Marco Vavassori Matriculation number: 40504087

Algorithms and Data Structures – Sudoku Game Report

Introduction:

This report is a deep analysis of the data structures and algorithms of my own, and of other online users' attempts to create a working Sudoku Game. It is in fact the aim of this coursework to create a working command prompt Sudoku Game, which will require a wise use of algorithms and data structures. All the material learnt from the lectures and the information gathered during my online research will help me with this task.

But why are data structures and algorithms so important? Everything that has been programmed and that will ever be programmed is about manipulating information, also called data in the computing field. Each data structure is a different way of organizing collections of data while each algorithm is a different way of handling it, without these two data would just be static and unusable.

This report contains other 4 sections, the Body, divided in "literature review" and "my approach", a Conclusion, where I define what I have understood from this research, References, citing every source I got information from and Appendices, containing some extra information.

Body:

Literature review:

For this literature review I am going to go over and comment some already made Sudoku puzzles found online and their data structures and algorithms, so I can get a better understanding on the field and improve my knowledge to achieve a better final result.

The code from Bryce Matheson (cplusplus, 2010), made in C++ looks very neat. I appreciate the choice of making it using such a programming language which, since operating at a very low level, provides speed of data handling to the end user of the Sudoku game. I think I will follow the way he is printing the cells row by row and block by block, it is a clever way I did not think of. I personally think that the algorithmic choice he made of storing loads of Sudokus in txt files and reading them to then display them is a smart way of avoiding hardcoding but at the same time it still is a kind of hardcoding because those files are manually filled in. Using a puzzle creation algorithm, like backtracking, could have absolutely given his code more value.

For this Java code (Java Game Programming, 2021) I find myself in agreement with the choice of using an Enum data structure to store the Cell state since a Cell can have a few different states, meaning it can be perfectly stored in an Enum. I will implement an Enum class to store the states of the cell as well instead of using class attributes, but I would not implement it in the same way: the four possible states could be reduced, from the ones he wrote, to just SHOWN, NO_GUESS and GUESSED; the existence of the 4th state (WRONG_GUESS) means that he is allowing people to insert numbers even if there is already the same number in the same column/row/block. This is of course a personal choice, but, since I am planning on not letting the user do such mistake, I will not use that logic. After deep research about Sudokus to discover different details needed for the generation of the partly filled grid, I have read that, depending on the given numbers, there could be multiple ways of solving one Sudoku so it is a reason for me to implement the Enum data structure and remove the attribute which holds the number that a certain Cell should contain.

During my research I also found a nice grid idea that I could possibly use instead of my current one (see Appendix A).

Most of the Sudokus that I have found online utilize the backtracking algorithm to generate a Sudoku grid. "Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time" (GeeksForGeeks, 2021).

I consider using backtracking the best solution that I found since hardcoding puzzle games into the code could be considered unprofessional. Despite that I might still try to implement my own algorithm to create a puzzle, generating each time a random number and checking if another same number is already in the same row/column/box, otherwise it can be written and the next cell can be examined.

The other issue is that not only the grid needs to be filled, but values also need to be removed. An article I found says that beside removing values one by one and testing if there is a solution, a technique that could stress the hardware less would be to "define a parameter p (where $0 \leq p \leq 1$). Each cell in the grid is now considered in turn, and is removed with a probability of exactly $1 – p$." (Rhys, 2007). But that way the generated puzzles are likely to be inappropriate from being solved by a human.

Some even suggest to just start adding numbers into an empty grid until one and only one solution is possible. I will try all those three approaches and consider which one works the best in terms of time and precision.

My approach:

Out of all the programming languages I could have chosen from I decided to utilize Python. The reason behind that thinking is pretty simple: Python is the language I am more familiar with. Python might not be the most adapt for this task, not because of a lack of data structures, since it owns a wide range of them, but for two other reasons:

1. This application will manage a modest amount of data, reason why I could have decided to use a programming language that is more adapt when handling streams of data, like C and C++.

2. In Python important and widely exploited structures like dictionaries and lists are not differentiated into loads of other minor structures, like in Java with Hashmaps, Linked lists, Vectors, etc. (see Appendix B and Appendix C for a clearer overview), but there is only one type of them, doing the work of all the substructures.

The next step is to talk about my approach on two different sides, which are design and architecture. Regarding the design I first considered the possibilities that printing through command line can give you, which are not many, and consequently decided to exploit this advantage to make my game look easy and clean to read. The features I decided I will implement are:

- The first game's output is a set of instructions that the user can read for as much time as needed and then start playing, of course the user can always re-display them if anything was missed.

- The game user interface is designed to have the least possible elements, in fact at the top is positioned the title of the game, in the middle the grid and at the bottom the user input section, where the user can choose the next move.

- Every time the user inputs something the command line is cleared and the three game sections (explained above) are reprinted, so the user does not get confused with previous outputs.

- The moves input will work through a coordinates system, like in a game of Battleship. The coordinates will be displayed on the top and left side of the grid and the user will be taught how to use them.

