Coursework Report

Pablo Sanchez Narro
40486559
Edinburgh Napier University - Algorithms and Data Structures (SET08122)

1. Introduction

The aim of this coursework is to demonstrate my understanding of Algorithms and Data Structures. The task is to implement a console-based Sudoku game, paying special attention to the algorithms and data structures used for it. For this coursework, I programmed in the C programming language a game where you can choose to play in an Easy, Medium, or Hard mode. It is also possible to save the current game into a text file to play it afterwards, so it is obviously also possible to import an existing game. Finally, there is an option to play against the time and it is possible to specify how many minutes you want the game to last.

2. Critical Evaluation

2.1 Sudoku Board

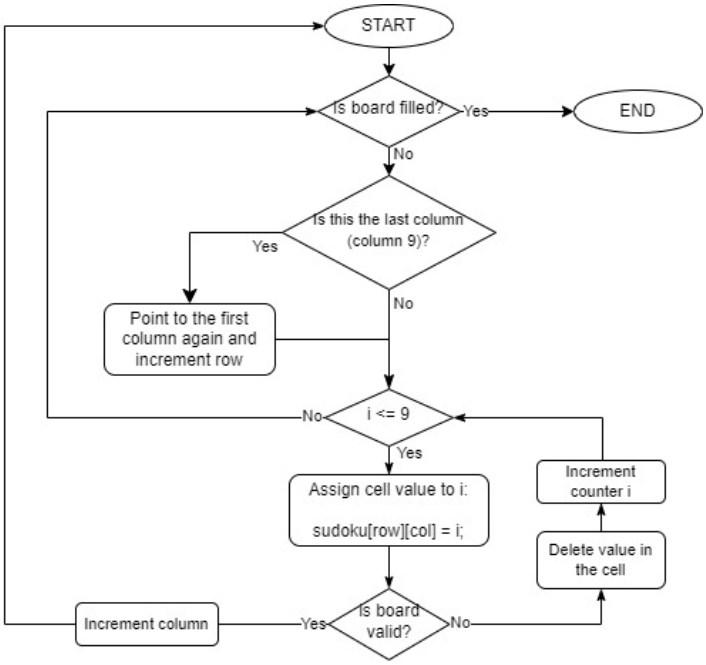
My Sudoku Board implementation is made with a Recursive-Backtracking algorithm. First, a 2D array is initialized to zero to hold the sudoku board.

```
int sudoku [9][9] = { 0 };
```

Then, if the user wants to play a new game (no game has been imported), it calls the *populateBoard()* function

```
void populateBoard(int sudoku[][9], int numToPopulate)
```

Depending on the difficulty selected in the menu (Easy, Medium, Hard), the variable *numToPopulate* passed to the function would have different values (more cells filled for the Easy mode). The *populateBoard()* function first calls the function ***populateBoardRecursively(),*** which is probably one of the most important functions within the application. This, creates a doable sudoku board using a Recursive-Backtracking algorithm **[1]**. Below is a flowchart of my implementation, made with *draw.io*.



Algorithm 1: Recursive-Backtracking algorithm. ***populateBoardRecursively()*** function.

This algorithm contained in the function is the core of my implementation. I chose it, because it is an extension to Brute-Force, in which the implicit constraints are evaluated after every choice (as opposed to after all solutions have been generated) **[2]**. This is done thanks to the abstraction of the *isBoardValid()* function, which checks if there is any duplicate in the row, column, or box (3x3) **(see Appendix A for the code).**
After this is executed, the *populateBoard()* function continues. It continues assigning zeros to random places in the board, using the *rand()* function in C. The board is stored in the 2D array called sudoku.

2.2 Undo and Redo

This feature is integrated with a stack. As explained in my previous report, a stack is a list with just the operations of inserting and deleting data items at the "top" **[3]**. To make the stack work, I first declared a Struct and added functions, *Push()* and *Pop()*. The stack is initialized by setting the top pointer of the stack to -1. Below are the pseudocode for the **Push** and **Pop** functions:

```
if top pointer == (max capacity (81) - 1) then
        stack overflow!!;
end
else
        top pointer++;
        place value into stack;
end
```

Algorithm 2: Push value to stack

```
if top pointer == -1 then
        No moves to redo!!;
end
else
        access the value stored in the current location;
        set the pointer to point to item below current;
end
```

Algorithm 3: Pop value from stack

Every time a value is placed, if it is valid, the number is pushed into the stack. The arrays *int rowToUndo[MAX]* and *int colToUndo[MAX]* keep track of the rows and columns played by the user, so we can keep track of where the values pushed or popped should be placed when undid or redid. When the player chooses to undo a move, the number at the top of the stack (value popped) goes to the *int movesRedo[MAX]* array, which allow us to Redo effectively (pushing that value again to the stack).

