Algorithms and Data Structures - Tic Cat Toe

Zsolt Varga
40212393@napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET08122)

Figure 1: Welcome screen of the program

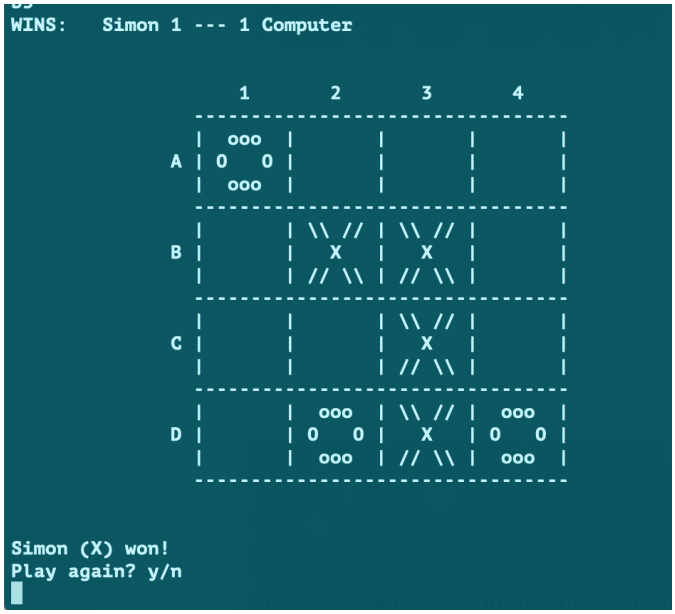

Figure 2: Number of wins kept track of during a game, with the option to play again after a win/tie

1 Introduction

The aim of this project was to implement a working Tic-Tac-Toe (or Tic Cat Toe 1) game in the C language, while keeping in mind the performance and overall cost of used algorithms and data structures. We have implemented a fully functioning game with several additional features, on top of the required base elements. These are as follows.

The game can be played on a board and keep score of wins after the game has been won (figure 2). Players can opt to play with a computer instead of a real person. The board size can be customized to any size between 3x3 and 10x10. Similarly, the number of marks in a line (winscore) required for a win can also be customized to any size between 3 and the size of the board. Players are able to insert their name, which along with other details of the game save to the list of previous games. An undo/redo functionality is also an option during the game. Past games can be replayed and navigated through from the first up to the last (winning) move. Naturally, past games can also be browsed to allow selection for replay. All of these functionalities are tied together in a classic-style main menu, where users are given options along with starting a new game.

Our challenge was to implement these features using algorithms and data structures that suit the problems of the game and therefore maximize performance, and perhaps minimize spacial cost. However, the latter was less of a concern considering the storage sizes on current computer architectures.

2 Design

The very base architecture of the game is a simple while loop, which depends on a gamestate variable that changes based on users turns. The way entities such as players and boards, win checking and computer player were implemented each add a level of abstraction. In this section, we will discuss these in the order they were implemented and build up our program to its current state.

2.1 Board

The board was initially implemented as a static two-dimensional array of integers. When the board is drawn, it's traversed row by row and displayed on the screen. The downside of this approach might be the space a board might take up on our storage. There is no way we can guarantee a size of an n x n sized board can be stored, as an array would require a consecutive chunk of space. Most importantly, with

this approach it wouldn't be possible to change the board size during runtime, which led to implementing it as a 2D array using pointers (int ** board) allocated dynamically.

2.2 Pieces/Marks

Since our board holds integer values, it means our marks are represented as integers. These are declared as non-zero constants MARK_X and MARK_O, which makes the code for handling them very readable. The values of these constants have been set to 1 and 2 (to correspond with Player 1 and 2) but are actually unimportant, as long as they don't match.

2.3 Player

It is only the player's mark (MARK_X or MARK_O) which is stored in the board. On top of this, players have other attributes that need to be kept track of - their name, number of wins so far, and whether they are automated. For this reason, players have been implemented as structs, which do indeed require more storage than individual variables.
However, there are always only two instances of these structs kept in the memory (as there are only two players). To keep track of who's turn it is during the game, we have created a pointer to a player struct called current_player. Passing this pointer to any function that needs to access the player eliminates the need to create unnecessary copies of our players in the memory.

