ ANTI-CHECKERS IMPLEMENTATION

Abstract: This report explores procedures taken in implementing the computer game Anti-Checkers. Two algorithms namely algorithm 1 and algorithm 2 are created to play the game. The two algorithms compete against each other on boards of different sizes. The moves of each algorithm are stored on different lists for efficiency and to save memory program memory. Different shapes of graphs are obtained to see which of the algorithms is more efficient. Algorithm 2 is better than algorithm 1. The program runs faster on Linux OS compared to Windows OS.

1. INTRODUCTION

This report shows an approach used in developing a C++ based computer program/game, Anti-Checkers. Anti-Checkers, also known as Suicide Checkers, Losing Droughts and Giveaway Checkers, is a type of a board game Checkers were the main objective or goal of the game has been reversed, i.e. players win when they by the Checkers rule loose.

Players win when they have lost all their pieces. If both players still have pieces on the board but cannot move, the game ends in a draw. If it is player's turn and the players still have pieces on board, but with no valid move, the game ends in a draw.

Two different algorithms are required for playing the game. The two algorithms take turns playing until one of them wins or when they come to a draw. The maximum number of pieces each algorithm can jump in one turn is 2. The two algorithms play against any board size between and including 6 and 12.

The board size is obtained from a file that contains an unknown number of lines/board sizes.

The program plays the game for each obtained board size saving the made moves and results to a file. The contents of the file are then examined to track the efficiency of each algorithm.

The program is then experimented on to see the effects of different aspects of the program. Slight changes are made to see their effects on the program efficiency and factors that affect it.

2. IMPLEMENTATION

2.1. Design

2.1.1. The main program

To make a playable board game, two boards were created first. The two boards are of C++ CHAR and INT type. The CHAR board is for storing the players' pieces while the INT board stores the positions numbering. The boards are initialized as vectors. The vectors are chosen as the storage containers because of their flexibility and ease of manipulation [1].

Vectors are also preferred over arrays because they eliminate the need to create large fixed arrays, wasting memory. They do so because they can shrink and grow during execution [1].

To play the game, two algorithms were created and implemented in C++. The two algorithms use the same function for finding available moves. That function however uses different methods to find moves for each algorithm depending on the algorithm that needs the moves. Three lists are created for storing the valid moves. Different lists store different types of moves. One list, “list 0”, is for storing non-compulsory moves, the other, “list 1”, is for storing moves that involve 1 jump. The other, “list 2” is for storing moves involving 2 jumps.

Different moves require different sizes of data. They require a different amount of information, as shown in Table 1 below. List 0, List 1, and list 2 refer to non-compulsory moves, moves involving 1 jump, and moves involving 2 jumps, respectively.

As shown in Table 1 below, all 3 types of moves each need a different amount of detail.

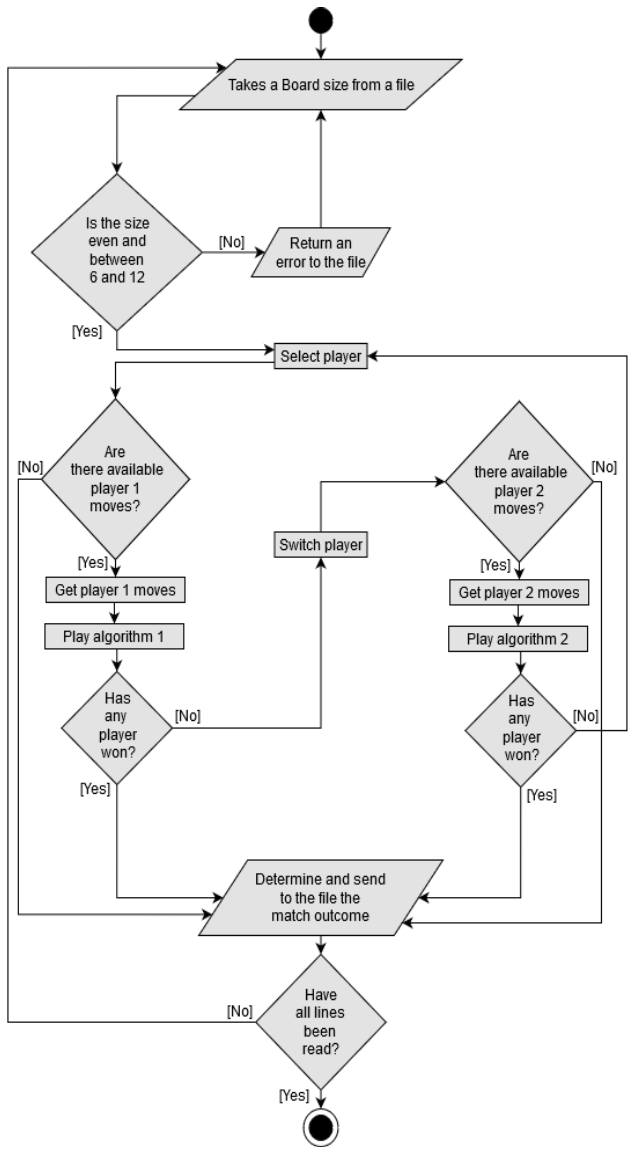
Table 1: Information needed by different types of moves.

