Algorithms and Data Structures Coursework Report

Lea Whitelaw
40279124@live.napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET08122)

1 Introduction

The objective of the coursework task is to create a text-based game of 'Tic Tac Toe' which demonstrates my understanding of algorithms and data structures relative to this module. The basic requirements of the game is to show the board, the players, the pieces and the positions and implement any other necessary features in order for the game to function. On top of these features, using knowledge of data structures I have implemented features which can undo, redo and replay the moves of the game. This report will detail the process which lead to the final outcome of the game that was submitted, the challenges that were faced during this process, and how the application could be improved in the future.

2 Design

2.1 Software plan

To begin tackling this coursework, I thought that the most efficient way to do so would be to write a brief plan of the functions and what they would do, and then build the code from there. A written list of these functions are shown in Figure 1. Having these functions written down helped me to write the plan for the loop to run the 'main' function which is shown in Figure 2. Once I had written these plans it was enough to begin writing the code. Obviously over time these functions evolved to fit the needs of the game and the extra functionality added.

Arrays The first data structure decisions that I made was the choice of a two dimensional array (matrix) instead of a one dimensional array for recording the board and the moves of the players. A two dimensional array is an array of arrays, so in this case would contain an array of three arrays which have three characters. There are advantages and disadvantages to both of these structures but due to the matrix type nature of the game board, I thought that a two dimensional array would be the most appropriate. To set this up I configured a two dimensional array named matrix which when set up contained the character ' ' of a space, then when the player entered co-ordinates, the space would be replaced with either an 'X' or an 'O' depending on the player. Having a 2D array made it very easy to iterate through and index the game board later on in the code. To get the game working, I wrote a function to get the move from each player and fill in the appropriate space in the matrix. After implementing this, it was easy enough to get the game functioning between two players.

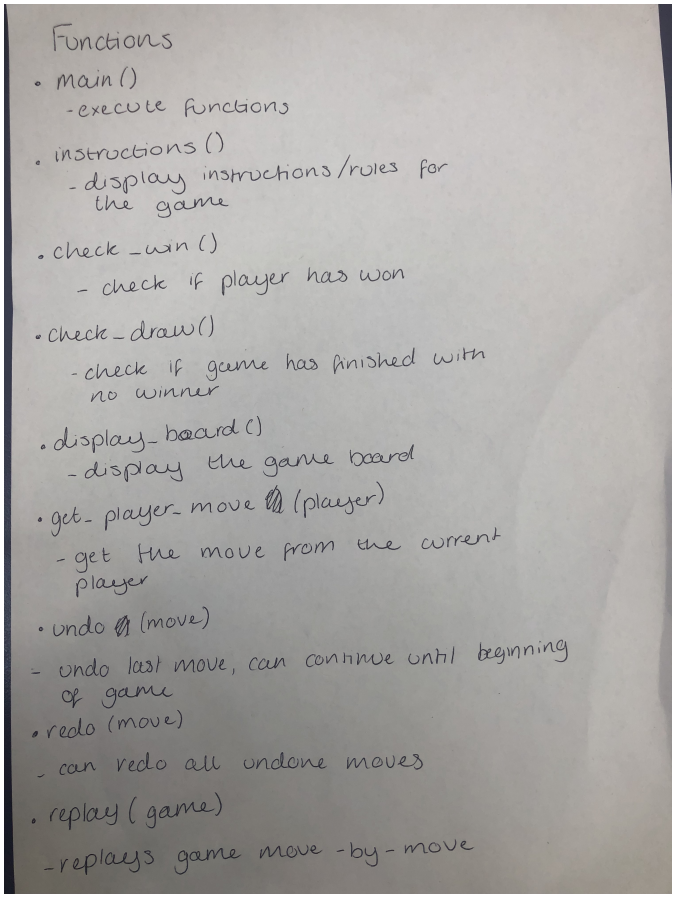

Figure 1: **Functions Plan** - Written plan which details functions for game.