2.4 Turns and History

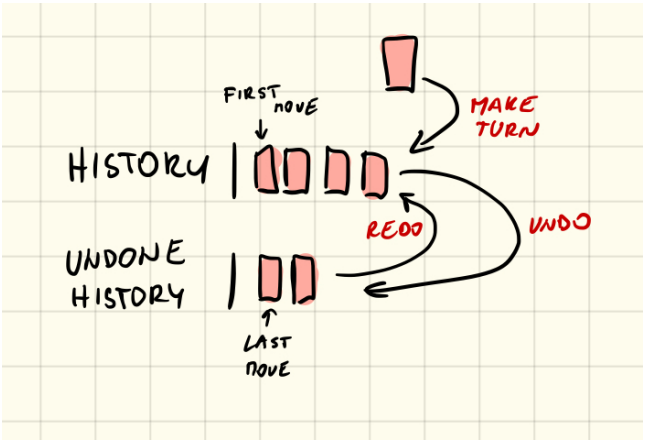

Figure 3: The way history and undone history are implemented to allow undo/redo functionality

Another important entity in our game were turns. A turn holds a pointer to the player who made it, as well as the row and column where the player placed their mark. Most importantly, a turn holds a pointer to another turn struct, which leads us to our first data structure. The main reasons turns were implemented, was so that we can keep track of the order in which marks were put in the game board - history. This will be crucial for the undo/redo functionality.

To create a history, a few different approaches have been considered, each with their pros and cons. Firstly, a simple array holding pointers to turns could have been implemented. We know the size of the history will never be bigger than the total number of slots in the board, therefore there would be no need to worry about having to increase the size of the array. Also, this would make life easy by having the power of random access to any point in the history.

However, we have chosen to implement this as a stack using a singly linked list for the following reasons. Even though traversing to the very end of the history would take longer than it would in an array, we will never actually need to do this. When undoing a move, we only ever need the top of the stack (as the undone move pops off the stack) and therefore only dereference one pointer.
This works perfectly for a singly linked list, we can go back one move each time until our history points to NULL. But what about redoing a move? We could transform our singly linked list to a doubly linked list to be able to move in each direction, but this would mean increasing the size of our structs as well as contradicting what has just been said about no need for traversing the history. Instead, two stacks have been initialized - one for the history, and one for undone moves (see figure 3). When a move is undone, it pops off the history and pushed into the undone moves and vice versa in case of redoing moves.

2.5 Checking wins and ties

Now, our history holds every mark and every position of our turns, which poses the question whether there is a point in having a board in the first place. The answer for this would be yes and no. We could very easily render our board by traversing the history, however, an ordered board becomes necessary if we want to check for wins in a sensible manner.

We do this in our check_game_state() function, which changes the game state as it traverses through our board. Again, for this algorithm several approaches have been considered. It would be difficult to use a reliable heuristic to check whether the gameboard contains the necessary consecutive marks required for a win without the need to traverse each and every position. If our gameboard was a set size with the winscore of the same size, we could simply check the total of each row, column, and the 2 diagonals. However, to adapt to the changing variables of boardsize and winscore we had to approach the problem in a different manner.

The whole algorithm is broken down into three sub-algorithms, each of which traverses the board in a different direction: horizontal, vertical, and the diagonals (left-to-right and right-to-left). In each of these sub-algorithms we simply keep track of a "consecutive" variable which increases when two consecutive marks of same value are found and resets when the values differ. The magic of these functions is a simple return statement, which halts the execution when a win has been detected.
The algorithm is not the most effective, however, a few adjustments such as the cut-off return statement have been added to optimize it. One of these adjustments was added in the diagonals sub-algorithm. For this, the winscore is considered when traversing the board, and the corners which cannot possibly hold the required number of marks are not even checked - speeding up the process (see figure 4).

