ADS CW Report

Sonas MacRae
40277542@live.napier.ac.uk
Edinburgh Napier University - Module Title (SET08122)

1 Title

40277542 ADS CW report

2 Introduction

The purpose of this coursework is to create a tic-tac-toe game in the C programming language to test our understanding and knowledge of algorithms and data structures.

When the game is launched, the user is brought to the main menu, which is where all of the features are presented. These features are as follows:

- Play against the computer, a bot was implemented who plays against the user.
- Play against another player.
- Update the names of the players, this doesn't change the name of the bot.
- Replay previous games, after a game is completed, it is saved, at any time the user can replay any game, which includes commentary that briefly describes what happens at each stage in the game.
- Help, this prints the rules of the game as well as some other information which is useful, such as an introduction to the bot.
- Quit, exits the game.

On top of this, during a game a player can undo to a previous board state and redo to a board state which has been undone.

3 Design

As an overview, 2 linked lists are used to store the different board states throughout the game, with a pointer node that keeps track of the current board state. One singly linked list is used to store every board state in a game, and one doubly linked list stores the board states that are relevant, this list only contains the board states that are available to undo back to and redo forward to. Each move is saved to the linked lists before the game board is checked to see if a player has won the game. After each move, the board is updated and printed to the console.

3.1 Data structure design

The doubly linked list is used to store the board states, each node contains a one dimensional 9 element character array to store the state of the board, and previous and next pointers to the adjacent nodes. The reason why a doubly linked list was implemented was because of the efficiency it brings to the memory management side of the project. When a new node is added to the list, memory allocation (malloc) is called, this allocates the right amount of space on the stack since the size of the node is known, which prevents waste. Another method for storing the board states would be an array, this array could store the differences between the board states. With a clever enough design, each iteration of the board could be represented using a single character. This was considered, letters could also represent numbers, upper case letters could represent player 1 and lower case letters could represent player 2. The entire game could be stored as a string. Although updating a linked list is more manageable and scale-able than using a string, nodes can be added and removed from any index in the linked list easily.

One of the main reasons why a linked list was used instead of an array was because of its flexibility and ease of use. Since each node in a linked list can hold more than one variable, the way that the data is handles is very neat. It can be said that arrays are better since any index can be accessed without having to iterate through the array, which is how you access a certain element in a linked list. Although since the linked list in this coursework has a pointer node that keeps track of the current board state, as well as the fact that at no point will a random board state need to be read from the linked list, this makes using arrays more complicated. To add to this, since arrays are fixed size, if it needs to store more board states then functions will be required to copy the entire array, and paste it back into a new array of a bigger size. This brings a new problem, how many more indexes should be added, too many and they never get used, which is a waste of memory, too few and this function has to be called too frequently which isn't efficient with the run time. Thus the conclusion was the implement linked lists to store all of the data.

Another linked list was added to save every board state throughout the game, whereas doubly linked list is used to store the current board states that can be undone back to and redone forward to.

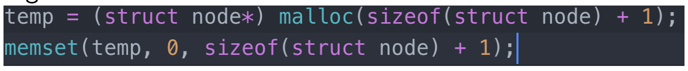
A string is used to represent the board, this string is stored within a pointer node, this node is a copy of the node that stores the current board state in the doubly linked list, when the board has to be displayed, the function responsible for this will call the pointer node and display its content. The pointer node also stores the next and previous nodes in the linked list, which is used for the undo and redo features.

The players have names that represent them, they are stored as global character arrays, the reason is so that they can be accessed by any function and can be updated easily.

3.2 Algorithm design

When a new move is made, the new board state is saved and added to both of the linked lists. First it is checked if the list is empty, if so the list is initialized with a blank board state before adding the new board to the list, this is the same for both of the linked lists. Memory is allocated using Malloc (shown in Figure 1.). Memset is then called to ensure the memory addresses are empty and can be used again if a new game is to be played.

Figure 1: Malloc and memset



```
temp = (struct node*) malloc(sizeof(struct node) + 1);
memset(temp, 0, sizeof(struct node) + 1);
```

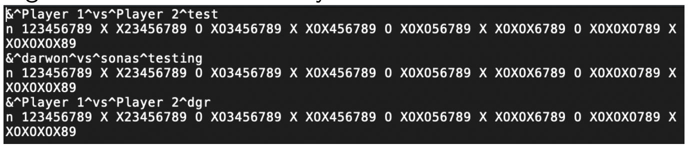
When an undo move is made, the pointer node, which stores the current board state is updated by shifting one index down the doubly linked list. When a redo move is made the pointer shifts one index up the doubly linked list. It is checked if there are nodes in the doubly linked lists after the index of the current board state (if a redo can be made) when a new move is made, if so all of these nodes are deleted to stop multiple time lines from being created from users performing undo's and making new moves, it would be too much work to keep track of all of the undo and redo structures that would be created.

After each move, a function is called which is responsible for printing the board state to the console. This function works by calling the pointer node, accessing the character array that stores the board state and printing its element in a 3x3 format.

After this, a function is called to check if the game is over, either a player has won or the game ends as a draw. This function loops through the board checking each horizontal, vertical and diagonal plane to see if either player has 3 of their symbols in a row. To make this function more efficient, a 3 iteration for-loop is called to take care of both the vertical and horizontal planes.

When a game is finished, all of the nodes in the history linked list are written to a text file, along with the names of the players and a custom game name chosen by the player. The data in the text file is split up using special characters and newline characters, this is shown in Figure 2 below. When a game is replayed, the text file is read from, each game is presented along with corresponding ascending numbers. Each board state throughout the game is displayed one by one, along with a line of text that briefly describes what happened on that turn.

