Edinburgh Napier UNIVERSITY

Coursework

Ovidiu-Andrei Radulescu
40283288@live.napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SET08122)

1 Introduction

Tic-Tac-Toe or Noughts and Crosses is a very popular two player game, where each player takes a turn in placing their marker on a 3x3 grid, with the first player to get 3 of the same in a row, column or diagonal wins. The game is very simplistic and often popular with young children, the game often resulting in a draw after you know how to counter.

For this coursework, the task has been to implement a text-based version of Tic-Tac-Toe. As I thought this would be a very simplistic game and not as fun to play after a while, I decided to extend mine to a version of Recursive Tic-Tac-Toe(RT3 for short). RT3 is a 3x3 grid, in which every tile has another 3x3 grid where a normal game of Tic-Tac-Toe is played, therefore an RT3 game is actually 9 games of Tic-Tac-Toe played simultaneously. Another specific rule of RT3 is that wherever a player chooses to place their marker in a small board, the next player plays in the same position in the big board(for example if a player chooses to place a marker in tile 3 on the board 4, then the next player chooses their tile in board 3). If a small game is won, then the next player gets to choose to play anywhere in the big board, or if they need to play in a board already won, they can again choose to play wherever they want. The win condition is the same as a normal game of Tic-Tac-Toe, whoever gets 3 markers in a row,column or diagonal on the big board wins.

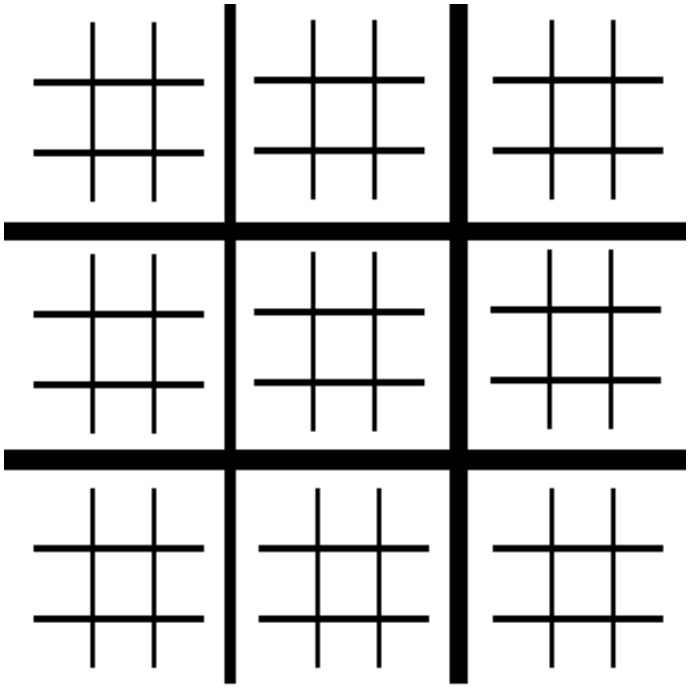

Figure 1: **A recursive Tic-Tac-Toe Board** - A representation of how an RT3 board looks

The app that I have made implements all of the above rules of RT3, as well as an undo-redo system, and a score for each player. The game displays the succession of moves that have been made by the players during the past game. The game has validation on player input, as to ensure the game won't crash.

2 Design

The coursework has been written in C, as specified in the specifications. After getting a good grasp on C in the first year and during the practicals for this module, I wanted to improve on my skills by doing something both exciting and complex, which is why I decided to do RT3, which is the main focus of this work.

In order to keep the data structures for the board simple and easy to understand, a normal Tic-Tac-Toe board is actually just a struct that contains an 1D array(the board itself) of 9 characters (a string basically), and a char that signifies if the board has been won, used to check easily for a win condition [Listing 1]. The big RT3 board is an 1D array of nine Tic-Tac-Toe structs. I have concluded that keeping it to 1D arrays makes it easier to work with, making both of them 2D arrays would have made the whole program unnecessary complicated, as the rules for winning Tic-Tac-Toe can be easily implemented in a 1D array.

The player struct [Listing 2] stores the player's chosen name, their sign(X or O, which is a char) and their number of games won. This seemed like the most sensible choice, as to not clutter the main function with too many variables, using a struct for each player keeps it organised.