2.2 Undo and Redo

As I was sure from the beginning of this assignment that I wanted to implement the undo and redo functions, I knew I needed to come up with a plan of the structure of these functions and how they would work. To decide which data structures to use, I considered the 'pros and cons' to each viable data structure in regards to the task at hand.

Doubly Linked List The first data structure that came to mind was a doubly linked list. This seemed like a viable option due to the fact that it stores the previous and next positions in the list which would be good for going backwards and forwards(undo and redo). As stated, the main advantage of a doubly linked list is that is can be traversed both forward and backward direction. Another advantage is that it can quickly insert a new node before a given node. This could be relevant to our game if we wanted to undo a move and then insert a new move in place of the one that had been undone. This

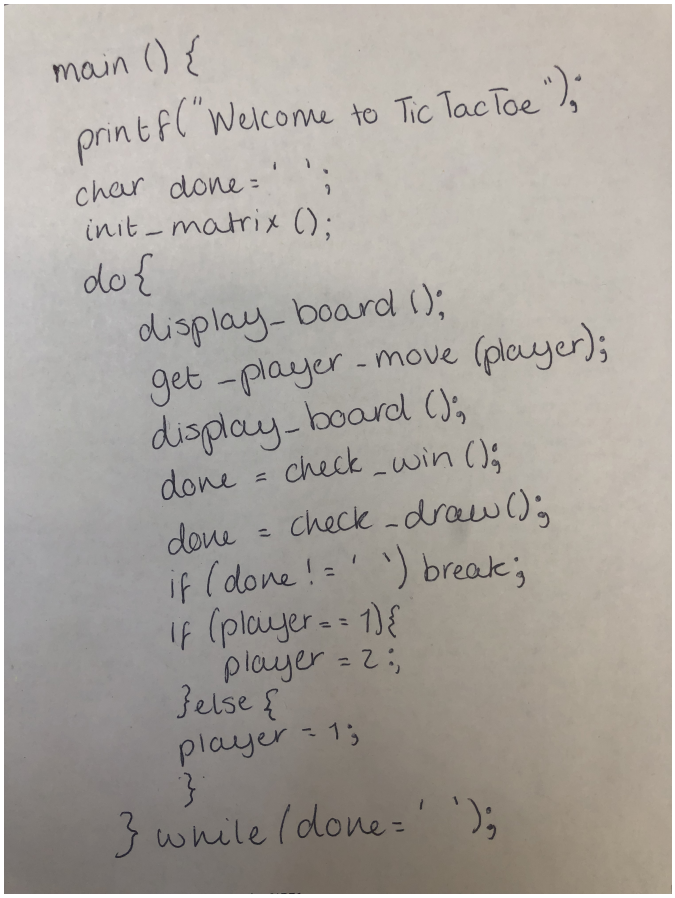

Figure 2: **Main() Plan** - Written plan which details functionality of 'Main'.

may get confusing though if it came to undoing lots of moves and then instead of redoing them, replacing them with new moves - this would definitely be doable but I wasn't sure if it was the most efficient way when considering the downside of linked lists which is that that all operations require the pointer to the previous node to be maintained at all times, this can make the functions longer and more complex.

Queues The next data structure that I considered was a queue. A queue is a FIFO (first in, first out) structure, one of its big advantages are that it has an infinite length compared to the use of structures like arrays. This is a big advantage in general for data structures but wasn't entirely necessary for this game as we know the amount of plays that can be made. A queue allows for addition from the end and deletion from the front. I thought that this might be useful for a 'replay' function but decided to keep looking for a solution for undo and redo.