- There will be a countdown that will make the game more challenging giving the user only a few minutes to complete the puzzle. This is an extra feature and, if it will be implemented, the user will be allowed to switch it off.

- I worked hard on the indentation of every single line of output, so that it is not either too close to other text or to borders and corners. And I kept texts as short and informative as possible.

- I decided to display a fancy Sudoku game name (Vavassudoku) at the very top of the command prompt tab, to give it a little bit of a personal touch.

Regarding the architecture there follow my personal architecture choices:

- I could have implemented my game using tons of different architectures, what I decided to do was to store all my data in two main classes: a Board class that contains the game settings and the cells and the Cell class, which stores all the data required for every single cell to work properly.

- To know all the time if a cell is overwritable, empty and not empty I will implement an Enum data structure to select the Cell's state from.

- To store the Cell objects I am currently using a 1d array since the array does not have a fixed size, meaning it will be faster than a 2d array. But I will consider making this array of fixed size, since the grid's cell number will not change, and then change it into a 2d array.

- To record the history of the moves I will save every single movement as a string into a list. Managing undo e redo movements will be quite challenging and that is how I was thinking of implementing it: to go back (undo) the program will read the list, search for the current move, pop it off from the end of the list and add it to another list of movements to redo, and undo it. To go forward (redo) the program will read the list of movements to redo, and get the following move, redo it and pop it off from the beginning of the list and add it to the list of movements. Of course every time the user goes back with moves, and makes a new move, there will not be the possibility of redoing anymore.

- Even if not needed I thought that using the help of some useful tools could make my Sudoku game look even better, those useful tools are libraries. Python is known to be the best programming language for libraries, but, since we are not allowed to import libraries that provide special data structures, for now I will only import two libraries called Colorama and Playsound, to respectively introduce colours and sounds into the game.

Conclusion:

To conclude my analysis, after all my considerations, I can say that, even if there is not real need of it, I could have chosen a programming language with a slightly wider choice of data structures, for example Java or C#. I am not sure if it would be worth translating all the already written Python code since it would not bring major advantages and since the knowledge grown from this report made me think differently about the real weight of the data I am handling and I now have no regrets about my programming language choice.

I am also confident about the data structures that I decided to utilise but if any point of my previous analysis or of a future one will lead me to think otherwise, I will change what necessary.

References:

- cplusplus, (2010, December 3). *Sudoku Game*. Retrieved March 3, 2022 from http://www.cplusplus.com/forum/beginner/32467/

- cppsecrets, (2021, August 12). *Sudoku Console Game using C++*. Retrieved March 3, 2022 from https://cppsecrets.com/users/141731151079711010097114105641121044610510511611446979946105110/Sudoku-Console-Game-using-C00.php

- GeeksForGeeks, (December, 2021). *Backtracking | Introduction*. Retrieved March 2, 2022 from https://www.geeksforgeeks.org/backtracking-introduction/

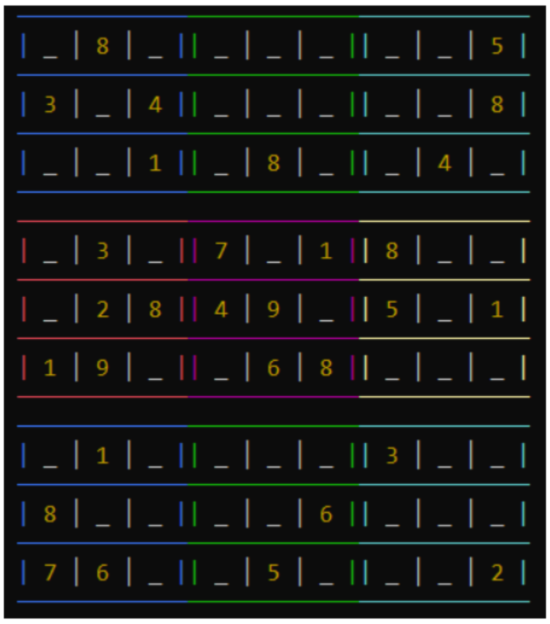
- Java Game Programming, (May, 2021). *Java Graphics Programming Assignment – Sudoku*. Retrieved March 3, 2022 from https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaGame_Sudoku.html

- Rhys, L. (2007, May 1). *Metaheuristics can solve sudoku puzzles*. Retrieved March 2, 2022 from https://link.springer.com/content/pdf/10.1007/s10732-007-9012-8.pdf

- Wikipedia, (2021, October 31). *Java collections framework*. Retrieved March 2, 2022 from https://en.wikipedia.org/wiki/Java_collections_framework

Appendices:

- **Appendix A**: Sudoku grid.



```
| _ | 8 | _ || _ | _ | _ || _ | _ | 5 |
| 3 | _ | 4 || _ | _ | _ || _ | _ | 8 |
| _ | _ | 1 || _ | 8 | _ || _ | 4 | _ |

| _ | 3 | _ || 7 | _ | 1 || 8 | _ | _ |
| _ | 2 | 8 || 4 | 9 | _ || 5 | _ | 1 |
| 1 | 9 | _ || _ | 6 | 8 || _ | _ | _ |

| _ | 1 | _ || _ | _ | _ || 3 | _ | _ |
| 8 | _ | _ || _ | _ | 6 || _ | _ | _ |
| 7 | 6 | _ || _ | 5 | _ || _ | _ | 2 |
```

Figure 1 Sudoku grid (cppsecrets, 2021)

- **Appendix B**: Java Collection class.

