Algorithms and Data structures Report

Luke Reeder
40277803@napier.ac.uk
Edinburgh Napier University - Algorithms and Data structures (SET08122)

1 Introduction

The aim of the coursework was to design and implement a text based tic tac toe game. This was to be done using C with a focus on the use of Algorithms and Data structures. The core of the coursework is to produce a working game that allows users to place pieces on a board. The game should be able to check for a winner after a move is made, and if a winner is found the game ends.

2 Software Design

The initial design of my software was to firstly get a working base of the game going. This was achieved by having a function design the board using print statements with character variables throughout the spaces to allow for inputs of either blank spaces, x's or o's. While making the board, the function calls a Clear screen command to clear the command prompt. This was done to make the screen look cleaner, and prevent spam as the screen would quickly become filled with old board states and user inputs, as well as potential error messages from invalid inputs. The board state was stored using an array of characters, initially set up as "-" to indicate blank spaces available as choices.

The next algorithm added was the check win function. This scanned the board for user inputs and compared the rows and columns as well as the diagonals to see if any three matched, and if so return a true variable. At first this was done by eight separate line checks, but was optimised to be in two loops to reduce clutter in the code.

Once the framework had been implemented for the game, the next step I took was to allow user inputs to the game, validate the inputs and then add the piece to the board. The game stores which user as an int and alters it between 1-2 depending on which turn it is, and uses this to set which piece is to be played. After each move is made, the game runs checks by calling the display board function to update the board, then checks to see if a winner is found. If no winner is found, the game continues and swaps to the other player. The game also has a running turn counter local to the current game played which if reaches turn 9 without a winner, the game sees the game is a draw and ends the game stating so.

After the initial game had been fully implemented with the full game functionality, I then started working on a replay system for the game. I chose to do this using a linked list to store the game states throughout, although a stack or queue could have also been used to store the states. I decided on linked lists due to feeling that they were more suitable to my needs, as I was able to store the game states as a doubly linked list, as well as have a single linked list able to easily link to the start of a game's linked list. To implement the linked lists, I created a struct using an array of characters to store the current board state as well as two pointers for the next node in the list, and the previous node in a list. A function to create the initial node was made by allocating the memory of the node, and then reading in the game board state into the node's array, as well as initialising the two pointers to NULL. The addNode function was then also implemented by creating temporary nodes to use in the function, calling the create node function, setting the head if the current head was NULL, and then setting the pointers from the previous and current nodes to each other to create the double link in the list. This was used to prepare for the undo and redo functions as I felt it was the best approach to tackle the problem with a clean solution.

After this linked list was set up, I incorporated it into the main body of the game system to record moves after they were made, as well as ensuring the final move was also recorded. This system was initially tested by only storing the last game played, before later adding a separate linked list to store a list of the actual games rather than the moves made within a game. This game list was made very similar to the initial linked list, by using a struct to store the data, this time using an Int for the game ID, a pointer to the head of the linked lists in which the moves were made, and a pointer to the next node in the list. This list was also given the same functions to create and add nodes to the list, however for the add node function, it did not have the previous pointer set as it was a single linked list.

The replay algorithm was implemented by feeding in a linked list to the function, which if not empty would loop through the list updating the board array, display the updated array, pause for a second, then move to the next node in the list until it had reached the end of the list. Initially for testing purposes, this was done by using the last list that had been created.

Since the program now had two options for functions for the user, a menu system was implemented using a case statement in a while loop to allow the user to choose whether they wanted to play a game or watch a replay, as well as exit the program.

For the undo/redo function, I added a separate options menu from within the game that allowed an input of 0 rather than the usual 1-9 for piece selection. If the user input 0 as an option, the game would then ask if the user wanted to undo a move, redo a move or exit the game early. The undo function was implemented by reading the current node of the linked list, setting the node to the previous from the pointer if a previous node existed, and then continuing on from there, while also adjusting the player and turn counter to match. The redo function was much the same, as it took the current node into the function, checked to see if there was a non NULL pointer for the next, and moved the board state along to that next node's array, again compensating for player and turn counts.

Once the undo and redo functions were finished, I amended the Game linked list to add a node to the list when a game was finished, with a pointer to the head of the double linked list from the game played, as well as storing a unique game ID. This allowed for a search function to be added for the replay system to view more than just the last game played. This was done by taking a user input, searching the linked list using a search algorithm through the linked list to match the user input ID to the game ID, and then if a match was found to call the print list algorithm to display the list.

3 Enhancements

If I had the time, I would like to try and implement a MiniMax AI to the game. This would be done by having the AI check the next possible move available to it and see if any possible moves are winning moves and if so, assign a positive score to that move. If none are available, assign a neutral score and then use recursion to check the next possible move. Since the next move however is an opponents move, it would assign a negative value to their possible moves and continue along this routine until either a winning state was found, or the game ends as a draw. One the AI had checked all of these possibilities, it assigns the best move available to it at that time and makes the move.