List 0 moves	**List 1 moves**	**List 2 moves**
Initial position	Initial position	Initial position
Final position	Jumped position	Jumped position
	Final position	Jump to position
		jumped position
		final position

It is because of this difference in data needed my different moves that the moves are stored on separate vector lists. This is to make sure that each list is always filled with needed data only. It is to make sure that there is no case where a vector stores zero only to make sure that the vector size is valid. This storing of only the needed data in turn saves memory. This then makes the program lighter. It also saves a relatively small amount of time from not taking time to store unnecessary data.

The program fills the lists of available moves by going through the board containing the players’ pieces, searching for moves, and storing them in appropriate lists depending on the size of data needed to make the move.

These lists are emptied after each use my any algorithm. This is to make sure they are empty and ready for the next player. The overall functionality of the program is illustrated in figure 1 below.



Takes a Board size from a file
Is the size even and between 6 and 12
[No]
Return an error to the file
[Yes]
Select player
Are there available player 1 moves?
[No]
[Yes]
Get player 1 moves
Play algorithm 1
Has any player won?
[No]
[Yes]
Switch player
Are there available player 2 moves?
[No]
[Yes]
Get player 2 moves
Play algorithm 2
Has any player won?
[No]
[Yes]
Determine and send to the file the match outcome
Have all lines been read?
[No]
[Yes]

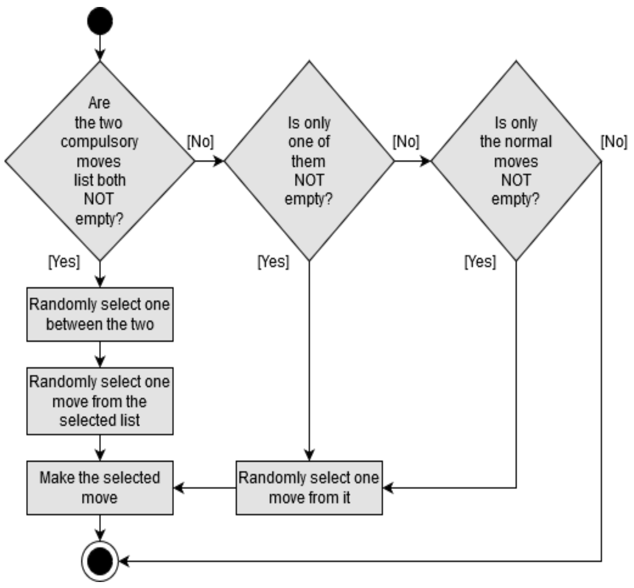
Figure 1: Main program functionality.

As illustrated in figure 1, the two players/algorithms take turns playing/making moves for as long as neither of them has won the match or ran out of moves to make.
When one of the players has won, the program gets another board size from the file if there is any left. The two algorithms then repeat the same thing with new received board size.

2.1.2. Algorithm 1

Algorithm 1 randomly selects its moves. When two lists created for storing compulsory moves both have moves in them, the algorithm first *randomly* selects one of the lists. It then *randomly* selects one of the moves from the randomly selected list.
The two lists are examined first to make sure that compulsory moves are made first always before non-compulsory moves.

The full functionality of algorithm 1 fully shown in figure 2 below.



Are the two compulsory moves list both NOT empty?
[No]
Is only one of them NOT empty?
[No]
Is only the normal moves NOT empty?
[No]
[Yes]
[Yes]
[Yes]
Randomly select one between the two
Randomly select one move from the selected list
Make the selected move
Randomly select one move from it

Figure 2: Algorithm 1 moves making mechanism.

The algorithm executes its move by removing the jumped pieces from the pieces board and moving its pieces to the final positions from the list on the board.

2.1.3. Algorithm 2

Algorithm 2 makes the first move it finds. However, to minimize the number of the opponent's pieces removed, it jumps a minimum number of pieces possible. When examining the compulsory moves lists, it first examines list 1, the list containing moves that involve only 1 jump. If this list not empty, it then takes a move from it. this eliminates the chance to jump 2 of the opponent's pieces, which increases the opponent's winning probability. Figure 3 below shows fully the computation and decision making involved in making algorithm 2's moves.