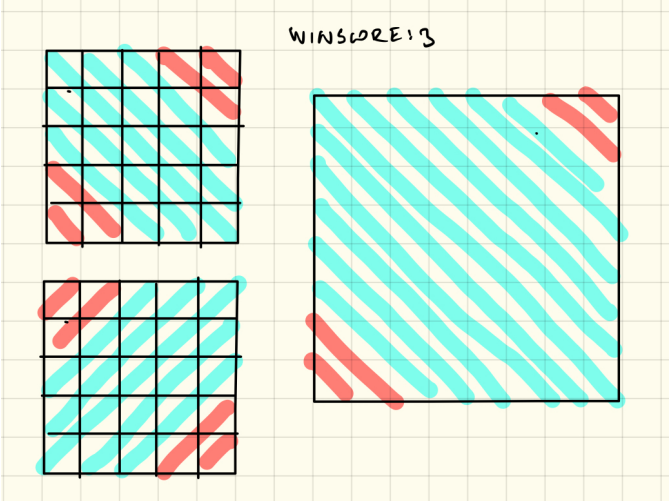

Figure 4: Red lines during diagonal traversal of the board are not checked as they cannot contain a win (winscore set to 3)

2.6 Computer player

To introduce a computer player, a few changes were needed. Firstly, along with indicating whether the player is automated, we need two more attributes stored in the player struct - the row and column of the best move.
Secondly, an algorithm to evaluate how good moves are was needed. One option would be to traverse our board and evaluate each free slot and assign a weight to it. This is exactly what we have done, however, to avoid too much traversing we have extended the functionality of the check_game_state function to behave differently when the current player is automated. This way, on top of checking for a win, we also compare our slots to a set of simple patterns that have been identified.
To explain these patterns, let us look at what is important when actually making a move. Let's say we are playing the classic 3x3 board (with the winscore of 3). When looking at a line (either row, column, or diagonal) we want all slots to be either empty, or to contain our mark. We also need to consider the opponents marks though, as we need to prevent them from winning. Therefore, the empty slots in a line will have the weight of

$$(number\_of\_marks\_in\_the\_line * 2) + 1$$

if the marks are all ours, or

$$number\_of\_marks\_in\_the\_line * 2$$

if marks are opponent's only, or

$$1$$

if all slots are empty, or

$$0$$

if line contains both players' marks.

