Algorithms and Data Structures Report

Robbie McNeil
40296075@live.napier.ac.uk
Edinburgh Napier University - SET08122

Abstract

For this report I will be detailing how I implemented a noughts and crosses game in command line using C and the decisions I made during development of the project, including what data structures were used and why. I will discuss the design of the game, what features the game includes, what improvements could be made as well as an evaluation the final product as well as my own performance.

Keywords – Algorithms, Data structures, C, command line, noughts and crosses

1 Introduction

For this coursework I was tasked with creating a noughts and crosses/tic tac toe game using the C programming language, this meant that as a minimum the program would need to have: a visual game board where the game takes place, a representation of each player, a piece that represents each player - either an 'X' or an 'O', and positions on the game board for the pieces to be placed. On top of this the program would need to be able to search the game board and determine whether a player has won/drew or not. The program searches for a winner after every move (despite the fact that a winner cannot be determined until at least the 5th move), there are 3 outcomes from this, either a winner is found and the game ends, there are no more available moves leading to a draw or there is no current winner but there are available moves that can be made so the game continues. I also included an undo and a redo feature so that users can go back if they made a mistake or go forward a move if they accidentally went back too far. Because of this there is some validation that must take place, firstly I had to make sure that if a user had undone a move then the next user would not be able to use the redo feature, as a redo can only be done executed by the same player who used an undo (the redo must also be used within the same turn as a corresponding undo). I also had to make sure that if there were no moves to undo or redo then a player would not be able to use those functions, as this may result in some unwanted actions, errors or even crashes.

Due to the fact that the program was to be run on the command line I was limited in what I could do with the game both in terms of visuals and interactivity. Because there is no mouse input in the command line it wasn't as simple as letting the user choose which square they would like to choose simply by clicking on it, so to combat this each square of the game board was assigned a number from 1 to 9. The players can type the number of the square they wish to place their marker in, as long as that square isn't already occupied by a marker then the numeric value will turn either into an 'X' on an 'O' depending on which player made the move.

There was some validation that had to be included to stop users from a number of things including: choosing an occupied square, undoing a move before a move has been made, redoing a move before an undo has happened and redoing a move that was undone by the other player. Including these validation methods ensured that the games mechanics could not be abused and that the game was fair for both players.

2 Design

When beginning this project I had to think about the details of the game and how they relate to the project, for example, a standard noughts and crosses game consists of a 3 x 3 matrix, with each empty square being a possible place to put a marker. Therefore with this in mind we know that there can only be a maximum of 9 moves in a game, no matter the circumstances or the order of moves made.

When I started development the first thing I did was create a method that would draw a game board. As previously mentioned there were limitations in terms of visuals, I ended up using a series of printed dashes and vertical bars to represent the game board and each square had a character value in it. The character values (%c) are initially filled by the contents of an array that contains the numbers 1-9, however they are later replaced by the player's markers, this is why the board had to be populated with char values rather than int values. Although arrays aren't as powerful in C as they are in other programming languages, mainly due to the fact that arrays in C cannot change in size once they have been implemented, I saw it entirely appropriate to implement an array here as I already knew the definitive size I would need the array to be - the array would only need to store 9 characters. Because of some of the limitations that arrays in C have there is also some improved performance over other data structures/arrays in different languages, which is another reason I chose to use arrays here.

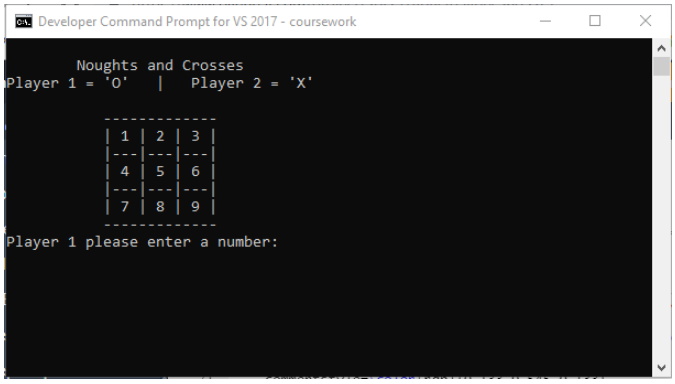

Figure 1: **Finished game board**- here you can see the board as it appears once the program has started

After this I focused on some smaller details, namely the players, moves, markers and the user input. The players, moves and markers are all represented by basic data types, player and moves are integers and the marker was a char. The player number was determined by using the modulus function. Each turn the player integer was increased by one and then divided by two, if the modulo operation returned 1 (aka player is an odd number) then it was player one's turn, otherwise it was player two's turn. The markers corresponded to each player, if player was equal to one then the mark would be 'O', if the player was equal to two then the mark would be 'X'. Finally the move was determined by user input, the user would have to input a number from 1 to 9, where each number corresponded to a space on the board. The program would then check if the input was first a valid number and then a valid move, if the move was valid then that player's marker replaces the number in the chosen square and the player count is increased.