Stacks The final data structure that caught my eye for these functions was a stack. The reason a stack appealed to me is because it is a FILO (first in, last out) structure and stacks are good for programs or functions when you need to reverse something- perfect for an undo function. Stacks are good for keeping track of the state of things, so would work well for a reverse function, but are terrible for indexing - but the needs of this function did not require indexing. To create my undo function, I decided, I would push each move to a stack and then when the user wanted to undo, I would pop the move from the stack. An important thing that struck me when constructing this plan was the question 'what would happen when the stack had 'popped' off the undo move and then the user wanted to redo it and it was gone?' This question led me to the plan which I constructed of having two stacks which bounce off each other for 'undo' and 'redo' functionality. The sketch I made, shown in Figure 3, shows the way in which this would work. The idea being that there is two stacks, and each time a move is made in the game, it is 'pushed' to stack 1. When the player decides to 'undo' a move, it will be 'popped' from stack one and pushed to stack 2 (whilst the space for the 'popped' move is cleared on the board.) This seemed like the most efficient way to approach these functions because this way, if the user does multiple 'undos' the when they decide to 'redo' a move, it would be the most recent move which had been added to stack 2 that would be recalled. Another important thing to note is that for the structure of my game due to my preference, I added a function that clears stack two any time a new 'move' is added to the first stack (one which wasn't a redo) so that the players cannot redo any more after resuming playing. To create these stacks, I created the following two structures of code:

Listing 1: Move Structure

```
1   struct move{
2   char x;
3   char y;
4   int player;
5   };
6
7   struct stack
8   {
9     struct move array[MAX];
10    int top;
11  };
12
```

using the 'move' structure, I created stacks of 'move' objects which contained an 'x' co-ordinate, a 'y' co-ordinate and an integer of which player made the move. For the stack I created the functions 'init stack' 'push' and 'pop' where init initialised the stack, push added 'move' objects to the top of the stack and 'pop' removed them (whilst returning the 'move' object.) To create these functions I used my knowledge of the stack structure and took inspiration from the labs that I had done earlier in the term.

2.3 Replays

Replays of previous games was another functionality that I wanted to have for my game. As mentioned earlier, when considering the 'queue' data structure for other functionalities, it became apparent that it might be suitable for a replay function. The reason that I believe it may be ideal for this is because of its 'FIFO' structure. This way, each move could be recorded and stored in a queue and then when it comes to replaying it, the first move in, would be the first move played back to the user, therefor the user could watch the replays of the moves from the start to the end of the game. To have this function working, I would need to store each game in it's own queue. To implement this function I created a queue of 'struct move's, an 'enqueue' and a 'dequeue' function. After this functionality had been added, it became possible to replay a game once it had ended. To do this I used a time delay in a for loop to show the different steps. I

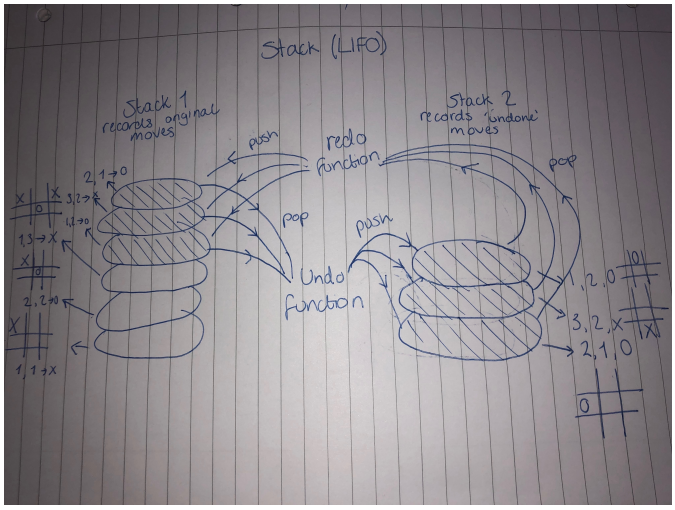

Figure 3: **Stacks Plan** - Visual plan for creating undo and redo using stacks.

used a basic function for the time delay which I found on an coding website. website[1]. Originally, I would have like to have the replays of the games moved over from memory to storage so that I user could view the past 5 or 10 games, but I did not have sufficient time left over to fully implement this after the rest of the functions had been created. The idea that I have is that the game could write output of all the 'moves' to a file and then when the game is re-opened and a player wants to watch the replay of the previous games the program could read the output file and display the moves in the same format that it is doing them now.

Implementation After all of these plans had been made, it was straightforward to implement most of the features of the game.

