

Algorithms and Data Structures Coursework

Christopher Jones

40274924@napier.ac.uk

Edinburgh Napier University - Algorithms and Data Structures (SET08122)

Knots and Crosses Coursework

1 Introduction

In this coursework for Algorithms and Data Structures, we were tasked with creating a noughts and crosses based game. The game was to have extra features such as the ability to undo and redo moves made by the players using a circular linked list to store the values that have been played. To be able to save games and then read them back in and be allowed to continue to play.

It also has the ability to replay a game back that has been played viewing every move. It has a 2.5 second delay so the player can actually understand and see what moves they made.

The game also has an AI bot coded into which can be used instead of player if the player wants to play by themselves. There is a main menu which allows the user to pick from 3 options, which are player vs player, player vs computer and, the rules for the game.

2 Design

I designed my software have a main menu so that the user could choose what they wish to start off with. They are given 3 options which are player vs player, player vs CPU or the game rules. The program waits for the user to pick one of those options and then does whatever that option is. This can be seen in the picture below.

```
Windows OS detected
Welcome to Noughts and Crosses Game :D
Made by Chris Jones Pog
-----
Please choose an option

1. Player vs Player
2. Player vs CPU
3. Rules and how to play
```

I used an array to store the game board values. These are the values that are used to draw the board for the players to play on. It is an array of chars because it meant I could use "strcmp" for key words such as "read", "save" etc and it also is very easy to populate an array with values. The

game board starts at element 0 rather than 1, which means the value in element 0 is the number 1 in char form.

```
char gameBoard[9] = {'1','2','3','4','5','6','7','8','9'};
```

This can be seen from the above picture and means the game board will need to always have its loop - 1 to compensate for 1 != 0. The game itself uses a while loop that will only break if the overall game counter is above 11 (Note: the game counter starts a 1). In this loop the game "scanf" which looks for a user input. If the user input matches any of the key words or is an area on the game board then the respective action will take place. If it is an area on the game board then that number that the user typed in will be turned into an X or O depending on who's turn it is. The turns are decided by a counter which checks if it can be divided by 2 with no remainder then it is X's go and if it does have a remainder then it is O's go.

To allow the user to replay the game after every go that has been played the value that the user chose is added to a circular linked list. I chose a circular linked list because it means there is no NULL value and the last value just point back to the start value. This means it won't crash because if a loop goes outside of its linked lists index. The values are added from the front rather than the back for this. This means the first value added by the user will be the the last value stored in the linked list. If a player undos or redos then the turn they undid and or redid is added to a another circular linked list.

```
struct Node *turnUndo = NULL;
struct Node *turnRedo = NULL;
```

The above pictures shows that I used another 2 linked lists to store the turn that an undo or redo has been done. This is so when the game is replayed the replay can say that the play to the players when they have did an undo and redo and when it happened in a message.

```
2
2
n 2: O undid and redid their last go in square 2
```

So as you can see the game can use those linked lists to display a message to the user.

If the player chooses to replay the game then that linked list is reversed this is so the first move is now at the top, this is the same for the undo and redo lists. It then loops around the circular linked list and compares the values of undo and redo against each other. If they are the same value then the player undid and redid but if they aren't the same then the player just undid. The game has a 2.5 second timer for each

of the game boards to be drawn plus a message saying what happened on each go. This is so the user can actually understand what is going on screen. Once the replay is complete the user can either restart or quit the game.

Writing to a file was pretty easy to implement, it was the exact same as writing a normal value expect you pass in a linked list instead and loop through the values. It writes whatever is currently in the game board array but, if the player has undone a go then they the writing data linked list gets a 0 added to represent an undo. This is so when the pointer meets a 0 it knows an undo as been done.

