Report for C1 Research Computing Coursework

CRSiD: tmb76

University of Cambridge

December 16, 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 how one can use it.

Chapter 2

Solving Algorithm and Prototyping

2.1 Solving a Sudoku Puzzle

When solving a sudoku using brain-power, one has multiple techniques they can use. Most simple is to go through each cell, and using the sudoku constraints, eliminate impossible value, and hopefully be left with one. Relatively easy sudokus can be solved in part using this method. This can be referred to as the candidate-checking method [4]. However, there usually comes a point where that process is no longer sufficient. From there, a good 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 [4].

2.2 Backtracking Algorithm

A backtracking algorithm is the formal name of the process described above [4]. 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.

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 6×10^{21} possible solutions [2]. However it is a brute-force algorithm, it simply iterates through all possible combinations of values, until it finds one that is valid for the sudoku we have. On average, its speed is dependent on the number of empty cells, and the number of possible values for each of those cells. But it may happen, voluntarily or not, that a sudoku is particularly slow to solve, due to the placement of the clues. Additionaly, this version, being naïve, will test all values from 1 to 9. The complexity of the algorithm is then at most $\mathcal{O}(9^{N_{emptycells}})$, where $N_{emptycells}$ is the number of empty cells in the sudoku.

2.3 Modified Backtracking Algorithm

Thus, the algorithm could take note of what values are possible for each cell, ahead of iterating through the sudoku. This only means doing the validity assessment before the backtracking algorithm, rather than during it. But it should mean reducing the number of operations overall. 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 combines the both approahces, reducing the overall complexity of the algorithm to a maximum of $\mathcal{O}(N_v^{N_{emptycells}})$, 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:

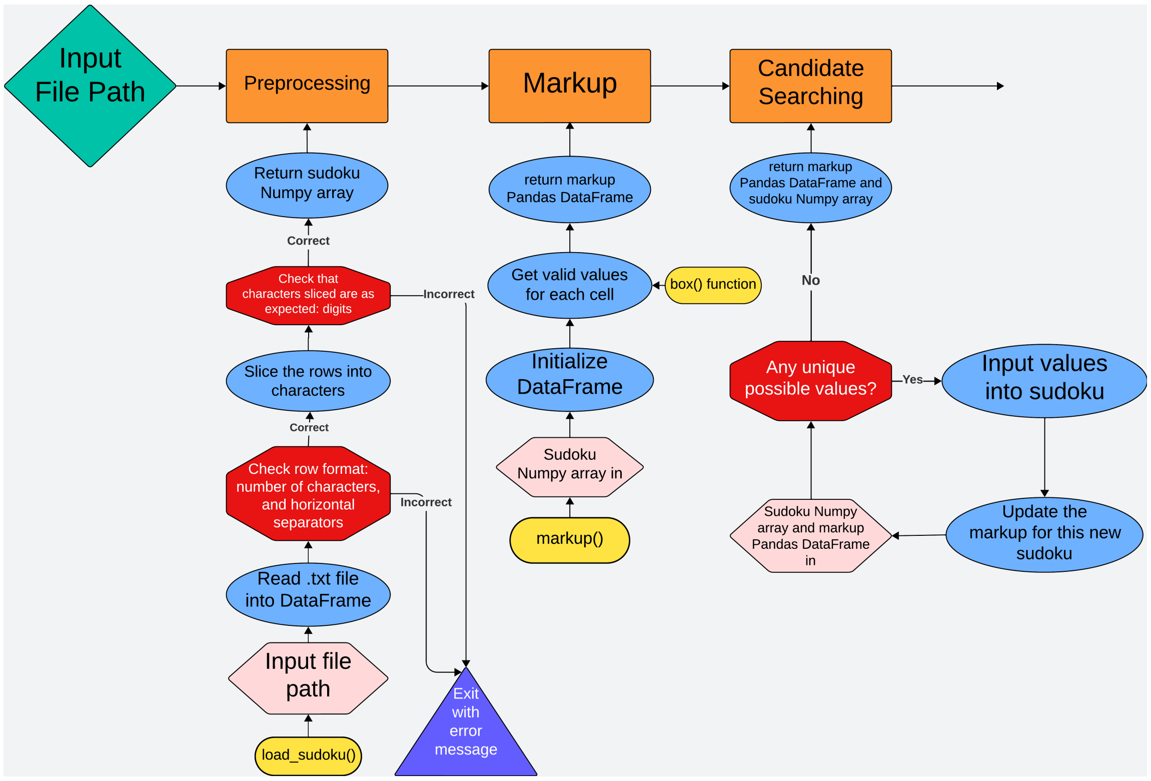
Input File Path
Preprocessing
Markup
Candidate Searching
Return sudoku Numpy array
Correct
Check that characters sliced are as expected: digits
Incorrect
Slice the rows into characters
Correct
Check row format: number of characters, and horizontal separators
Incorrect
Read .txt file into DataFrame
Input 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