The history struct [Listing 3] is a node that contains a move made by a certain player, it stores the player's char(X or O) and the tile it was placed in, shown by the indexes of the big board and the small board. The struct is based on the linked list structure, as each node holds a pointer to the previous and the next node. The history struct is used in the program for two things: to store all the previous moves made by the player, as well as a list of moves the player has undone(used for the redo functionality). The way the undo function [Listing 4] [Listing 5] of the program works is by popping the last node of the history list(similar to a queue), and then adding that node to the undo list. I have decided to use the mix of linked list nodes and queue functions in order to make it more efficient on time and memory, as I only remove a node from a list and add it to the other without deleting and creating any other nodes. By doing this, the redo functionality uses the same function, by passing the list arguments in reverse and specifying the type (0 for undo and 1 for redo). Whenever a player makes a new move, the undo list is cleared.

Listing 1: The Board Struct

```
1  struct Board
2  {
3  char board[SIZEOFBD];
4  char won;
5  };
```

Listing 2: The Player Struct

```
1  struct Player
2  {
3  char sign;
4  char name[256];
5  int score;
6  };
```

Listing 3: The History Struct

```
1  struct History
2  {
3  int bigBoard;
4  int slotChanged;
5  char sign;
6  struct History *next, *prev;
7  };
```

Listing 4: The Undo Function

```
1  void UndoMove(struct Board **bigBoard, struct History **listFrom , struct History **listTo, int way)
2  {
3      struct History *removedMove = popMove(listFrom);
4      if(removedMove != NULL)
5      {
6          Change(bigBoard, removedMove, way);
7          appendNode(listTo, &removedMove);
8      }else
9      //if undo/redo is unsuccesful, turn doesn't change
10         turn -= 1;
11 }
```

Listing 5: The Undo helper function

```
1 //used by the undo/redo function, the 'way' variable specifies if it's an undo(0) or a redo(1)
2 void Change(struct Board **bigBoard, struct History *h, int way)
3 {
4    int index = h->bigBoard;
5    int slot = h->slotChanged;
6    char sign = h->sign;
7    if(way == 0)
8        {
9            bigBoard[index-1]->board[slot] = (slot+1)+'0';
10           bigBoard[index-1]->won = 'f';
11           bigBoardIndex = index;
12       }
13   else
14       {
15           bigBoard[index-1]->board[slot] = sign;
16           bigBoardIndex = slot+1;
17       }
18
19 }
```

3 Enhancements

The game has, in my opinion, a good core functionality and for the user, it is fun to play, adding more decision-making and excitement compared to a normal game. The biggest improvement that can be made is to the core gameplay. A big feature that could be implemented is to choose the amount of levels of recursion a game can have. A user can select to play just a normal game(no recursion), the version currently implemented for this coursework(1 level of recursion), or 3, 4 or even more levels of recursion. This coursework started by building a normal game of no recursion, with all the functionality working(undo/redo, players etc), and then adapting it to recursive. As I modified the program to work with RT3, not that much had to be added, the way user choice was inputted and the undo/redo being the biggest modifications, apart from adding the board struct. I therefore concluded that making more levels of recursion wouldn't be too hard, but it would take time to make sure it will work, as well as finding a different way of displaying the board, as the user would find it increasingly hard to tell which tile is which.

Another big feature would be the implementation of single player games versus an AI, but as that would have been straightforward for a simple Tic-Tac-Toe, as there are only 9 moves in total to make and there's not much strategy to it, the problem becomes more challenging for recursive, as the total possible moves is now 9^2 (9^3 for another level and so on). Apart from the moves that need to be calculated, the strategy to win a single level RT3 game is one where you're forcing your opponent's hand at giving you

a free move. By adding another level of recursion, I have noticed that the strategy changes considerably, as you can influence the 'zones' you can play in and win the board bit by bit.

Smaller features I have not implemented, mainly due to time constraints could have been persistent score keeping, and have the game be presented in an arcade style with high scores(beating the other player in the fewest moves). Other than player's scores, past games could be saved and either replayed from a point or continued if not finished, which would be absolutely necessary for multiple levels of recursion, as it takes hours to finish one game.

4 Critical Evaluation

I started this coursework with the goal of making a game that shows a good use of data structures and algorithms, while being exciting and challenging for the users to play. All the features of the game work well, and the user input is carefully verified as to not cause an issue with the functions. I am very pleased with the memory management on the history of moves list and the undo list, as nodes get moves from a list to the other depending on the user input, without deleting and creating new nodes, the only time nodes are created being when new moves are made, which is also when the undo list is cleared(as the moves in it are now redundant).

5 Personal Evaluation

As I've had experience working in C before, writing a big complex program that is ultimately a game, and requires a lot of data structures and algorithms for logic,wasn't something that I have done much in the past, even less so in C. The coursework has been a great way to apply all the new knowledge learnt during lectures and tutorials in an imaginative way that left a lot of space for finding how to combine that knowledge to the task at hand in order to create something useful.