Algorithms and Data Structures

Scott Postlethwaite
40281026@live.napier.ac.uk
Edinburgh Napier University - Algorithms and Data Strucures

1 Introduction

The aim of this report is to explain and justify the design decisions for my application. This includes the optimization of the application for both size and performance due to the use of optimal data structures & algorithms. The application consists of a tic-tac-toe game which runs on the command line and allows the user to decide the size of the game board, store previous games as well as being able to undo and redo their moves. This has been achieved in the optimal way for performance and storage thanks to the dynamic allocation of memory as well as the clever use of pointers & references to reduce the amount of memory used.

2 Design

2.1 The Board

The board is a 2 dimensional array of chars. This is dynamically allocated through asking users for a board size and using the malloc function for each element of the array. This is done so that the program does not waste any unnecessary resources while also giving the user more control over their game. A 2 dimensional array was used so that coordinates can be used as well as implementing features such as the dynamic win clause as well as the move checking. While a 1d Array would have used less memory, potentially sped up the program and would have been significantly less complicated to implement, I decided that the benefit of having a board made of coordinates would have made it an easier game to play and would open up more room for growth. If I had used a 1d array it would have been visualized as a grid of numbers as to allow the user to see which space corresponded to which number. This would have been less user friendly as well as less scalable when using larger game boards. It is for this reason that I decided the more complicated to implement, more user friendly 2d array approach was more worthwhile.

2.2 The Moves

The moves are inputted as coordinates. This is inside a while statement where while nobody has won the game and the move is not legal, the player will be asked to make a move. The player is then changed after a legal move is made. These moves are simple integers which decide the position of the 2d array that the player's char will populate. Upon inputting their move the position will be checked to see if it is first of all within the limits of the game board and secondly whether there is already a player in this space and finally whether this move will win the game. To implement these checks I wrote 2 dynamic functions that will read in the 2d array as well as the move and the size of the board in order to loop through the 2d array. The move checking simply checks whether the coordinates are each less than the size of the board and more than or equal to 0. After this it will go to the position in the array and if it is empty (populated with a space) it will return true. The win clause loops through the array checking each possible winning combination to see whether the game has been won. If it has it will output the name of the winner along with the option to save a game. I decided to omit any kind of player class as I feel this would be an unnecessary addition as all it would do is make the file size bigger while not adding any additional functionality to the application. Instead the player is simply a char variable that is used to populate the array. Each move this char is changed to the other player automatically so that the user does not need to input x or o every move.

I also allow players to undo and redo their moves. I do this through the use of a 3 dimensional array that is populated by each state of the game board. To loop through this I have a counter that increments each move that is made and is used to point to the current position of the game board. When a move is undone the game board is simply set to the position in the array of the counter minus 1. I had previously implemented a doubly linked list for the same function however it ran slowly and had several errors with the redo portion of the function where it would simply crash the program. I have decided to stick with the array solution rather than using a doubly linked list as it runs faster and more seamlessly while omitting the glitches and bugs present in the previous solution.

2.3 The Games

Players are able to save a replay of their past game to a file in order to watch it again later. This is achieved by having the user input a name for the game, providing it with a file extension through string concatenation and writing the 3d array of game states into the file line by line. To re-watch the game the user simply inputs the name of the game, the file extension is added as before, then the file is read back into the 3d array line by line. This 3d array is then stepped through visualizing the game board at each step. While this could have been achieved by outputting the coordinates of each move rather than the entire game board I felt that having the entire board would lead to more flexibility and stability in the long run. If I were to go through this again I would have used coordinates to avoid the growing pains present when implementing this solution as I feel it was an over complicated solution to a simple problem with few perceivable performance benefits.

3 Implementation

The implementation is very simplistic. Users decide the size of the board, make moves, undo and redo these moves as well as saving a copy of the game to replay later.
This is all done on the command line. In order to effectively showcase the game board to users a visualization function was implemented.

4 Implementation Evaluation

4.1 Does It Meet The Spec?

As the games moves can be undone and redone, completed games can be saved and replayed and the size of the games board can be chosen by the user, I feel this not only meets but exceeds the specification provided. I put a lot of thought into the data structures used as well as the layout of my functions and the program runs quickly and smoothly as a result. As this is the aim of the module I feel that I have succeeded in this regard and have therefore performed well in this coursework overall.

4.2 Potential Enhancements

The only enhancement that I would consider implementing would be a single player option against an AI opponent. If I were to implement this I would use the minmax algorithm for the opponent to find the best moves. The only reason that I did not implement this is that I felt it far exceeded the scope of the coursework and of this module so my efforts would be best put to the refinement of the core elements of the application.

5 Personal Evaluation

I feel that although the application is extremely simplistic, the fact that C requires you to allocate memory yourself as well as having far less native data types, this taught me a lot about resource management and optimization while giving me a greater appreciation for the native data structures in modern languages.

I faced many challenges while working on this project. The most notable of these was dynamically re sizing the game board. My main issue with this was that there is no documentation online for dynamic user allocated 2 dimensional arrays. I set about this by learning to allocate the memory for a standard array. To do this I had to create a global num variable to store the size of the array. This allowed me to pass it through to my main method, move validation and win clause. After this I experimented with the 2 dimensional array. Initially my problem was that the num variable was initialized as null and was changed by a command line argument. This caused a null pointer exception. To sort this I initialized the variable as 1 and created a while loop so that while the num variable was less than or equal to num it would prompt the user to change the size of the board.

Another challenge for me was creating a dynamic win clause. As I initially had a static win clause that only allowed for a 3x3 board, when the dynamic board was implemented it would state that a player had won when someone had 3 concurrent pieces in the top 3x3 spaces in the grid. To sort this I had to create a function with a series of for loops and if statements to check whether the X and Y axis had the same number of concurrent pieces as the game board itself. Once this was implemented I had to find a way to handle the diagonal win clause. To set about this I had an outer for loop which incremented a counter from 0 to the size of the board. I then checked whether the player that just made a move had pieces in each of the points where the x & y coordinates equaled the counter. If there was a piece in each of these positions it notifies the player that they have won. The horizontal and vertical clauses work by adding a nested for loop which sets the x or y coordinate as the original i counter and sets the other as a new j counter which loops through all of the horizontal or vertical spaces to check whether they are populated by the current player and if they are it will notify the player that they have won.

Yet another issue that I encountered while developing this application was during the implementation of my undo/ redo feature. I initially set about implementing a double linked list to store the game board's position. My issue with this is that it would only store a pointer to the game board which meant that each node in the list had the same value. I then set about implementing an array to store the game board. Initially it was simply a pointer to the board which used little memory but also did not allow me to change the game state. To fix this issue I had to convert it to a 3 dimensional array in which the first would act as an index, storing which state we were in, and the second two being copies of the current game board. This was to allow seamless undoing all the way to an empty board and redoing until a game over.

The final issue that I encountered was int the replaying of saved games. Initially I had it set up to allow the user to do this after the game was finished however I did not allow the user to save the game to replay it later. To add this functionality I saved my 3d array to a file line by line and read it in when the file name was input. This caused several errors for me. Firstly, I was unfamiliar with string concatenation in c. This caused several crashes without error messages. Upon fixing these it became apparent that I was writing the array to the file with the syntax from C++ i.e. trying to store the full array with an fprintf statement. Once I realized this I set about outputting each element of the array to my output file. This left reading the array from the file to be the only issue. When I tried to read it the same way as it was written my game board was skewed with the values in the wrong places. To fix this and ultimately to complete the program I had to dynamically allocate the memory to the 3d array by placing the array's size at the top of the file and using a malloc statement based on this. From here I...

References