2.3 Exporting and Importing games

My implementation for this turned out to be simple, but effective. In essence, when the player decides to save the current game, it first asks the player to give the current game a name. Then, it saves the sudoku 2D array into a text file. This is done with two for loops, as if we wanted to print the board. By contrast, when the user decides to import a game, it searches for the name in the text file, and once found, it reads the 2D array and saves the values into the sudoku 2D array.

2.4 Playing against the time

In the initial menu, the user can choose to play against the time, and they can also choose the minutes. This is done thanks to the *time.h* header file. The game first saves the initial time in which the sudoku is displayed ("initial_time"), as well as another time called "current_time", which will be updated every move. Once the difference between the initial and current time (calculated with a function provided by *time.h*) is equals or bigger than the time specified by the player, then the game is ended.

3. References

[1] Abiy, T., Oli, B., & Lazar, A. (n.d.). Recursive Backtracking. Retrieved from Brilliant: https://brilliant.org/wiki/recursive-backtracking/#mini-sudoku

[2] Andhariya, A. (2017, March 31). Sudoku Solver Using Recursive Backtracking. Retrieved from Code Pumpkin: https://codepumpkin.com/sudoku-solver-using-backtracking/

[3] Mehlhorn, K., & Sanders, P. (2008). Algorithms and Data Structures: The Basic Toolbox. Springer

Demo of my Sudoku

https://www.youtube.com/watch?v=-OJvFgxryew

4. Appendices

Appendix A:

isBoardValid() function. Checks if the board follows the sudoku rules.

```
int isBoardValid(int sudoku[][9])
{

    int i = 0;
    int j = 0;

    for (i = 0; i < 9; i++)
    {
        if (duplicateInRow(sudoku, i))
        {
            // Found a duplicate in row so board invalid
            return 0;
        }
    }

    for (i = 0; i < 9; i++)
    {
        if (duplicateInCol(sudoku, i))
        {
            // Found a duplicate in column so board invalid
            return 0;
        }
    }

    for (i = 0; i < 9; i++)
    {
        for (j = 0; j < 9; j++)
        {
            if (duplicateInBox(sudoku, i - i % 3, j - j % 3))
            {
                // Found a duplicate in column so board invalid
                return 0;
            }
        }
    }
    // All tests cleared so board is valid
    return 1;
}
```

Appendix B:

Functions that look for duplicates: *duplicateInBox()*, *duplicateInRow()*, and *duplicateInColumn()*

```c
int duplicateInCol(int sudoku[][9], int col)
{

    int colDigitCounter[9] = { 0 };
    int i = 0;

    // Counting occurence of each digit in column
    for (i = 0; i < 9; i++)
    {
        if (sudoku[i][col] != 0)
        {
            colDigitCounter[sudoku[i][col] - 1]++;
        }
    }

    // Now checking if any character appeared more than once
    for (i = 0; i < 9; i++)
    {
        if (colDigitCounter[i] > 1)
        {
            // Found duplicate in column
            return 1;
        }
    }

    // No duplicate was found
    return 0;
}
```

```c
int duplicateInBox(int sudoku[][9], int r, int c)
{

    int boxDigitCounter[9] = { 0 };

    int i = 0;
    int j = 0;

    // Counting occurence of each digit in box
    for (i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (sudoku[i + r][j + c] != 0)
            {
                boxDigitCounter[sudoku[i + r][j + c] - 1]++;
            }
        }

    }

    // Now checking if any character appeared more than once
    for (i = 0; i < 9; i++)
    {
        if (boxDigitCounter[i] > 1)
        {
            // Found duplicate in column
            return 1;
        }
    }

    return 0;
}
```

```c
int duplicateInRow(int sudoku[][9], int row)
{

    int rowDigitCounter[9];
    int i = 0;

    for (i = 0; i < 9; i++)
    {
        rowDigitCounter[i] = 0;
    }

    // Counting occurence of each digit in row
    for (i = 0; i < 9; i++)
    {
        if (sudoku[row][i] != 0)
        {
            rowDigitCounter[sudoku[row][i] - 1]++;
        }
    }

    // Now checking if any character appeared more than once
    for (i = 0; i < 9; i++)
    {
        if (rowDigitCounter[i] > 1)
        {
            // Found duplicate in row
            return 1;
        }
    }

    // No duplicate was found
    return 0;
}
```