3 Enhancements

Although overall I am happy with the game and the way that it has been implemented, there are always ways to improve things, and there are a few things that I would work on for this game if I had more time.

Connecting to storage and replaying. As I mentioned before, the replay function that I created only replays the game that has just been played but does not commit them to storage. Ideally, I would have liked the game to have stored the last 10 games and be able to read and write to an output file which records the moves of each game, reads them into a queue and then replays them. I looked into some options for reading and writing to files in C[2], but did not have enough time to implement this function.

User Interface One improvement I would have made if I had more time would be to clean up the user interface slightly. Currently, as the user makes a move, they have to decide whether to press 'i' for information, 'r' for redo, 'u' for undo or any other character to continue to play. Realistically, this is not very user friendly, as the user has to stop after each play to tell the game to continue to play. If I had more time, I would have merged the two functions to get the players move and to get the players 'play/undo/redo' option to be one of the same function, however when I tried to do this last minute it came up with a few errors which made me realise that I would have to restructure a lot of the game to do this and I did not have enough time to ensure I could do that successfully before the deadline. So, for now, the user has to continue to go through two steps to play their turn.

Computer Functionality One of my original goals for this project, if there was enough time, was to implement a function where the user could play against the computer. In my initial design, I drafted a function, with inspiration taken from this site[3], where the user could play against the computer. The problem with my function was that the computer would just take the first empty space that it would find, so it wasn't really 'playing' the game. After creating this function, my goal was to get the computer to have different levels of difficulty - 3, to be specific - where the first one would be the function that I had already written, where the computer just picked the first available space it came across. The second one, where the computer would pick a space at random (this could be more difficult to play against) and the third one, where the computer actively tried to beat the human player. There would also be a possibility for the computer to play against itself. Due to the fact that I didn't get all of these functions working, I removed the computer player functionality altogether for the purpose of the hand in.
Although I am disappointed that I did not manage to implement these functions, it is important to note that they are beginning to leave the scope of this coursework, and becoming less related to data structures and algorithms (although I'm sure they would have been helpful to implement these functions.)

Bigger Game One possibility that I considered implementing was a game inside of a game. This would work such that to win a square, the players would have to play for each square and win it. For example, if players wanted to play for square 1,1 , they would enter a game of tic tac toe, and thee winner 'X' or 'O' of that game would then get to place their 'X' or 'O' in that square. This would have been a fun implementation if I had more time, and could have made more use of data structures that I hadn't yet implemented in the game I have currently created.

4 Critical Evaluation

To get a feel for what worked well in this game, I had to play around with it multiple times. Currently, the undo and redo functions are quite well structured and airtight, but at the beginning of their creation there was a lot of error handling that had to be done. It was a re-occurring theme throughout this process to get a 'segmentation error 11' when errors were not caught properly. To debug, I used print statements to try and find where the computer was executing the commands and where it was quitting. To stop these errors occurring when undo and redo were used one too many times (i.e. the stacks were empty) , I had write some code which returns the top 'move' in the stack that is not empty and returns the

computer back to the function to play. The code is shown below.

Listing 2: Pop structure (for undo and redo functions)

```
1  /*****pop(remove) from stack*****/
2  struct move *pop(struct stack *s, struct stack *s1, struct ↩
     stack *s2)
3  {
4    struct move *data;
5    if(s->top == -1)
6    {
7      printf("Cannot repeat this action any futher.\n");
8      if(s1->top == -1){
9            data = &s2->array[s2->top];
10           return data;
11       }else{
12           data = &s1->array[s1->top];
13           return data;
14       }
15   }
16   data = &s->array[s->top];
17   s->top--;
18   return data;
19 }
20
```