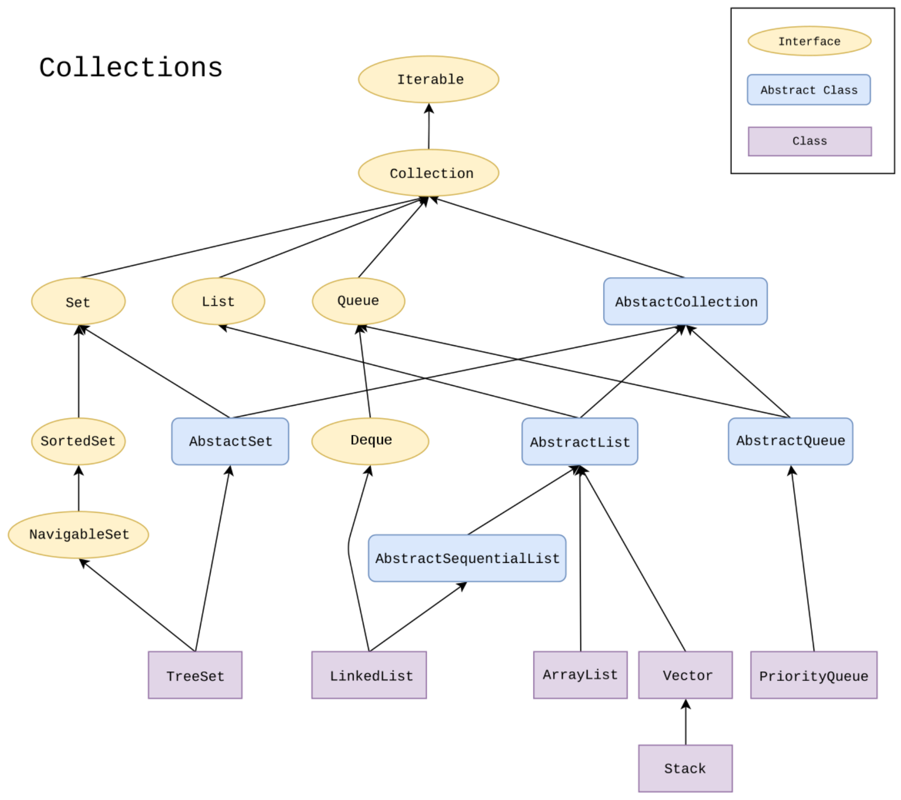


Figure 2 Java Collection class (Wikipedia, 2021)

- **Appendix C**: Java Map class.

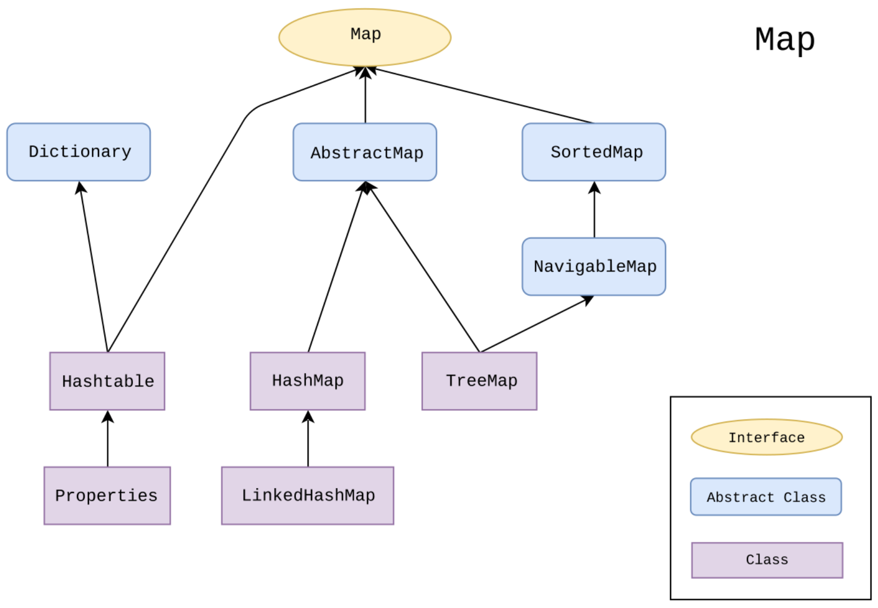


Figure 3 Java Map class (Wikipedia, 2021)