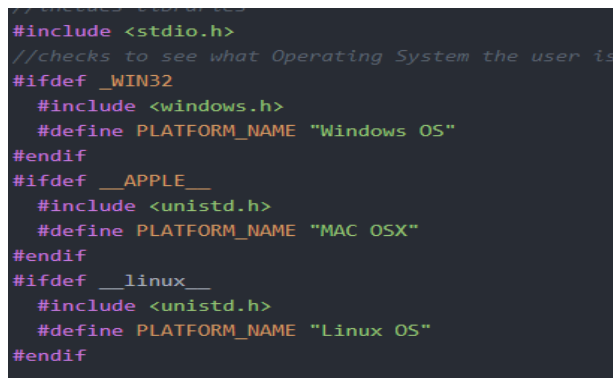
Reading from that was again same as reading from a normal file expect rather than putting it into a normal variable it goes into a linked list of read data. Once the game has been read in it can be played like a normal game and again once a player wins then they are able to replay the game. However, it works slightly differently from a normal game in the back end. Instead of adding values that the user has entered after they have been read in to the start of the linked list. They are added to back, this was so it was easy to understand the difference between a read in value and value the user entered.

To replay the game using read in data is also slightly different. If the pointers data is a 0 then the program checks if the previous value is the same as the value after 0 and if so moves the pointer forward twice. This means the player undid and then redid their go however, if they are different this means the player undid but then chose another value on the game board. This can be seen in the below code snippet.

```
1 if(last->data == 0){
2     if(last->next->data == prev){
3         last = last->next->next;
4     }
5 }
```

(Note: last is just the name of the linked list)

The program will also automatically detect what operating system you are the code on. This is because the Sleep() function is different depending on OS and therefore needs to be checked so that if the code is compiled on a MAC OSX or Linux machine then it will compile without errors. This can be seen below in the included picture.



```
#include <stdio.h>
//checks to see what Operating System the user is
#ifdef _WIN32
#include <windows.h>
#define PLATFORM_NAME "Windows OS"
#endif
#ifdef __APPLE__
#include <unistd.h>
#define PLATFORM_NAME "MAC OSX"
#endif
#ifdef __linux__
#include <unistd.h>
#define PLATFORM_NAME "Linux OS"
#endif
```

The final part of the design of the game was the AI bot that the user can play against. It is quite simple in the way it works in terms of its back end. It was intended for an alternative for the user to play the game if they didnt have another player.

```
1 char aiBot(int n, int num_shuffles){
2     char temp[9];
3     for (int i = 0; i < 9; i++) {
4         temp[i] = gameBoard[i];
5     }
6     srand((unsigned)time(NULL));
7     for (int j = 0; j < num_shuffles; j++) {
8         for (int i = 0; i < n - 1; i++) {
9             size_t j = i + rand() / (RAND_MAX / (n - i) + 1);
10            char t = temp[j];
11            temp[j] = temp[i];
12            temp[i] = t;
13        }
14    }
```

As you can see from the above code snippet, the bot simply picks a character from a temporary array of shuffled characters. If that character is an X or O then we just rerun the method with a new amount of n shuffles. Then the value is returned to the program in a char form so it can be used.

3 Enhancements

I think a feature that I really would have liked to have made better is the bot. This is because of the way it works by just choosing the first element in a shuffled array of chars. I did this because it was quite an easy way to implement a bot but, it means it has no way of choosing a value based off how good a move it actually is. It was more intended as a gimmick type thing than anything else but, if I had a bit more time I would have liked to have used a minimax algorithm which is a recursive algorithm for choosing the next move in an n-player game. This would have been perfect for a game like noughts and crosses because it is used used for a two-player game and would have given the bot a bit of logic and understand of the game. It would have allowed the bot to make intelligent moves and may have actually made it win more games.

Another enhancement I would have liked to have made would have been to have given the bot option all of the "undo", "redo", "save" and, "read" functions however, this caused some problems with the way the bot worked. For example: if you try to undo you don't get a chance because the bot makes its go within 1 sec. This means you are actually undo the bot and that causes a crash to occur. This is the same with redoing moves. I could allow the game to be saved with no problem however, you wouldn't be able to read it in so there wasn't much point. The replay function does however, still work because it just takes in the last number to played on the board.

Another enhancement I would have liked to have made would have been to how undos and redos are checked in the main game. I would have liked to have made them like the read in where a 0 is added as a sort of flag however, since I had already implemented this way I didn't want to break everything so decided not to change it. This would have cleaned up some of my code because there is quite a lot of different linked lists being initialised and added to at different points throughout the program. This would have made it easier to read and reduced clutter to make more understandable if I was to ever try and edit this program with the enhancement.