Once my program was able to draw a game board, make moves and swap between players and markers I had to implement a search function that would allow the program to search the game board and check if there is a winner or not. There are 8 combinations of markers which lead to a win, initially I implemented a 3x8 matrix where each row was a winning combination. I wanted to write a function which would read each row of the matrix, check if those 3 spaces were occupied by the same marker, if so the game would stop as a player had won. Although I started to implement this it cause problems when I came to test the program, causing crashes upon running the program, so the matrix was removed. It it's place I added some if/else statements which in practice does the same job as the matrix, it checks the first combination and either returns a winner or moves on to the next combination. Although it took up more code and may not be as efficient or as graceful as the matrix method could have been the search method works. I would usually try to not have large chunks of if/else statements but because there was only 8 winning combination and 1 instance of a draw (when there is no winner but no more moves remaining) I wasn't too worried about implementing the search in this way. The search function returns a numeric value that corresponds to a state in the game with 0 meaning ongoing game, 1 meaning that the current player has won and 2 meaning that the game is a draw.

Finally I added the undo and redo feature after the 'base game' had been finished. Both the undo feature and the redo feature were implemented as stacks, although I had played around with the idea of using an associative data structure instead. The thinking behind this would be that there would be 9 elements in the data structure, each with a key (which specifies location in memory) and a value, the key's would go from 1-9 and correspond to a place on the game board. Then after a player had made their move the marker would be placed in the corresponding key's value. I decided against this method as it proved to be overly complex and because the structure is not sequential I would have had problems determining the order of moves. Stacks were a much easier and efficient data structure to use as stacks use the last-in-first-out structure. This means that the last item added is the first to be removed, which is exactly what is needed for an undo feature as it's the most recent move that that is to undo/redo. The undo stack contains a list of all move up until the current move, if a player wants to undo it takes the last move added (which is at the top of the stack) and pops it out of the undo stack. Subsequently the popped move is pushed onto the redo stack, however once the player makes a new move the redo stack is emptied, preventing players in the future from redoing moves which they have not undone themselves in that turn.

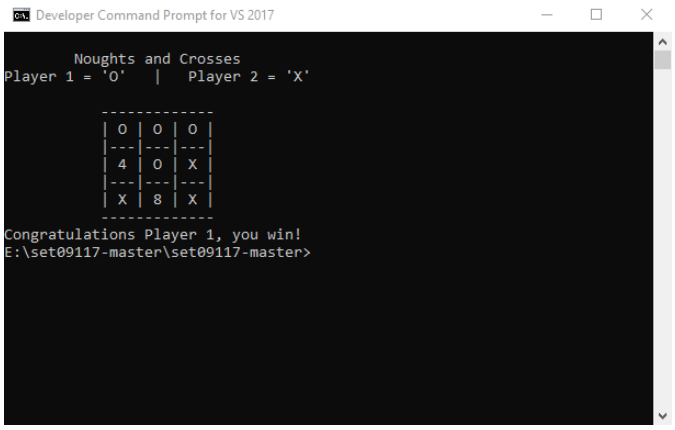

Figure 2: **A winning game**- this picture demonstrates that the program can detect a winner

3 Enhancements

There are a number on improvements that could be made to the program, both in term of features that could be added and improvements to the existing code. There were a number of additional tasks which I did not have the time to implement, such as the ability to record game history so that games could be 're-played' later. I also could have implemented a computer player that could play against a human player. It could be fairly easy to implement by having the program choose a number at random from a list of remaining spaces, however, programming the computer to be able to choose the best available move would be a harder task that required an AI algorithm such as the A* algorithm.

The two functions I would improve are the function which searches for a win and the function which determines which

move to make based on player input. These functions both work as they are intended to but they both consist of large amounts of if/else statements which take up a lot of code and aren't the most efficient way to do both these operations. I had tried to implement these operations differently, originally opting to use a for loop in order to tidy up the code however I had some troubles doing this as the user input is an integer, whereas the numbers on the game board are characters. Because I could not find a working solution to parsing the int into a char I changed the functions to if/else statements for the mean time and decided to come back and change them later if I had time.

As I mentioned the user input is an integer, which was a problem when it came to implementing the undo and redo as the 1-9 integers were used to determine moves the only remaining single digit integer was '0'. Initially I set the '0' input to open an undo/redo menu where a second user input would be taken and the user could type 'u' to undo and 'r' to redo. When I tested this in practice the message to choose either 'u' or 'r' would flash up on screen and before the user even had a chance to react and choose an option the program would return an 'invalid input' message. Although I tinkered around with this concept I could not find a solution as to why the user input wasn't working. As a short-term fix I assigned undo to the '0' input and redo to the '11' input. This is far from ideal but the makeshift solution works. I also attempted to change the user input to a char to match the game board numbers and allow the user to just type 'u' or 'r' without having to press 0 to open a menu, however this messed with other code which I had previously written and I quickly became confused when trying to update everything. Because parsing between fundamental data type in C isn't as straightforward as other programming languages I was limited to what I could do given the time I had, but this enhancement would definitely improve my program.

