Report for C1 Research Computing Coursework

CRSiD: tmb76

University of Cambridge

December 15, 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 [3]. 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 [3]. 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 [3]. 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 iterating through the above described steps consistently, one can develop a board that is specifically made to be "difficult" for this algorithm. In the context of general newspaper sudoku puzzles, it comes down to chance whether the algorithm will be fast or slow. Moreover, it is naïve. This version will test all values from 1 to 9, even when one case may only have 2 possible values. The complexity of the algorithm is then $\mathcal{O}(9^{N_{emptycells}})$, where $N_{emptycells}$ is the number of empty cells in the sudoku. This is a very large number, and so the algorithm is very slow for sudokus with many empty cells. So rather than wait until it is iterating through the cells, the algorithm could take note of what values are possible for each cell, and then only iterate through those values. This only means doing the validity assessment before the backtracking algorithm, rather than during it. But this means reducing the number of iterations, and so reducing the time it takes to find a solution.

2.3 Modified Backtracking Algorithm

Bringing back our attention to the candidate checking method, we know that we can identify “obvious” values for cells, and fill them in. Noting that to do so, we have to mark up the possible values for each cell. We can therefore use a modified backtracking algorithm that uses the candidate checking method prior to backtracking, to reduce the overall complexity of the algorithm (Maximum $\mathcal{O}(N_v^{N_{emptycells}})$, where N_v is the number of possible value for each cell, and $N_v \leq 9$).

Modified Backtracking Algorithm with candidate checking for a 9×9 Sudoku

1. Markup the sudoku with the possible values for 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, i.e. the markup does not change.
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 Step 5.
7. In case of 5.b., go back to the previous cell and, following from Step 5., try the next value.

2.4 Prototyping

The first idea of what the solver's code would look like was something like this:

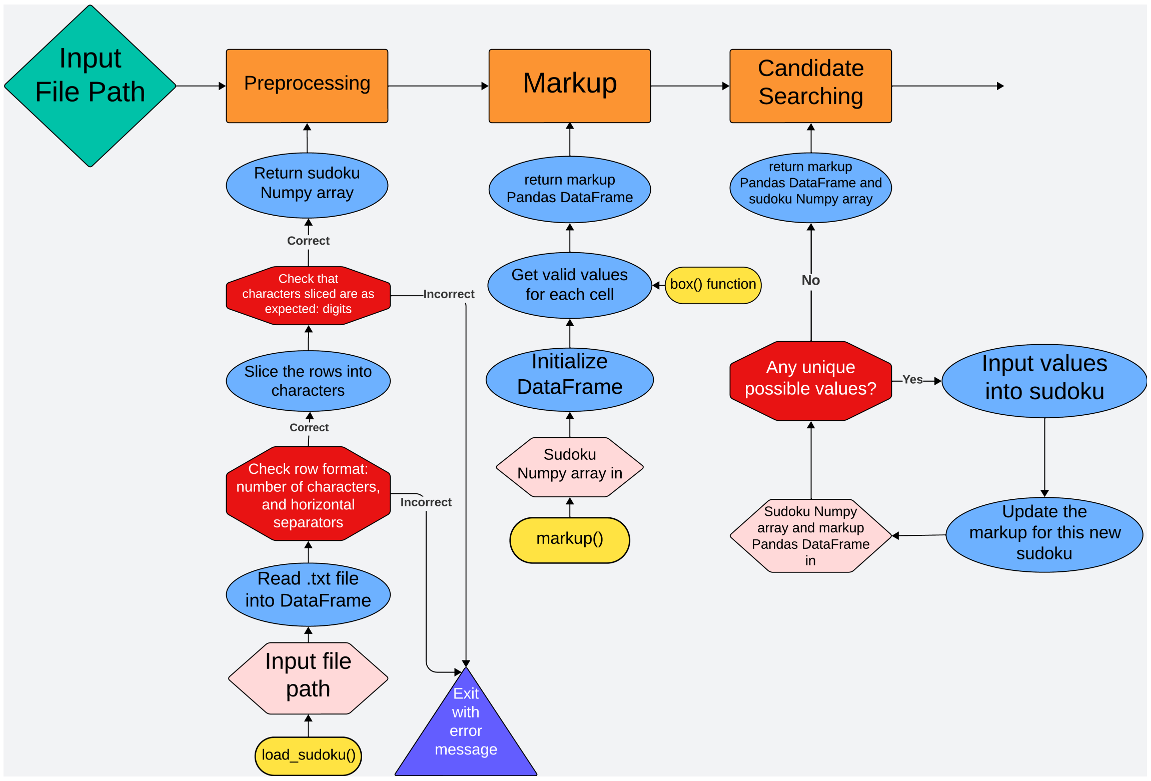
Input File Path
Preprocessing
Markup
Candidate Searching
Return sudoku Numpy array
Correct
Check that characters sliced are as expected: digits
Incorrect
Slice the rows into characters
Correct
Check row format: number of characters, and horizontal separators
Incorrect
Read .txt file into DataFrame
Input file path
load_sudoku()
Exit with error message
return markup Pandas DataFrame
Get valid values for each cell
box() function
Initialize DataFrame
Sudoku Numpy array in
markup()
return markup Pandas DataFrame and sudoku Numpy array
No
Any unique possible values?
Yes
Input values into sudoku
Update the markup for this new sudoku
Sudoku Numpy array and markup Pandas DataFrame in