Another enhancement would just have to be to the overall code in general. This is probably more efficient ways to do things than the way I have tried to do it. For instance my replay game function basically checks for every possible way to play the game. By this I mean it checks if the game contains undos and redos by checking if the passed in linked lists are empty or not. If they contain a value then one of them has been used. Then a flag is checked to play the replay game differently but, it makes the overall function very chaotic and hard to tell if you're in the main() program on the replay. It also makes the length of the code incredibly long, so trying to find a certain part can be quite challenging.

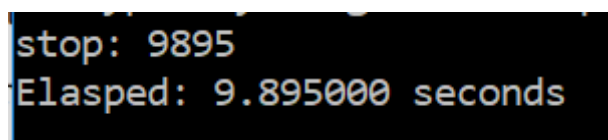
Another enhancement I could have made to the program would have been the ability to increase the game board size from 3x3 to like 6x6 and or 9x9. This would mean the game would go on for longer and perhaps be more enjoyable because playing 3x3 against yourself is tedious. It would also mean I would have also had to have increased the winCheck() method to include a win on a bigger game board. There would have to be a bigger game board array to store all of those values of that board or I could just populate the array with values depending on the size rather than just pre-defining the array with values.

I think my final enhancement I would have like to have added would have been to allow the game to have multiple games saved. At the moment this is not possible and the previous game is just overwritten if you save your current game. I did this for simplicity reasons as it made it easy to tell where one game had started and ended. If I had more time I would have looked into how to read line by line which could store different games as currently it writes down the file rather than along making it almost impossible to tell when the game has been won.

4 Critical Evaluation

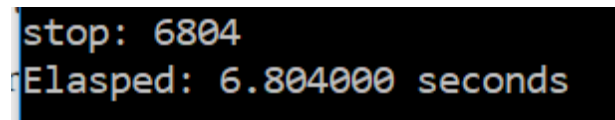
I think a feature that worked really well was the way I implemented my circular linked list for storing the areas the players had played in. This was because I added in from the beginning of the linked list for each new value being added. This meant that the first value that was entered was the last one to be stored inside the list which came quite useful for the doing the undos and redos. This was due to the way I had implemented those features, when I added them to another list the first value was now the last and vice versa which meant I could just loop through with the game counter to get the second to last value for an undo and the last value for a redo. It also allowed me to save on coding space because I could just use one method for undoing and redoing.

Another feature that I think worked well was the way I have implemented the game in terms of experimental results. After running the timer I found that an average game with players taking 2 seconds to take their turn took about 10 seconds with both players making optimal moves.



```
stop: 9895
Elapsed: 9.895000 seconds
```

From the above picture we can see the experimental results for the most optimal with a player winning in 3 moves. Which overall means the game play of the game is smooth and can be over quite quickly so players can replay or restart the game to play and such. I think this is a pretty reasonable amount of time however, I can make it quicker if I play against the bot because it only has a 1 second delay between plays.



```
stop: 6804
Elapsed: 6.804000 seconds
```

From the above picture we can see that with the bot the time is 33 percent quicker than it is with 2 players making optimal moves. This could have been made faster again if I had implemented a bot vs bot features both with 1 second delays however, since they randomly pick they wouldn't be making optimal plays and therefore could be slower.