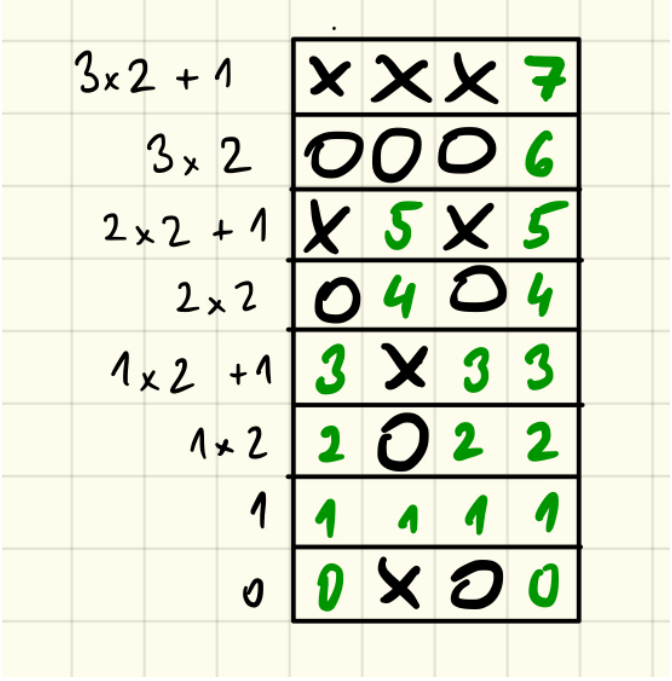

Figure 5: Possible patterns and calculated weights for a scope of size 4

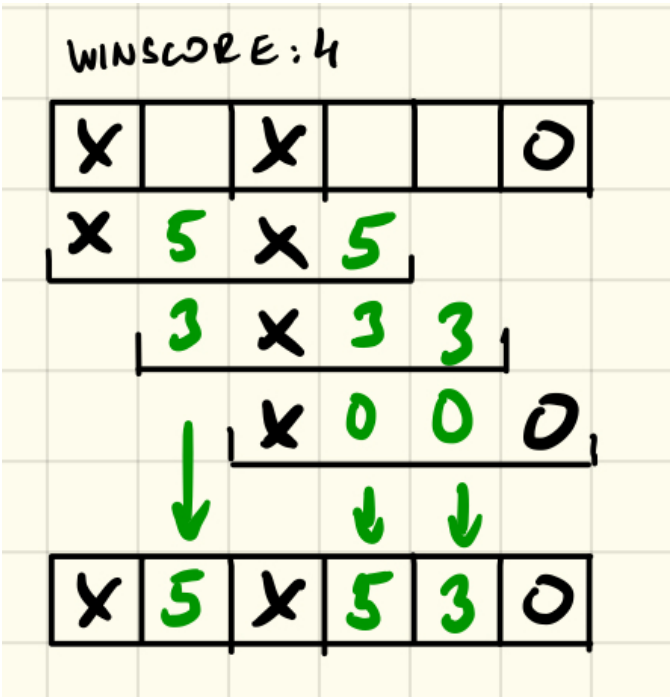

Figure 6: Scope implemented as a deque traverses through a line of the board. The size of the scope is identical to the winning score, 4 in this case. Weights of the board are assigned based on our patterns and the max values are kept.

To see a chart that better explains this logic see figure 5. The $+1$ in our set of rules makes sure this "heuristic" focuses on winning more than preventing losing (ie. when both players are one step away from winning, it is more important to choose a winning move than preventing the opponent's victory).

It is important however, that only higher weights overwrite the lower ones and not vice versa. This overwriting rule comes handy for boards where the boardsize is bigger than the winscore. In this case, we create a simple deque (the scope) of size winscore, where values are entered at the top and the bottom value is popped. To see an example of how this works, see figure 6.
We implemented the scope as an array of fixed size, as there will be no need for a size change. Also, this array is traversed several times when traversing the board itself, it is therefore reasonable to make use of the speed of direct access.

2.7 Menu and Old Games

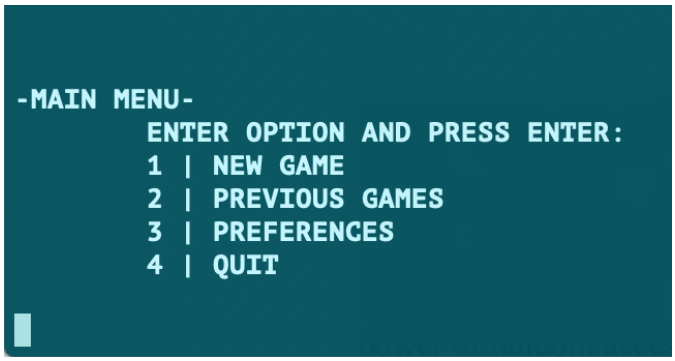

Figure 7: The main menu of the program giving users 4 main choices

A few more features have been implemented, such as the main menu itself (figure 7, for which the game loop has been wrapped within another while loop. This loop relies on a sessionstate variable, which based on what the "current screen" is changes between constants MENU, GAME, OLD_GAMES, PREFERENCES, and EXIT.

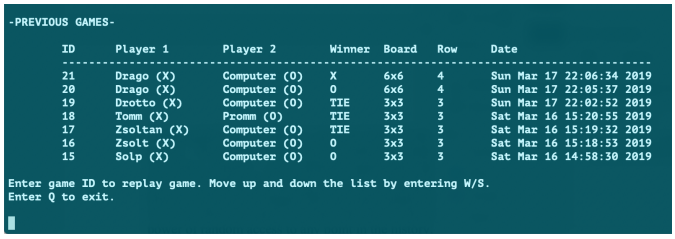

Figure 8: The previous games view displaying part of the list of old games

One of the last implemented features was the browsing and replaying of old games. Every finished game is saved into a text file as comma separated values and retrieved from this file when browsing through old games. A game in the "previous games page" (figure 8 can be selected based on its ID and navigated through from the first turn up to the winning turn (see figure 9). When rendering a specific state of the gameboard, a new temporary gameboard is created each time and the loop of rebuilding the turns stops at the specified index. As previous games are only replayed as a sort of a movie, and not actually played, there was no reason to reconstruct the history. This way it has become a very procedural task of drawing a board for each state.