Figure 2.1: First prototyping of the solver

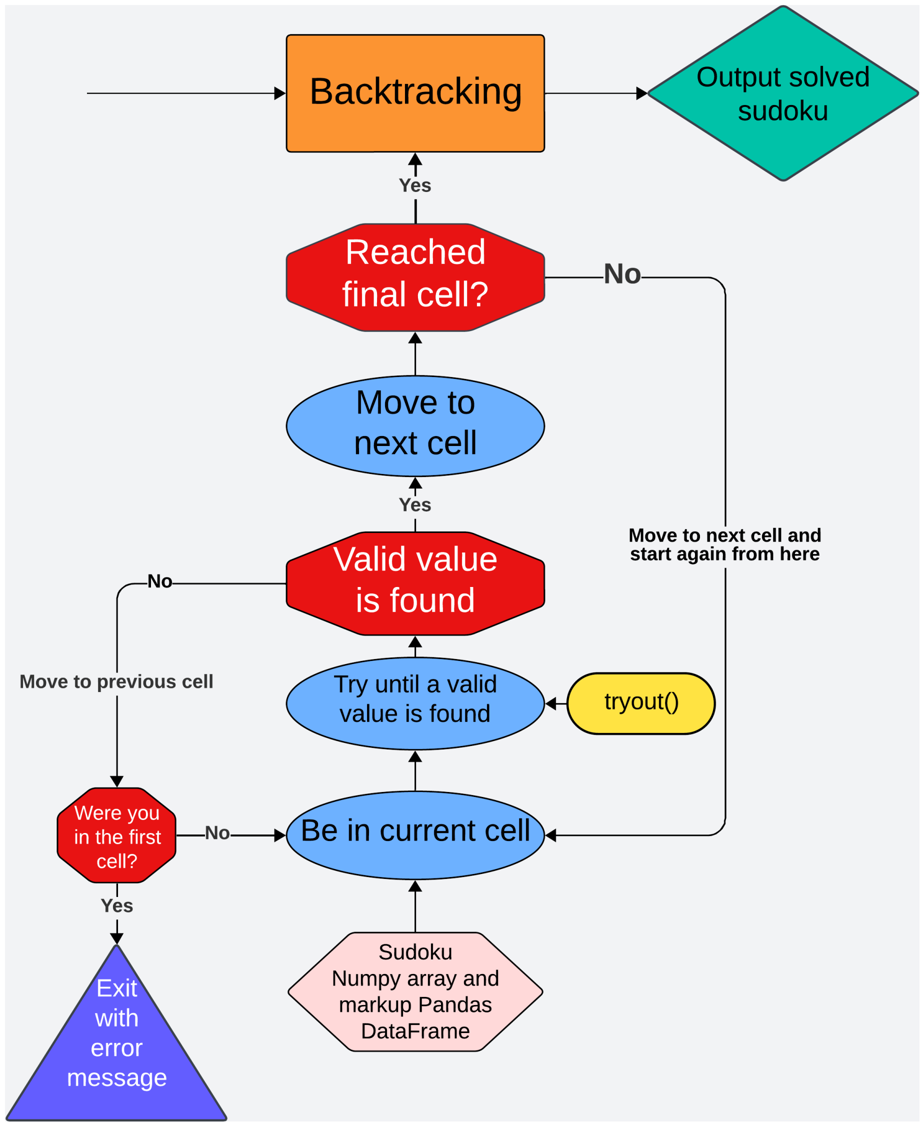

Figure 2.2: First prototyping of the solver (continued)