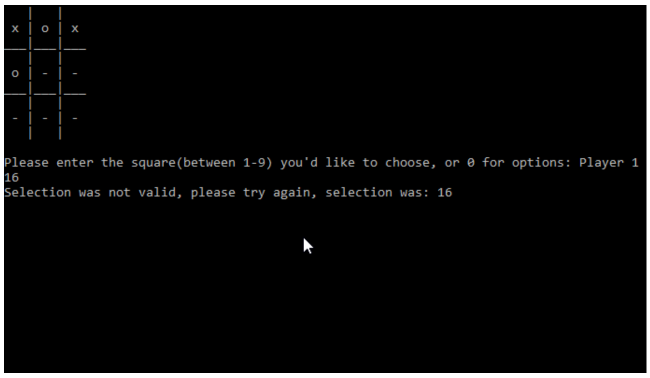
Other possibilities would be allowing users to add their own names rather than just being called player 1 and player 2. This would just be a cosmetic change to allow the users to add a personal touch to the program.

Some of the harder options to add in however would be scaling the size of the board. I did have ideas for how to do this such as changing the board array for the inputs to a 2d array, for while a 1d array works well for storing 9 characters, having 16 or 25 or higher would be a bit more tricky. I feel as if a possible solution to this would be have a 2d array of a set size, then pass this function into a new win check function that loops based on how big the array instead of just a set 3x3 array check. As for the game functionality, the main changes would just be to allow for the larger inputs for validation, but the main placements and win checks should be able to scale for larger sizes as well as the replay, undo and redo functions all retaining their functionality.

4 Critical Evaluation

For this coursework I feel as if the features I have implemented all work well, and are very robust and have their fully intended functionality.

Some minor adjustments I would also possibly make would be to possibly separate the code into separate header files just rather than have it all in the single main, however due to not being fully confident with how to do this and wanting to mainly focus on the actual functionality of the program this was left out for the time being. I did however try to optimise the code wherever possible while I was going through for efficiency.

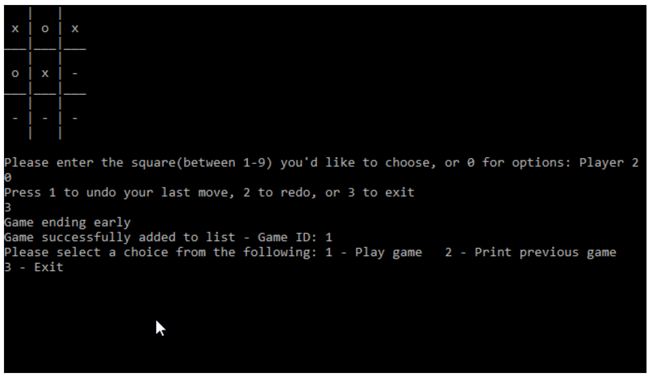


```
   |   |
 x | o | x
___|___|___
   |   |
 o | - | -
___|___|___
   |   |
 - | - | -
   |   |

Please enter the square(between 1-9) you'd like to choose, or 0 for options: Player 1
16
Selection was not valid, please try again, selection was: 16
```

Figure 1: Example of an invalid input being handled

The menu system is concise and clear and has a default case for invalid inputs which correctly loops back to allowing for a new different input from the user.



```
   |   |
 x | o | x
___|___|___
   |   |
 o | x | -
___|___|___
   |   |
 - | - | -
   |   |

Please enter the square(between 1-9) you'd like to choose, or 0 for options: Player 2
0
Press 1 to undo your last move, 2 to redo, or 3 to exit
3
Game ending early
Game successfully added to list - Game ID: 1
Please select a choice from the following: 1 - Play game   2 - Print previous game
3 - Exit
```

Figure 2: Main Menu for the user with options menu in game

The undo and redo functions work as intended, however there is a slight issue with the memory allocation for the undo function. This is caused due to C as a language not having a garbage collector implemented, so if a move is undone, it still has a link to the next node in the list to allow a redo function to take place. However, if a new move is made, a new link is made and the previous node is still allocated in memory but never used. A potential fix for this would be while making a move to check if the current node's next pointer is NULL, and if not, delete the next node from the list before continuing as normal adding the new node to the list in its place.

The replay function is designed to be robust and display a message stating if a game is unable to be found, and if one is found it will replay the game step by step using a delay between each move to allow the user to clearly see the game being played out.

The program has been tested to make sure there are no memory leaks, especially when replaying games, or undoing moves.

During implementation the game was not storing the initial nodes correctly so when it tried to read a NULL node it was spitting out random characters as the node had been created but not set.

5 Personal Evaluation

From this coursework I felt as if I had a better understanding of how to step back from a problem and consider the possible solutions for the problem, such as how to store the games. In this case I used linked lists but as stated I did also consider the ups and downs of queues and stacks, however normally I would think of the first solution and just try my best to implement that solution rather than consider all the ups and downs of multiple solutions.

As for a lot of the main testing and problems for the program, I tried to separate the code into manageable functions and work on it piece by piece before moving onto the next section. This was done by wherever possible reusing code in algorithms to only have to test it the once rather than rewriting it and possibly having different bugs. I also utilised the print function liberally to make sure I could see exactly what was happening where to try and further identify a problem when it arose. Some problems were not quite able to be solved as such though such as the memory leaks. These were solved by luckily having past experience and recognising that they were a potential error and then tried to use that as a fix before looking up other possible solutions.