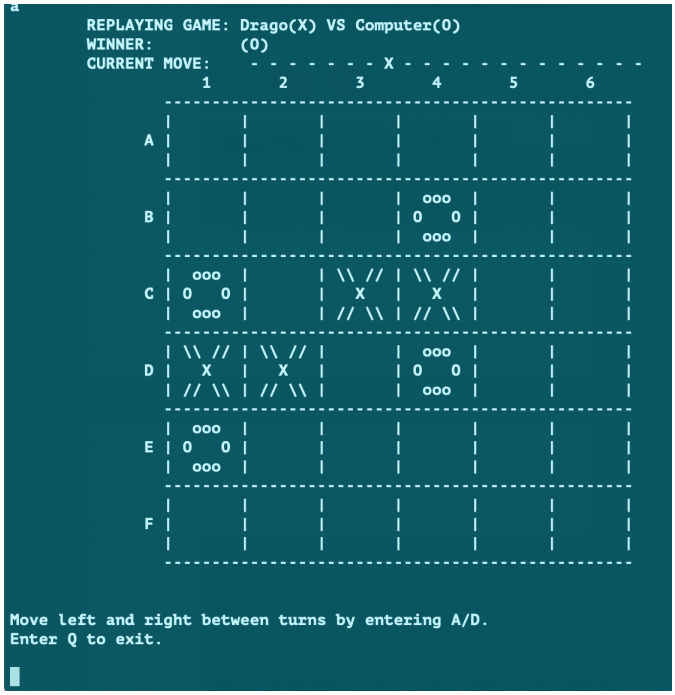

Figure 9: Navigating through a saved game. Notice current move at top indicating which state of the game-board is currently displayed

Surprisingly, the most challenging part of this functionality was the reading of saved files. Even though games are saved in a very specific format, the number of values saved for each turn varies based on the number of turns taken.
For this reason, we first need to build a matrix of sizes for the file, which is a dynamically allocated 2D array. Where the rows represent the number of games and lines represent the number of values saved for each game. Only then we can access our file and safely traverse through it knowing the dimensions.

3 Enhancements

The program could be definitely enhanced by the addition of other features, none of which however are crucial for the smooth run of the game.

Similar to the list of old games, a leaderboard could be introduced. Players already have a way of tracking how many times they won during the game. This could be saved to the external file and used to evaluate the ranking of players. The only challenging part about this would be figuring out a way to retrieve a user. If this is done solely on the basis of name, no two players could have the same name and perhaps a password would also be required to prevent users from using the wrong "account".

Games are already getting saved, however it might also be useful for an unfinished game to save, so that our program

can be closed and the game reconstructed. This functionality seems rather trivial considering the scale of the game, however some might appreciate the ability to keep a long run of the same 2 players. The logic for this is already partly implemented in the saving of finished game.

Most importantly there could be giant improvements made to the computer player. The current way our algorithm approaches the game is estimating which moves are the closest to winning/preventing loss, but at the same time acting randomly on the best moves. The problem with it is not choosing illogical moves but choosing moves without a strategy. Perhaps having the current way of weighing each slot in the board combined with an algorithm such as minimax could provide a good way of approaching this problem and create a player that seems more aggressive as opposed to careful.

4 Critical Evaluation

In overall, the software is rather performant at the current scale. The algorithms combined with the chosen data structures work surprisingly fast, however, there is always space for improvement. As can be expected, our biggest issues are how algorithms scale. More specifically, how traversing our gameboard becomes an issue when checking for wins in a game and weighing possible moves.
To check for game wins, we have to traverse our board in each direction - horizontally, vertically, and diagonally. As mentioned before, it was difficult to come up with a heuristic that would eliminate the need to check every single slot when recognizing a win, as consecutive boxes need to be compared to determine a win.
However, as a means to lift at least some computational load, we exit the loop right after a win is identified but also, when checking for diagonals, it is unreasonable to traverse lines shorter than a winscore (refer to figure 4). For this reason, we skip these lines altogether.

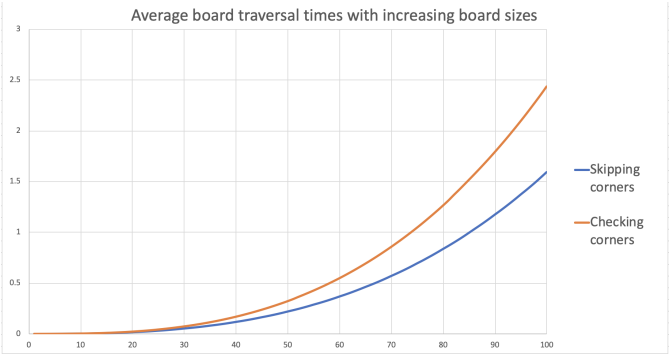

Figure 10: Average traversal times for the diagonal board check with and without unnecessary corners

In figure 10 we can see a comparison of average board traversal times (all worst case where end of array is reached). These times are for increasing board sizes from 1x1 to 100x100. It is also important to mention the winscore has been set to 3/5 of each board size.

