Report for C1 Research Computing Coursework

CRSiD: tmb76

University of Cambridge

December 14, 2023

Contents

Chapter 1

Introduction

In this report, an overview of the developping process of a python sudoku solver is given. The aim is to detail the software development of the solver, delving into the experimentation as well as how the code was improved, beyond it functioning as intended. The solver relies on a non-naïve backtracking algorithm. First covered, will be a rational of the choice of solving algorithm and the prototyping of said solver. Then, a larger section will describe the actual development of the code, where the prototyping was wrong, and what solutions were found. This will include profiling, after the solver was finished, to deal with any performance bottlenecks. Beyond the development of the solver, the report will also cover the validation and unit testing of the code, which ensures the code is robust. Finally, the report will cover how the code was packaged and the useability of the solver.

Chapter 2

Solving Algorithm and Prototyping

2.1 Solving a Sudoku Puzzle

When solving a sudoku using brain-power, one has multiple technics they can use. Most simple is to go through each cell, and using the sudoku constraints, eliminate impossible values, and hopefully find that there exists only one possible value for the cell. Then, if the sudoku is easy enough, a large part of the cells can be filled in this way as finding the solution to one cell may “unlock” another and so on. This can be referred to as the candidate-checking method [2]. But there usually comes a point where that process is no longer sufficient. From there, a tedious but very useful option is to mark up the possible values of each cell, and then spend a varying amount of time finding impossibilities of some of those possible by picturing future scenarios, similar to chess [2]. This is where the average human brain has difficulties, and where a computer performs very well.

2.2 Backtracking Algorithm

A backtracking algorithm is the formal name of the process described above [2]. In its most naïve form, it can be described as follows:

Naïve Backtracking Algorithm for a 9×9 Sudoku

1. Go through each cell (in a chosen order).
2. In the current cell, enumerate from 1 to 9, until:
    a. A value is found that is valid.
    b. 9 is reached, and no valid value was found.
3. In case of 2.a., go to the next cell and start again from Step 2.
4. In case of 2.b., go back to the previous cell and, following from Step 2., try the next value.

This algorithm has a few advantages. It is a rather simple algorithm to understand, for that reason it should also be relatively simple to implement in code. It offers the guarantee of finding a solution, if one exists, eventually. It can even solve an empty grid, though it will only find one solution out of the 6,670,903,752,021,072,936,960 possible solutions [1]. More on dealing with multiple solutions later. But being one of the simplest algorithms, there are also reasons why one would use another algorithm. It is a brute-force algorithm, it does not use any heuristics to find the solution faster. It simply iterates through all possible combinations of values, until it finds one that is valid based on the already filled cells. In general, its speed is dependent on the number of empty cells, and the number of possible values for each of those cells. Interestingly, because of it being so consistent in its method, one can develop a board that is specifically made to be "difficult" for this algorithm.

However, it is naïve. In other words, this version does not really take into account the constraints of the sudoku. So it will go through all possible combinations of values, even if it is obvious that some of them are not valid. In terms of code, this would mean running a check on the validity of the cell trial value each time we try one. To avoid this, we can use the constraints of the sudoku to mark up each cell with the possible values, as one would on paper, and then only try out those values.

Furthermore, as discussed above, the candidate checking method can be used to eliminate the "obvious" cells, that only have one possible value.

It so happens that by marking up the sudoku, we implicitely use the candidate checking method, as we can see which cells have only one possible value.

Thus, we can combine the candidate checking method and the backtracking algorithm to create a hopefully more efficient algorithm. This would then look like:

1. Markup the sudoku with the possible values of each cell.
2. If the cell has only one possible value, fill it in.

3. Go back to step 1 and repeat until no more cells can be filled in.
4. Start going through each cell (in a chosen order).
5. In the current cell, enumerate through the possible values for that cell, until:
    a. A value is found that is valid.
    b. all values have been tried and none were valid
6. In case of 5.a., go to the next cell and start again from 5
7. In case of 5.b., go back to the previous cell and try the next value.

2.3 Prototyping

Chapter 3

Development, Experimentation and Profiling

3.1 Development

3.1.1 Markup

The markup function is the first function that was developed.

3.1.2 Candidate Checking

To implement the candidate checking method, the markup function simply implemented in a loop whose condition was that the markup DataFrame no longer changed.

3.1.3 Backtracking

Another crack in the backtracking algorithm is for sudoku's that have multiple solutions. In that case, the solution it finds is simply a matter of what order the backtracking algorithm tries the values in. This means it will only find one solution, and only that one everytime. Making the assumption that the user may be simply interested in just solving the sudoku puzzle they have, a warning was added to the solver, to let the user know that the sudoku may have multiple solutions, and just one of them will be given.

3.2 Profiling

3.3 Validation, Unit Tests and CI set up

A check_sudoku function was written to check that the sudoku is valid. It checks that each row, column and box, and that there are no duplicates.

3.4 Packaging and Usability

3.5 Summary

Bibliography

[1] Michael Meerkamp. Counting sudokus. `https://pi.math.cornell.edu/~mec/Summer2009/meerkamp/Site/Counting_Sudokus_2.html`, 2009. Accessed: [11/2023].

[2] Michael Meerkamp. Solving any sudoku i. `https://pi.math.cornell.edu/~mec/Summer2009/meerkamp/Site/Solving_any_Sudoku_I.html`, 2009. Accessed: [11/2023].