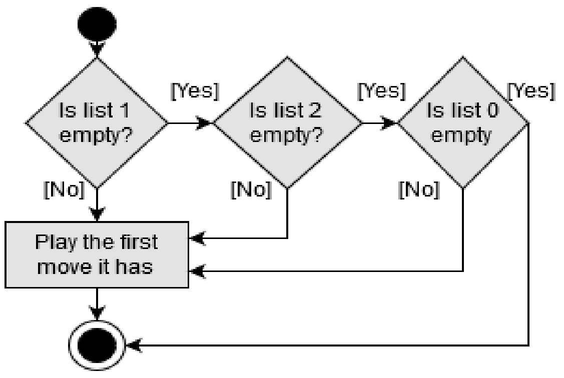


Figure 3: Algorithm 2 moves making mechanism.

2.2. Results

After a thousand runs of the program, saving the status of each list for all the turns/moves, the graph in figure 4 was obtained, showing how frequent each list was available for usage.

The figure was plotted with RStudio. A total of 257 819 points were used in doing so. The points were obtained from running the program 1 000 times. This is achieved using a for loop (i.e. for i in {1..1000}; do ./run; sleep 0; done).

As the graph shows, there is never a case, board size, or scenario where any of the algorithms need to store data for 3-jumps moves. This was also manually observed using a display function that sent the pieces' board before every move. It was at the end concluded that both algorithms never need the 4th list, the list for storing the 3-jumps moves. Because of its obsceneness, the part was removed to save time and space.

The figure also shows how often list 0 (no jumps list) has moves for the algorithm to select from. It also shows how rare it is for list 2 (2-jumps moves) to have moves in it.

This shows how unlikely it is for the two algorithms to have moves involving jumping over 2 pieces. This in turn increases the program run time. That is because the fewer jumps involved, will delay a win, or draw since there still more pieces on the board.

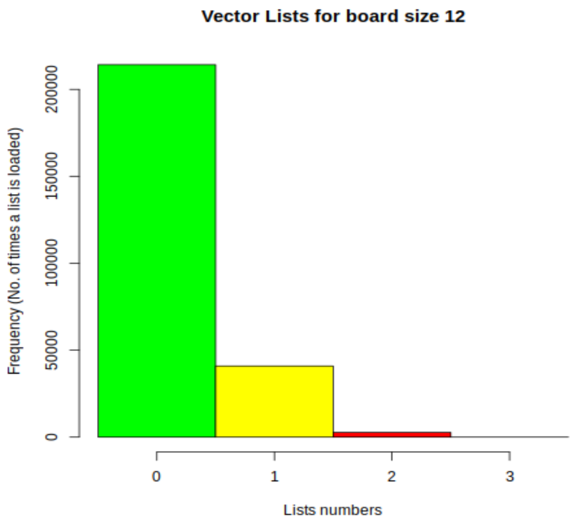


Figure 4: Lists availability for usage frequency

Figure 5 shows the outcomes of the game when the matches are played with no time difference between them. When a command 'for i in {1..1000}; do ./run; sleep 0; done' is used. Most of the matches end up in a draw. Algorithm 2 wins the ones that do not end up in a tie. Algorithm 1 does not win any. This shows how inferior algorithm 1.

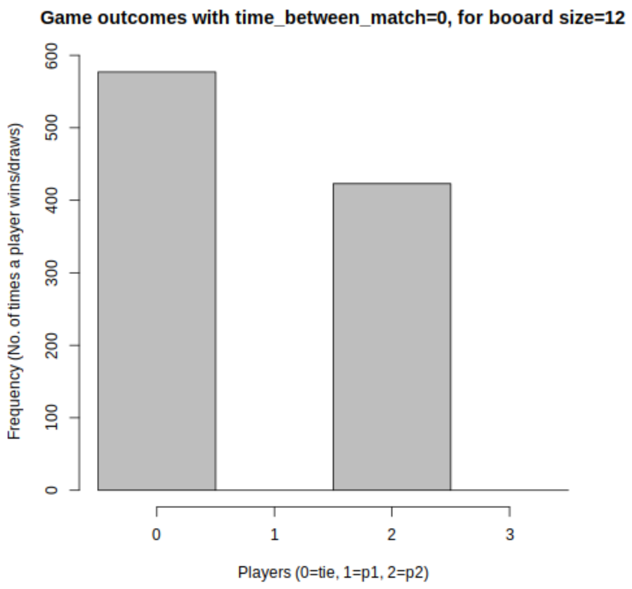


Figure 5: Players winnings after a thousand games

The outcomes in figure 5 however are also time dependent. It appears the less time there is between 'runs', less likely algorithm 1 will win. This can be seen in Figures 6 and 7. It can be seen there that as there is the time between the runs, algorithm one wins more, compare to when there is no time difference. However, this change in probability only increases for time 0s to 5s, after which it remains the same.