The difference between these results is clearly visible. Unfortunately, both rise at an exponential rate O(nx). The limit set for our boardsize within the game has been set to 10, as it is very obvious from our chart the trend starts picking up drastically after this value.

The checking and weighing of good moves heavily rely on this algorithm as well, therefore, it would be very difficult to come up with a computer player implementing any kind of strategy with a bigger board while maintaining reasonable sizes. Therefore, it would be crucial to find a good heuristic for traversing the board.

On the other hand, one of the elements that has been well implemented for the computer player was the scope. The scope is used to keep in memory the last n elements during board traversal (n = winscore). As mentioned before, a fixed size array has been used for this as the winscore variable cannot possibly change during a game. This way, the time it takes to traverse a line of increasing length is a linear function, as opposed to an exponential rise in case of using a linked list. This can be seen in figure 11.

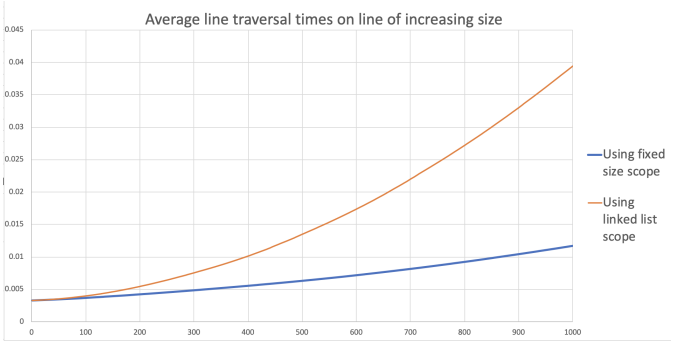

Figure 11: Line traversal times of scope of constant size on a line of increasing size. Time difference between scope implemented as fixed array and linked list.

The rest of the features in our program do not seem like a concern performance wise, except for, perhaps, the way csv game histories are read and converted. As we have mentioned before, since each line has a different length/number of values, we need to first build a size matrix for the file itself. As this function was written before implementing the game history functionality itself, the aim was to write an algorithm that can convert any length file with any type of variables. This is a good example of how planning and customizing code to one's need would simplify the task. If we were to set a maximum number of values in a line, we could set the unused values to 0 and prevent having to create 3D arrays of int's in order to read a simple text file. This, however, would only be possible to if we know the maximum board size and therefore number of turns possible in a game. Which, on the other hand limits us from changing the maximum board size in the future.

Aspects of the project could always be improved, the choice of used data structures could be changed, but in overall, the program seems to work well with the current choices made.

5 Personal evaluation

This project has been somewhat of a challenge, especially considering this being the first time working with C. There were several takeaways and lessons to be learned, as well as challenges that were overcome.

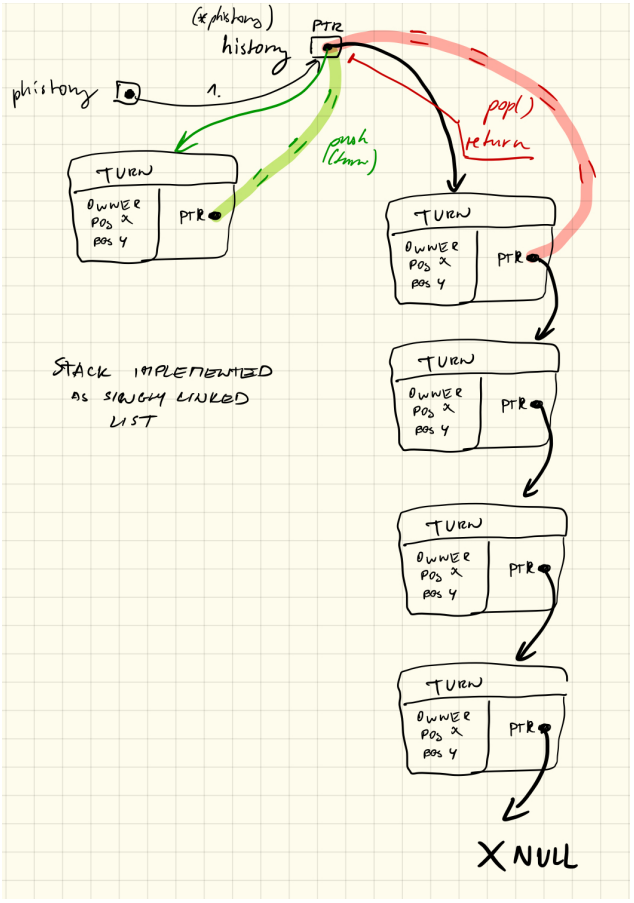

Figure 12: Notes on the approach for a stack using a linked list for the implementation of history and undone history