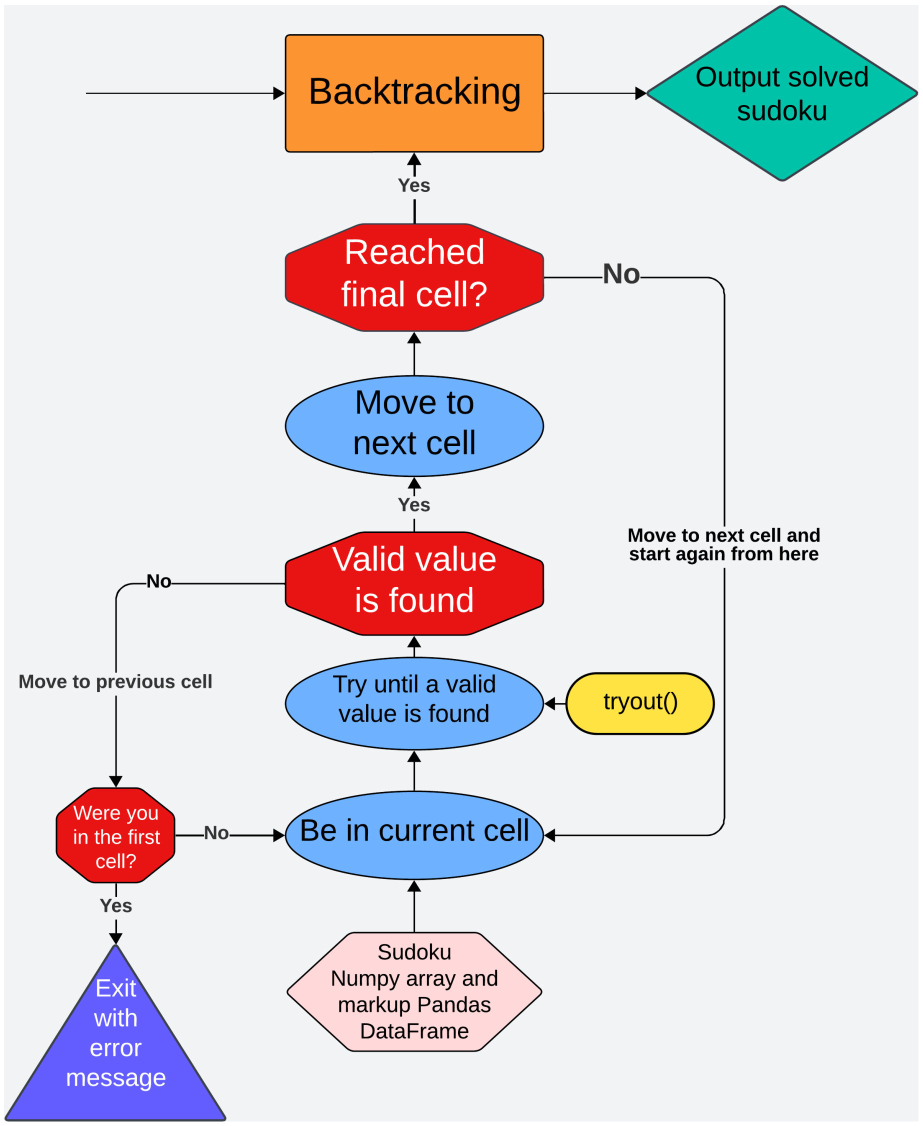

Figure 2.2: First prototyping of the solver (continued)