As well as the aforementioned enhancements there as some smaller things I would improve given the chance, such as performance. The game itself performs quite well however it can be slow to compile sometimes. I would also maybe work more on the presentation of the game, but as I have mentioned, there is only so much one can do with the command line interface. The program also crashes if the player enters an invalid input, although I included some validation for this it does not work as intended so some improved validation could be included if I had more time to test and research a solution.

4 Critical Evaluation

As with every program there are some aspects that work better than others and this program is no exception. For instance, no matter what action you take whether it is making a move, undoing or redoing the program will always keep tabs of which player's turn it is. This may seem like a simple addition as all it required was adding or subtracting the player value by 1 in certain situations, however it is an important aspect in keeping the game fair for both players.

Another aspect I was really pleased with was the redo feature. When I was in the middle of testing the game during the redo development I had discovered that either player could redo as long as there was some moves in the redo stack - even if these spaces had since been taken. I realized this was a problem so I developed an 'emptyStack' method which got rid of every element in the redo stack. Now every time a user makes a move the redo stack is emptied, meaning that you can only redo if you have used the undo in the same turn. When I tested this feature I was able to confirm that the solution had worked and the redo feature could no longer be abused.

There are some smaller details included which I think give the game some extra added charm. When the game is over a windows notification noise is played, letting the players know that the game is over (and also giving the winner some sense of achievement). This feature was implemented using the '\a' operator. Another small detail is the use of the system("cls") command which is used by the method that draws the game board. This command makes it so that the system clears any previous lines in the command line so that when you open the game it is the only thing on the screen, there is no previous work to distract players or get in the way. Although both these features are small they are a welcome addition to the program. I am pleased with the board I was able to create as well as how the game itself plays given the tools at my disposal. I also implemented a feature so that once somebody wins the game all markers are removed from the board accept the 3 winning markers. Although this doesn't do much for the game I thought it would be a nice touch.

I've already discussed some aspects which don't work as well as I'd like in the previous section but I will touch upon them here briefly. The user input has two main problems: validation and the unintuitive undo/redo input commands. As I've said the user input requires an integer, so if a char is entered instead then the program returns an 'invalid input' message. After this however if the user tries to pick another move, even if it is an int, they are met with 'invalid input' messages. Similarly this affects how users must undo and redo, with those commands being '0' and '11' respectively. Having the user type undo or redo would be a better design choice, although this is a smaller issue as it doesn't cause any crashes or errors. Other than this the game can take a couple seconds to run at times, which can be annoying, but once it is running there is no dip in performance.

5 Personal Evaluation

This assignment did put me somewhat outside of my comfort zone as I'm not very familiar with C although I have used it a few times in the past. As well as this I am used to using the Linux command prompt after using it in several other modules, but I am not as used to the windows command prompt. There are a number of similarities between the two and even some commands are the same so it wasn't too hard to adjust to, but I did find myself at times typing in incorrect prompts and having to look up the windows command prompt equivalent.

Because I didn't have a massive amount of knowledge about C before this module I found myself regularly going back to the lab slides to get information or code. For example I used the code from the second lab slides in order to implement my stack structure, however I ended up playing around with the code and adding in some of my own as I needed to have more than one stack. I also used the lab slides and the practicals I made from them in order to compare and decide between different structures. Having those small snippets of code to experiment with was also a big help in increasing my confidence with C.

Not everything I used in my program could be found in the lab slides however and I did have to use the internet in places in order to find solutions to some problems. Certain features such as the notification noise playing once a game had ended and using the 'system()' function to execute prompts from within the program were both found online. As I am not overly familiar with C I often made mistakes such as forgetting pointers here and there, when this would happen I usually looked up individual error codes on Microsoft docs in order to find out why those errors were happening and hopefully find a solution to the problem.

I also tested as I developed, playing and testing the game after every new feature added. Because of this I was able to find some bugs and come up with some fixes. For example during testing a game I found that I had an incorrect value in the search function which meant that getting a row of 3-6-9 would not result in a win. Developing and testing in parallel is a method which is often used in agile development cycles and is extremely useful not only for finding bugs but also finding where bugs are.

With that being said, my program isn't completely polished, I've already talked about enhancements, certain errors and missing (optional) features, and unfortunately I couldn't find every solution to every problem. Despite the fact I may not have had enough time to do everything I would have liked I think I managed my time well. One thing I didn't keep on top of was regularly committing work to github, previously while working on the command line I've had a repository on my local machine, whereas this time I only updated the repository by uploading the .c file to github through the browser. This is definitely not as efficient and next time I work on a project I'll be sure to take the time out of the beginning of the project to set up a local repository that I can push to from the command line.

Overall I am happy with my finished program and my performance, especially since I don't have a huge background using c. I managed to complete all of the tasks and some of the optional tasks with no major problems and the program works correctly when used as intended. despite this there are a few outlying problems that I would rather have fixed. If I were to do this project again there wouldn't be a lot I would do differently except for making the user input a char rather than an int, as this is where most of the problems lie. Otherwise I think that I used the tools and resources at my disposal well in order to both help create the project and fix any errors thrown my way.