Firstly, as for any newcomer in C, the pointers and memory allocations were one of the biggest obstacles at the start. By utilizing pointers, a completely new approach to how variables can be handled is unlocked. This can be a very powerful feature but at times also confusing, especially when combined with memory allocation, multi-level pointers, and scope.
Even after getting used to new efficient ways of writing functions and utilizing pointers, it was a challenge of its own to keep track of which parts of allocated memory needed freeing and which ones need to persist for the whole run of the program. A good rule of thumb is to have the same amount of free() and malloc() calls. This advice, however, becomes difficult to follow when memory is allocated in loops and appended to other structures - such as the pointers in a history stack. Even in this case, the memory should not be freed even after undoing a move, since we still might want to redo it.
A good reference in understanding the logic and beauty of pointers was an approach introduced by David Brailsford as part of the Computerphile series, as well as the Udemy course C programming for Beginners by Huw Collingbourne. The former resource was also a useful graphical explanation of different approaches to implement singly linked lists 12.
The greatest challenge, however, was figuring out a reasonable way to estimate the next best move for the automated player (see figure 13 for the brainstorm). Even though the current approach does not rely on any strategy, it works well and is surprisingly difficult to beat on the standard 3x3 board.

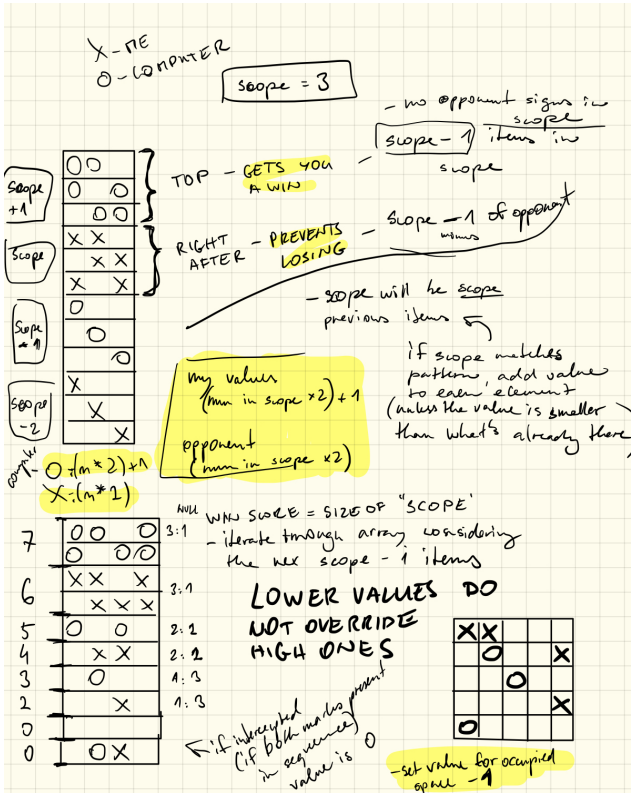

Figure 13: Notes on different approaches of implementation of a computer player

During development a sort of systematic approach/workflow was adapted. For every new feature/ data structure, the data structure has been implemented as an API in a separate project file to avoid cluttering and confusion. Once it was tested and working, it was integrated into the main project. This has been repeated several times.
The implementation of some of the features worked fine, however, to be able to build on top of these, they had to be changed several times throughout the project. An example of this would be going from a static to a dynamic array when representing the board itself or replacing an integer representation of the player with a struct.

The project definitely has been challenging but fun to work on and a great learning opportunity. The finished piece is a fully functional piece of rather well-written code, where data structures and algorithms used were kept in mind during development. There is certainly space for enhancements as well as improvements. For instance, better planning when it comes to the overall architecture in the first place could have eliminated the need for several changes in the way our for loops handle session and game variable. In overall, however, the project has been a success and many valuable lessons were learnt during development.