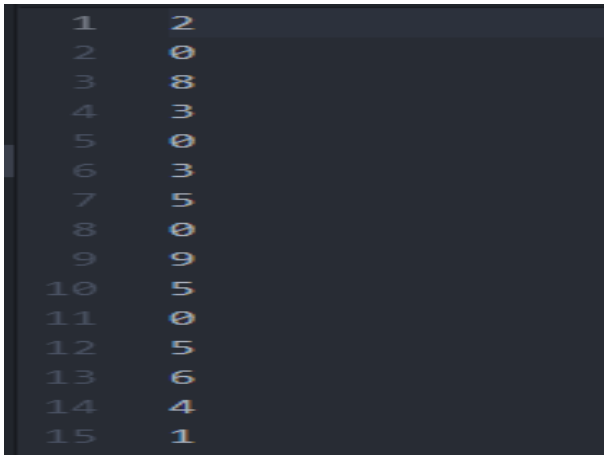
I think a feature that worked quite poorly was trying to replay the game after it had been read in from a file. This was because there is so many factors to take into account when a game has been read in. For instance the game has to read all of the data from the file back into a different linked list than normal. This was because the other linked list that would normally be used worked slightly differently and therefore wouldn't work for a game that was read in. This was a problem because it meant I had to initialise yet another linked list to store data. This meant I had one for a normal game, 2 for undo and redo, 1 for writing to a file and this one for reading from a file. This all adds up in the amount of RAM being required to store all of this data. Another problem that occurred by doing it this way was the fact that if you were to continue to keep playing you had to add to the end of the linked list rather than the start like I had being doing previously. This caused problems with the game was being replayed because it was in the wrong order so I had to write a new scenario for this type of replay. So overall it took a lot longer to actually create this section and took up more code because of how I had implemented it. If I were to do this coursework again I would find a much more efficient manner of doing this because this way was so bad to try to read and understand.

5 Personal Evaluation

In the coursework I have learned how to properly use a circular linked list and linked lists in general. Before working on this I sort of understood linked lists but, not to a point where I was fully comfortable using them. I actually managed to do the undo and redos using a 2-D array to started of with but, decided i wanted to challenge myself with something a little bit harder to implement as that was the point of this module. So I decided to create a copy of the game (hence why on GitHub the previous name had Copy next to it) and start the undo and redo again but, instead with a circular linked list. I chose the circular list of a normal because it meant there would be no NULL pointers as the end pointer just points to the start again.

A challenge I had to face in his coursework was how I was going to make the game save with its undos and redos in

a file. I first tried to delete the node if the player undid however, I found out that deleting a node caused the linked list to get random values from memory replacing the values that should be there. This was due to the fact that I didn't know about `memset()` which would have solved this problem if I hadn't already tried to do it another way. I ended up scrapping the idea to delete the node and instead add some sort of flag to check if an undo had been played. This flag had to be a value that was a number because my linked list only took ints and it had to be a single digit because I was reading in slightly stupidly as well. So I had only had one option which was 0, it looked like this



1	2
2	0
3	8
4	3
5	0
6	3
7	5
8	0
9	9
10	5
11	0
12	5
13	6
14	4
15	1

As you can see the file has some 0's written to it. They mean an undo has been done at that turn but if the value after the 0 is the same then it means they have redone or chose the same value as before. However, if it is different then it means they undid and then chose a different value. This meant the program could distinguish between when a normal move is made or an undo is made. This problem took me many hours to try and come up with a solution for as I didn't really know how to do it without physically deleting the node in the linked list and just re linking it together without out that node.

Another challenge that I had faced was how I was going to make this work on both Windows and MAC OSX because I wrote the code on a windows machine using the `windows.h` library for the sleep function. However, its a windows library which of course wouldn't run on my mac. So I needed a way of detecting which operating system is being used and then include the correct library and for mac its `unistd.h`. Once the OS has been established then I could use the correct `sleep()` function for each OS by a simple if statement saying if its windows do `sleep(milliseconds)` or UNIX `usleep(microseconds)` which is just divide by a 1000 to get it in milliseconds. This seemed like an easy thing but, trying to detect operating systems took quite a bit longer than I had anticipated because the if statements weren't working in the headers part of the program. I solved this by using `ifdef` statements rather than regular if statements which only worked for windows and not the other 2.

Overall in this coursework I think I have performed pretty well I have got most of the additional apart from the bigger game board and the ability to replay multiple older games rather than just the previous game. If I had more time and had focused on it a bit more then this could have been easily achieved I think. I overcame problems well and learned new

things about C because before this I never really liked C that much but, after this coursework I have started to like it just a bit more. So in the end I have created a pretty decent Noughts and Crosses game that contains different features for the players to use and utilise in C :D.

References

- [1] Stack OverFlow for General Stuff,
<https://stackoverflow.com/>
- [2] GeeksforGeeks for circular linked list,
<https://www.geeksforgeeks.org/>
- [3] w3schools some C stuff
<https://www.w3schools.com/>