The solver would be runnable from the command line like so:

```
python solve_sudoku.py parameter.ini
```

Where the `parameter.ini` file contains groups of parameters, mainly the input file paths. Which parameters to use would be specified in the main `solve_sudoku.py` file. Bearing in mind that the sudokus must be in a `.txt` file, with the following format, with 0's representing empty cells:

```
020|000|000
000|083|400
090|000|000
---+---+---
000|800|000
600|009|000
000|000|093
---+---+---
000|100|000
000|054|000
070|000|000
```

Using the `sys` package, the file path would be passed to the `load_sudoku()` function, which would read the sudoku from the file path and drop out the separator rows and vertical lines, place the remaining 9 rows of 9 digits into a numpy array. Some checks would be done to ensure the sudoku `.txt` format is respected. Any infringement would simply make the program stop and print an appropriate error message. There is no point in continuing or trying to correct the mistakes as that could lead to solving a different sudoku than the one the solver's user was intending to solve. Following this preprocessing step, the markup of the sudoku would be created. The `markup()` function would simply look at each cell in the sudoku and, applying sudoku rules, determine what values are allowed. It is relatively easy to check the row and the column the cell is in. However, checking the 3×3 box the cell is in is a bit more complicated. For this, a (box()) function would be created to, given the cell row and column number, return the values of the 3×3 box the cell is in. The `markup()` function would then use this function to check the box the cell is in. Finally, we use a Pandas DataFrame to put the markup values in, as their number may vary for each cell, and a Pandas DataFrame allows for that, being structure like a dictionary, where a Numpy Array doesn't. The candidate searching part of the solver would consist of repeating the marking up of the sudoku, inputing of values in cells that only have one possible value, and so on until the markup no longer changes. Then comes the backtracking part of the solver. Following the steps set out in the modified backtracking algorithm, the solver would go through each cell, and try each possible value for that cell, based on the markup. If it finds a valid one, it goes to the next still empty cell and starts trying out values again. If it doesn't find one, it goes back to the previous cell and tries the next value. If it reaches the end of the possible values for the first cell, it means that the sudoku is unsolvable. The `tryout()` function would take in the cell location and the markup DataFrame, and return the first valid value it finds. Then it would be placed in a recursive loop that would go through each cell, and try out values until it finds a solution, or until it reaches the end of the possible values for the first cell. The until part would be handled by a `while` loop, that would check the cell location. From there the solved sudoku would outputed by printing it to the command line.

Chapter 3

Development, Experimentation and Profiling

3.1 Development and experimentation

3.1.1 Markup and Box functions

The `box()` function was written first to simply split the sudoku into its 9 boxes and return all of them. This would mean unnecessary memory use, so the function was changed to take in the row and column number of the box we wanted (e.g. the top right box would be box(1,3)), and return just that one. However, that meant figuring out which box the cell was in, from its coordinates, before calling the `box` function. Because that would be somewhat messy since the `box()` function would be called in multiple places, it was then changed to simply take in the cell coordinates and return the box the cell is in. In its first version, the `markup()` function would go through each unfilled cell and get the possible values for that cell, putting these in the markup dataframe, at the same coordinates. One problem, which came up when moving on to the candidate checking part of the solver, was that the markup dataframe contained NaN values, for the cells that were already filled. This would cause issues when comparing the markup dataframes when checking if the markup had changed. So, though it was not the most elegant fix, the `markup()` function was changed to fill in the already filled cells' locations in the markup with those values. In the context of the candidate checking loop, this meant that when updating the sudoku, the values of the cells that were already filled would be overwritten with the same values.

3.1.2 Candidate Checking

To implement the candidate checking method, the markup function simply implemented in a loop whose condition was planned to be that there no longer were any unique possible values in the markup. But with the `markup()` function changes described above, it was changed to there being no difference between the updated markup and the previous one. Now it was entirely possible that candidate checking was a sufficient method to solve the easier sudokus, in which case an already solved sudoku would be given to the backtracking algorithm. So a break condition was added to the code, to return the solved sudoku if the sudoku no longer had empty (= 0) cells. Further along the development process, once the solver was working as intended, the `markup()` function was changed to allow for empty sudokus, giving a warning that there may be multiple solutions when the number of clues at the start of the sudoku is less than 17 [2].