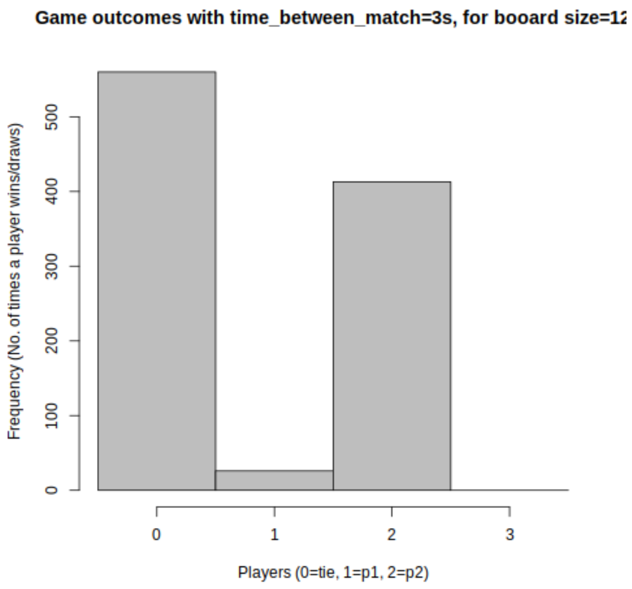


Figure 6: Players winnings for time-separated games

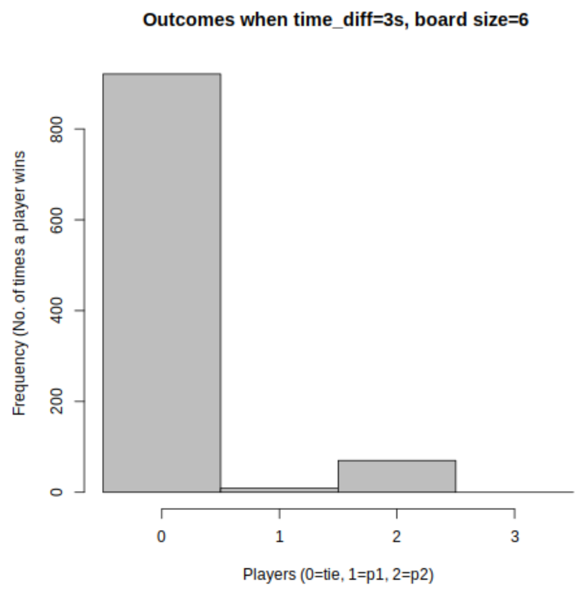


Figure 7: Players winnings for time-separated games

It is concluded from figure 5, figure 6, and figure 7, that the time between consecutive matches increase affects the winning chances for algorithm 1.

Many more tests were run to support this relationship. Figure 6 is one of the results proving it.

After numerous tests, it was concluded that for a computer of the same hardware specifications, the programs run faster on a Linux Operating System (OS) compared to a Windows Operating system. The program takes 0m.047s on average to run on Linux OS and 0m.104s on Windows OS.

2.3. Critical analysis

The devised solution was is well implemented. The program does all the required tasks. The two different algorithms are fully functional and well playing the Anti-Checkers game. The Anti-Checkers game ends as required.

It was planned for the program to take a small amount of time to run as possible. Because of that, the program had to be semi-smart. As a result, the program came out not so smart but not too simple either. It also turned out to have a significantly good runtime, as planned. It is because of that good/short runtime and its algorithms being not too simple that it made the program well within an acceptably expected quality range.

The project time breakdown, planned and actual, can be found in Appendix A.

2.4. Future recommendations

- The algorithm that makes the first move could be randomized.
- A different and smarter algorithm could be used in place of Algorithm 1.
- Algorithm 2 could be improved to make smarter moves, to make smarter moves.
- Kings could be implemented to reduce the number of ‘tie’ outcomes.

3. CONCLUSION

The program was fully implemented. The program has got an average runtime of 0.047s when run on Linux OS. It has an average of 0.104s when run on Windows OS. The two algorithm ends in a tie most of the time. Algorithm 1 winning probability increase with the time it between consecutive runs. This increase however is only significant between time 0-5s. Algorithm 2 play better than algorithm 1. Given more time, the program could have better smarter replacements and improvements to its algorithms.

REFERENCES

[1] MALIK D S. *C++ programming: from problem analysis to program design*. Cengage Learning Stamford, CT, Seventh Edition, 2015. Pp 1069.

APPENDIX A

Task	Planned(done by date)	Actual
Reading	29 feb	28 feb
Board	07 mar	10 mar
Algorithm 1	21 mar	21 mar
Algorithm 2	28 mar	25 mar
Playing	11 apr	17 apr
Testing	02 may	06 may
Report	14 may	22 may
Modifications	15 may	24 may

Activity	Time Breakdown(planned)	Actual
Background (problem understanding, requirements)	15%	15%
Analysis & Design	20%	22%
Implementation (coding)	20%	27%
Testing	20%	19%
Documentation	25%	17%