Appendix C:

Screenshots of my Sudoku game

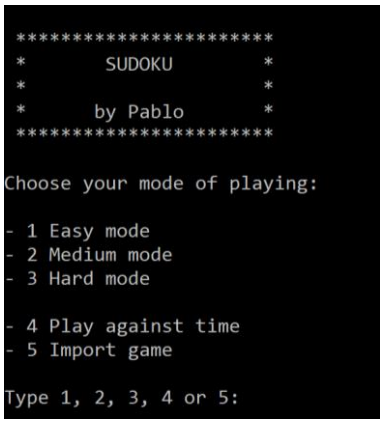


Figure 1: Menu

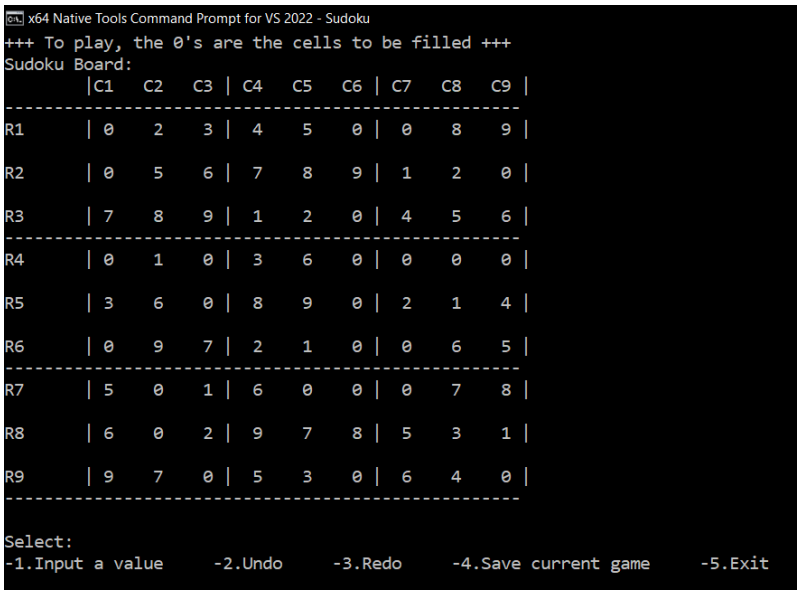


Figure 2: Sudoku board

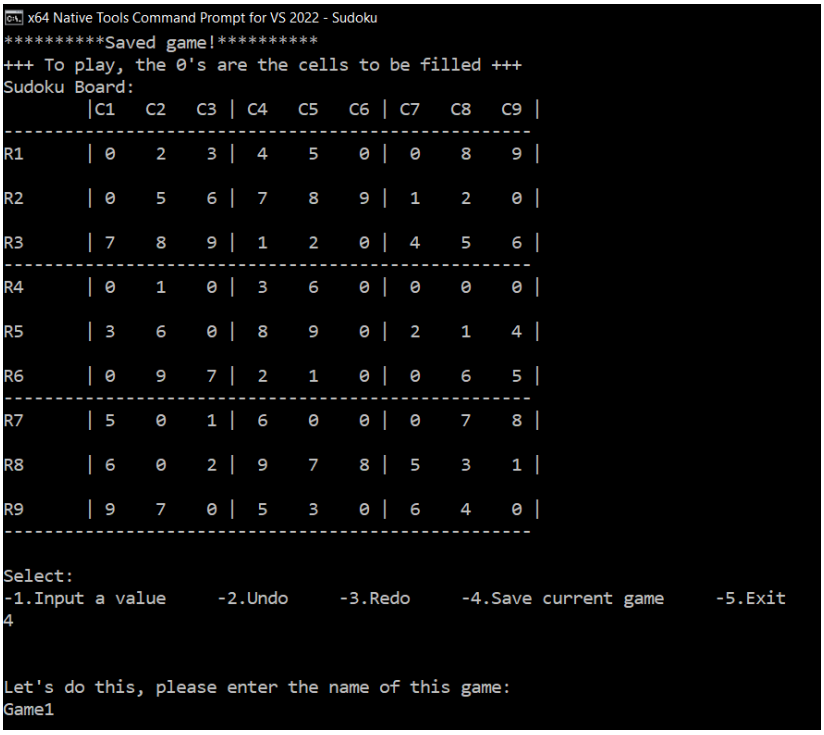