Figure 2: results from doubly linked list



```
&^Player 1^vs^Player 2^test
n 123456789 X X23456789 O X03456789 X X0X456789 O X0X056789 X X0X0X6789 O X0X0X0789 X
X0X0X0X89
&^darwon^vs^sonas^testing
n 123456789 X X23456789 O X03456789 X X0X456789 O X0X056789 X X0X0X6789 O X0X0X0789 X
X0X0X0X89
&^Player 1^vs^Player 2^dgr
n 123456789 X X23456789 O X03456789 X X0X456789 O X0X056789 X X0X0X6789 O X0X0X0789 X
X0X0X0X89
```

After a game is finished, the linked lists have to be emptied to free up the memory that was allocated to them so that this memory can be used for the next game. This is done by traversing through both of the lists and calling 'free()' on each node. Since if this memory was never cleared the user could keep playing new games until there is no memory to be allocated for new games, which would make the game crash.

A bot was written and added to the game, this works by analyzing the board before making a move. It is represented using an integer function. The function starts with a 2d array, containing the horizontal, vertical and diagonal planes of the board. Each array of the 2d array is looked at, counting the number of blank spaces, number of X symbols and number of O symbols. From this a score is assigned to each array, with the bot being represented by the O symbols. With the highest possible score of an array coming where there are 2 O symbols and a blank, where the bot can win. If this isn't found then the next highest scoring array contains 2 X symbols and a blank, where the bot can counter the player from winning, this score system goes all the way down to zero with a blank array. Although this isn't perfect, it was simple to implement and worked as planned. The bot is unable to perform undo's or redo's as this would be too complicated to implement given the timescale of this project.

4 Enhancements

Given more time, I would have liked to add the feature to make the bot play against its self, all of the code needed to do this exists within the script, so it wouldn't be difficult to implement.

Another feature that would have been added was the ability to allow the user to delete all of the saved data in the output text file, as well as resetting the names of the players. The reason why this wasn't added was because this idea came about after the script was completed.

Ideally, the user interface would look cleaner, for this project I came up with a simple design and stuck to it. The focus was not about the user experience since I had focused on making the back end more efficient and to ensure that everything works as it should.

It would have been really cool if players could have user profiles, where each player would have their own text file, this file would store the number of games the player has made, counting the number of wins, draws and losses. It could go into further detail and show the win percentage against certain players.

5 Critical evaluation

5.1 What works well

All of the functions are useful, they have their own purpose and they are neatly laid out. Because of the use of Malloc, the exact amount of memory is allocated each time it is called, which prevents excess memory being allocated.

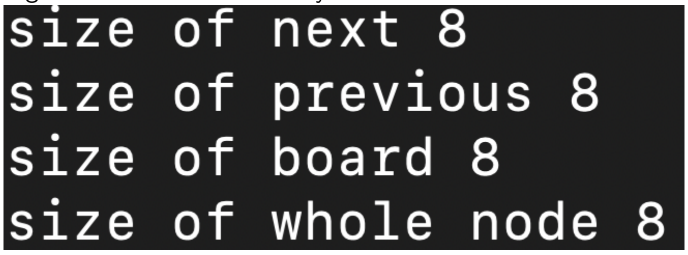
The pointer node, keeps track of the current board state, makes it easy to undo and redo and makes it easy to print the current board state using the render function

Some of the text is displayed one character at a time, this looks aesthetically pleasing, it is a popular method to print text from an NPC in video games so I adopted this idea and added it whenever the bot speaks and when the helper page is called

In terms of performance, the linked lists are memory efficient, I printed the size in bytes that each element of each node takes up, as well as the total size in bytes of the node, this is illustrated below.

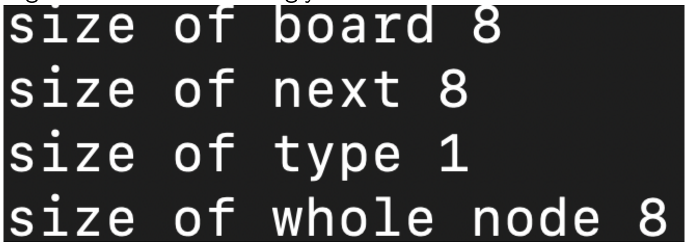
Although the was in which data is handled is very clean and scale-able, the fact that there are two linked lists seems clumsy. The history nodes could have been stored in a string, this would make the output file smaller, the information stored within the nodes

Figure 7: results from doubly linked list



```
size of next 8
size of previous 8
size of board 8
size of whole node 8
```

Figure 8: results from singly linked list



```
size of board 8
size of next 8
size of type 1
size of whole node 8
```

6 Personal evaluation

Many challenges were faced throughout the process of this project. For example understanding how memory allocation works in C, memory was being accessed after being freed up, to overcome this I had to set the memory locations to zero. This solution came about after a long debugging session where I had to go through the script manually debugging and printing to the console to see where the code jammed. The solution was finally found after reading many online articles about similar issues and learning about the subject.

Many bugs arose with the undo and redo features, for example the game would crash when the player would undo back to the original state, although the original state of the board would not display, as it turned out the function which saved each board state wasn't saving the blank board, stopping the user from starting the game over, the first move became permanent. To get over this, I added code which checked if the linked list is empty, if so create an empty board state and append it to the list before appending the new board state updated by the player.

I feel that I performed well on this coursework, I implemented my own design, managed all of the requirements and extra requirements as well as adding some features on top of this. I overcame every major problem that arose and learned a lot about memory management in C. My confidence with C has improved and it was interesting to strengthen my knowledge about the algorithms and data structures that I studied.