3.1.3 Backtracking

Now comes the main part of the solver, the backtracking algorithm. The first attempt consisted of having an embedded `while` and `for` loop in which the `tryout()` function would be used. However, it quickly became obvious this would either need a very convoluted series of loops or simply not be possible. The main issue was how one would be able to go back to the previous iteration of the loop. More precisely, while there is a `continue` statement to skip to the next iteration, there isn't one to go back one [5]. And though this cited stackoverflow post did provide an approach to have an iterator that could reverse one step, an other issue was the ability to keep track of the values already tried in the previous cell [5]. This would maybe mean having a list to which tried values get added, but this method could lead to code that was hard to follow. Thus, the method settled upon was to use a recursive function structure. The point of a recursive function is that it calls itself within its definition:

```python
def recursive_function(args_1):
  if base_case:
    return something
  else:
    # do something
    recursive_function(args_2)
  # do something
  return something else
```

In doing so, the function is called multiple times until a base case condition is fulfilled, a maximum number of recursive depth is reached [4], or something else depending on what the context and goal of the function is, which stops the recursion. The main benefit, apart from tidiness is that everything outside the function call is kept track of, as expected for the first call but that also means at different recursion levels.

For example, the recursive function above could be trying out values for the sudoku. Let's say the first value was tried for `recursive_function(args_1)`, where `args_1` would be the first cell, the function would then move on to the line where it calls `recursive_function(args_2)`. Now let's say `recursive_function(args_2)` has tried every possible option it had to try, which could mean having gone down multiple recursive levels, one would exit all recursive levels up to recursive_function(args_1) and be brought back to excatly where they were. The next value for the first cell could be tried and so on. Concisely, the recursive levels are akin to embedded for loops, but without the worry of having to set the number of loops. Backtracking for solving a sudoku falls under the category of exhaustive search where one tries all possible combinations. Following the Stanford lecture on Exhaustive search and backtracking[slides 1317] [6], this approach can be combined with returning booleans to handle whether to continue the recursion or end it. This gives the following structure:

Backtracking applied to sudoku

1. **If** there are no more decisions to make:

   a.**Base case** If the sudoku is solved, i.e. we've reached the end of the sudoku still-empty cells, end the recursion. Return True.

   b. **Base case** If the sudoku is not solved, i.e. we've gone through every possibility, but there are no more possible values for the last cell, end the recursion. Return False.

2. **Else:**

   a. **Choosing** Iterating over the still empty cells of the sudoku (markup cells with more than one possible value) using a recursive function. Choose a value from the markup cell's list, to which was applied the candidate checking method, to remove values that were no longer valid due to the filling up of the sudoku.

   b. **Exploring** Input that value in the sudoku, tracking our choice and move to the next cell, starting again at step 1.

   c. **Unchoosing** If going to the next cell, the valid values list is empty after running the validity check on it, reset the sudoku cell to 0 so that step 2.a. works properly, and go back to the previous cell and try the next value (i.e. go back to 2.).

3.2 Profiling

Having a working solver, profiling was done to identify where time was lost on operations. By running `cProfile` on our main `solve_sudoku.py` file, we got the following output:

Chapter 4

Validation, Unit Tests and CI set up

4.1 Validation

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.

4.2 Unit Tests

4.3 CI set up

Chapter 5

Packaging and Usability

5.1 Packaging

5.2 Usability

Chapter 6

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. How many clues are needed to solve a sudoku? https://pi.math.cornell.edu/~mec/Summer2009/meerkamp/Site/How_many_clues.html, 2009. Accessed: [11/2023].

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

[4] Stack Overflow. What is the maximum recursion depth and how to increase it? https://stackoverflow.com/questions/3323001/what-is-the-maximum-recursion-depth-and-how-to-increase-it/3323013#3323013, 2010. Accessed: [12/2023].

[5] Stack Overflow. Is there any way to go back a step in a python for loop? https://stackoverflow.com/questions/55380989/is-there-any-way-to-go-back-a-step-in-a-python-for-loop, 2019. Accessed: [12/2023].

[6] Stanford University. Lecture 11: Backtracking. https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1188/lectures/Lecture11/Lecture11.pdf, 2018. Accessed: [11/2023].