Figure 3: Save (export) game

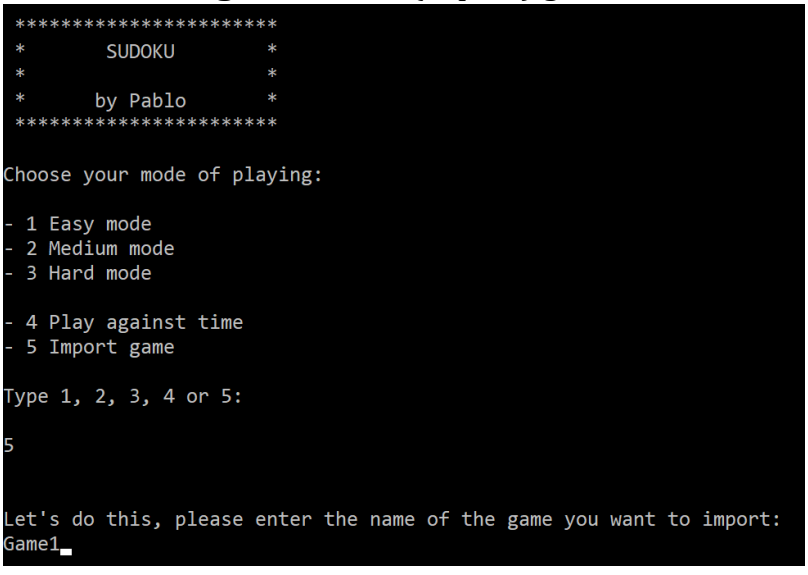
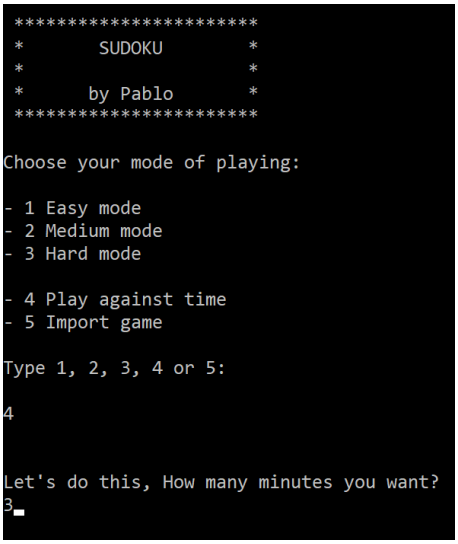


Figure 4: Import game



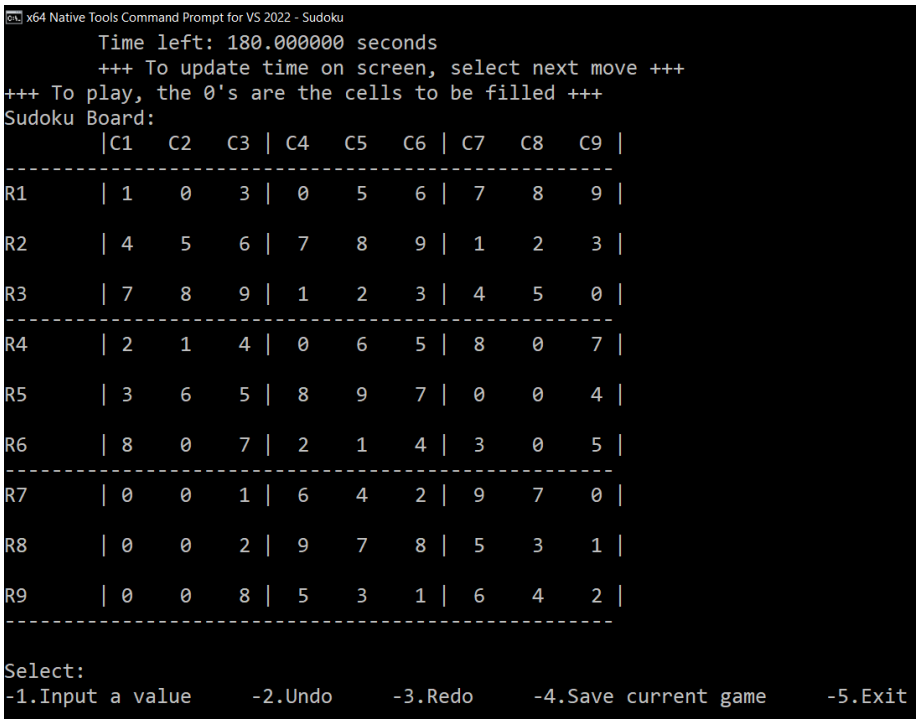


Figure 5: Play against time