As is shown in this code, the function takes three stacks (two of them are the same stack), and when the one it is working on is empty, it checks which of the other two is not empty, and returns the last move whilst exiting this function.
There were many other errors that occurred and a lot of tweaking had to be done. To get feedback, I played the game with a few friends and some of my flatmates. Their feedback was mostly positive- especially once the undo and redo functions had been perfected - but one critical feedback that was given was that the user interface is a bit tedious with the continuous need to tell the game to play at each turn - something which I mentioned in enhancements that I would like to have changed. One other point to mention is that with the redo function, I did not get enough time to implement a conditional that the player can only redo a move if it is the correct player. Due to this lack of restriction, it would need to be assumed that the players would be aware of whose turn it was and when they could redo their move. To help this problem, there is a function that does not allow the players to replay onto a square that has already been taken. This is a functionality problem that definitely needs to be addressed when improving this game in the future.
In terms of the structure of the code, I think that it has been written quite cleanly. I gave all variables clear and relevant names that related to their purpose, and wrote comments for each function, explaining its purpose. There is a lot of information in the one script, so it may take another developer some time to try and figure out what is going on in some parts. If I am being critical, I would say that the code could have been simplified in places, and that probably would have helped with the interface problem as well. I think that this problem is due to my lack of prior knowledge of the language before this module, so the code may not be as clean as it could have been.
Something that I came across during research on user input (after I was getting some errors) was that the 'scanf' function that I used for user input is not actually a very effective function to use, in that it is difficult to handle errors with. To rectify the problem I was having, I used a very small function that I found on this stackoverflow[4] where another person was having the same problem. If I was starting this project again, I probably would have used a different function for getting user input.

5 Personal Evaluation

5.1 Challenges

One of the major challenges of this coursework was trying to decide on the correct data structures/algorithms to use. In reality, there are probably many different ways to structure this game, but my choice was to use arrays, stacks and queues. I think that I implemented them well, and made a good decision when choosing the structures that I did.
One of the main problems in completing this task was the error handling. Due to the fact that the game needs to be able to move back and forwards seamlessly, and that there is multiple opportunities for user input (and multiple opportunities for the user to input the wrong type of data) there had to be a large amount of error handling for multiple different scenarios. The error handling required a lot of debugging, which because I was using a text editor and terminal, had to be done using print statements to step through and ending up consuming a lot of time. Throughout that entire process, it was surprisingly straightforward to implement the data structures that I had planned, but the part which took up the most time was trying to get them as airtight as possible so that they would not cause a 'segmentation fault 11' and force the game to quit early if a user mishandled them.

5.2 My Performance

I am quite pleased with my overall performance on this task. A problem I commonly face with coursework is time management, but I started this one quite early, and did more planning than I have done in the past before jumping in to coding. I think due to the nature of this coursework being largely about planning and choice of how to build the program, planning ahead of implementation was a useful tool which aided me well along the process. One point where I could have improved would have been adding error handling and debugging to my time schedule. If I had considered the time that those aspects would have taken up, I could have planned my time better and possibly implemented some of the other enhancements that I did not get a chance to implement. I'm sure that adding extra enhancements and finishing the ones that don't quite do everything that I originally wanted them to would have improved my overall grade and performance on this task, which is where I could have improved.

5.3 Conclusion

In conclusion, I am happy with what I have learned throughout this coursework and I am satisfied with the end result. I definitely feel as though I have improved my knowledge of memory, storage, data structures and algorithms whilst working through the task at hand and doing research in conjunction with the labs we have been provided with. I am also confident that I have improved my skills in the C programming language and with the use of pointers, which is an added bonus to completing this task. In the future, I will continue to plan my course works more carefully, and give each aspect of the creation process more time when planning.

References

[1] P. Chhajer, "Geeksforgeeks: Time delay in c." https://www.geeksforgeeks.org/time-delay-c/. Accessed: 25.03.2019.

[2] R. I. Pitts, "Intro to file input/output in c." https://www.cs.bu.edu/teaching/c/file-io/intro/. Accessed: 25.03.2019.

[3] R. Belwariar, "Implementation of tic-tac-toe game." https://www.geeksforgeeks.org/implementation-of-tic-tac-toe-game/. Accessed: 16.03.2019.

[4] Jamesdllin, "Stackoverflow: How to clear input buffer in c." https://stackoverflow.com/questions/7898215/how-to-clear-input-buffer-in-c. Accessed: 25.03.2019.