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

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

Where the `parameter.ini` file would contain groups of paths to sudokus. Which parameters to use would be specified in the main `solve_sudoku.py` file. 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
```

Listing 2.1: Sudoku input format example

The `load_sudoku()` function would read the sudoku from the file path and drop out the separator rows and vertical lines, and place the remaining rows of digits into a numpy array. Some checks would be done to ensure the sudoku `.txt` format is respected. Otherwise, the program will stop and print an appropriate error message, as trying a potential fix may mean solving a different sudoku than the one intended to be solved. Following this preprocessing, the `markup()` function would look at each cell in the sudoku and, applying constraints, determine valid values. To check the 3×3 box the cell is in, there would be a (box()) function to, given a sudoku, return the 9 3×3 boxes of the sudoku. Finally, the markup values would be put in a Pandas DataFrame, as it allows for elements of different sizes. The candidate searching would consist of inputing values in cells with unique possible values in the sudoku and updating the markup, until there are no more unique possible values. 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 through the possible values. If it finds a valid one, it sets that value in the sudoku, and goes to the next still empty cell and starts trying out values again. If it doesn't find one there, 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, the sudoku is thus unsolvable. The `tryout()` function would take in the cell location and the markup DataFrame, and return the first valid value it finds. It would be placed within 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. This would be handled by a `while` loop on the cell location. Finally, the solved sudoku would be printed out 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. To avoid unnecessary memore use, the function was changed to take in the row and column number of the box wanted (e.g. the top right box would be box(1,3)), and return just that one. However, that meant finding out which box the cell was in, from its coordinates, before calling the `box` function. With the `box()` function being called many times, it was better to take in the actual cell coordinates and return the box that 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, was that the markup dataframe contained NaN values, for cells that were already filled. This could cause issues when comparing the markups. So, it was changed to input the cell's value in the dataframe. Suboptimally, updating the sudoku would mean re-inputing values that were already there.

3.1.2 Candidate Checking

To implement the candidate checking method, the markup function was put in a while 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. In the case of an easy sudoku, this could be sufficient to solve it. So a break condition was added to the code before the backtracking, to return the sudoku if solved. Later, 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 [3], rather than stopping the program.

3.1.3 Backtracking

The first attempt consisted of having an embedded `while` and `for` loop in which the `tryout()` function would be used. However, this would either need a complex set of loops, or somehow setting the number of embedded loops based on the number of empty cells. The main issue was how one would be able to go back to the previous iteration of the loop. Indeed, while there is a `continue` statement to skip to the next iteration, there isn't one to go back one [7]. 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 [7]. Maybe by having a list to which tried values get added, but this seemed hard to follow, codewise 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 [10]:

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

Listing 3.1: Recursive function structure

In doing so, the function is called multiple times, at multiple "recursive levels" until a base case condition is fulfilled, a maximum number of recursive depth is reached [9], or something else depending on what the context and goal of the function is, which stops the recursion. The main benefit, is that recursive levels are akin to embedded for loops, but without the worry of having to set the number of loops, so it is much neater when coded. 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] [12], 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.**Exhausted** 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.).

One issue with backtracking is it generally depends on the number of empty cells, and the number of possible values for each of those cells, but also the placement and values of the filled cells. In most cases, the algorithm only tries a tiny fraction of the possible combinations. But an "anti-backtracking" sudoku can easily be created or randomly found. The following sudoku is one such example, taking 17843.329 seconds to solve [8]:

```
900|800|000
000|000|500
000|000|000
---+---+---
020|010|003
010|000|060
000|400|070
---+---+---
708|600|000
000|030|100
400|000|200
```

Listing 3.2: anti_backtracking.txt sudoku

But this is purely dependent on the order in which the algorithm iterates through the sudoku. Indeed, setting up a reverse order backtracking algorithm, this sudoku takes only 1.5 seconds to solve. This lead to also trying an “ordered” backtracking algorithm, filling the cells with the least number of possible values first [8]. To assess performance, a `perf_timer.py` file was created, which used a large number of sudokus [5], and timed how long it took to solve them. Though the “ordered” method was expected to perform better, all 3 performed equally well.

3.2 Profiling

Having a working solver, `cProfile` was ran on our main `solve_sudoku.py` file, we got the following output:

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.083    0.083 /src/solve_sudoku.py:25(solve_sudoku
     3    0.001    0.000    0.051    0.017 /src/solver_tools.py:54(markup)
   237    0.017    0.000    0.027    0.000 /src/solver_tools.py:94(<listcomp>)
318/1    0.001    0.000    0.019    0.019 /src/solver_tools.py:120(backtrack_alg)
   243    0.001    0.000    0.017    0.000 /pandas/core/series.py:1180(__setitem__)
   279    0.001    0.000    0.014    0.000 /pandas/core/series.py:1396(_maybe_update_cacher)
   317    0.011    0.000    0.014    0.000 /src/solver_tools.py:159(<listcomp>)
   243    0.001    0.000    0.011    0.000 /pandas/core/frame.py:4430(_maybe_cache_changed)
  3282    0.005    0.000    0.009    0.000 /src/preprocessing.py:89(box)
  3287    0.002    0.000    0.006    0.000 /numpy/core/fromnumeric.py:1768(ravel)
     3    0.002    0.001    0.006    0.002 /src/solver_tools.py:18(check_sudoku)
   279    0.000    0.000    0.006    0.000 /pandas/core/frame.py:3779(_ixs)
   643    0.002    0.000    0.006    0.000 /pandas/core/frame.py:3856(__getitem__)
   243    0.003    0.000    0.005    0.000 /pandas/core/internals/managers.py:1045(iset)
     6    0.000    0.000    0.005    0.001 /pandas/core/frame.py:668(__init__)
     5    0.000    0.000    0.004    0.001 /pandas/core/internals/construction.py:423(dict_to_mgr)
  3287    0.004    0.000    0.004    0.000 {built-in method numpy.asanyarray}
  3282    0.003    0.000    0.003    0.000 /src/preprocessing.py:128(<listcomp>)
   279    0.000    0.000    0.003    0.000 /pandas/core/frame.py:4387(_box_col_values)
   400    0.001    0.000    0.002    0.000 /pandas/core/series.py:1016(__getitem__)
  ....    .....    .....    .....    ..... mostly built-in functions of packages
```

Listing 3.3: First profiling output

A large amount of time was being spent on list comprehension lines used to get possible values, boxes, etc. This could be fixed by using sets. Sets are very similar to lists but are unordered and only contain unique values [1]. This means that originally, the code would search through the full row, column and box lists, with duplicate, and zeros. This meant python unnecessarily checked the same elements in those lists. Whereas when using the union of the rows, columns, and boxes sets, one set of at most 10 values would be checked. The time on these lines to be run could thus be reduced significantly. The profiling output after this change was:

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.047    0.047 /src/solve_sudoku.py:26(solve_sudoku)
        3    0.002    0.001    0.027    0.009 /src/solver_tools.py:56(markup)
      243    0.001    0.000    0.017    0.000 /pandas/core/series.py:1180(__setitem__)
      279    0.001    0.000    0.014    0.000 /pandas/core/series.py:1396(_maybe_update_cacher)
      243    0.001    0.000    0.011    0.000 /pandas/core/frame.py:4430(_maybe_cache_changed)
    318/1    0.003    0.000    0.010    0.010 /src/solver_tools.py:147(backtrack_alg)
      279    0.000    0.000    0.005    0.000 /pandas/core/frame.py:3779(_ixs)
      643    0.002    0.000    0.005    0.000 /pandas/core/frame.py:3856(__getitem__)
        3    0.002    0.001    0.005    0.002 /src/solver_tools.py:20(check_sudoku)
        6    0.000    0.000    0.005    0.001 /pandas/core/frame.py:668(__init__)
      243    0.003    0.000    0.005    0.000 /pandas/core/internals/managers.py:1045(iset)
        5    0.000    0.000    0.005    0.001 /pandas/core/internals/construction.py:423(
    dict_to_mgr)
```

```
      279     0.000     0.000     0.003     0.000 /pandas/core/frame.py:4387(_box_col_values)
      400     0.001     0.000     0.002     0.000 /pandas/core/series.py:1016(__getitem__)
     1283     0.002     0.000     0.002     0.000 /src/preprocessing.py:112(box)
     1288     0.001     0.000     0.002     0.000 /numpy/core/fromnumeric.py:1768(ravel)
      317     0.002     0.000     0.002     0.000 /src/solver_tools.py:191(<listcomp>)
5206/3341     0.001     0.000     0.002     0.000 {built-in method builtins.len}
        1     0.000     0.000     0.001     0.001 /pandas/core/frame.py:10039(map)
        1     0.000     0.000     0.001     0.001 /pandas/core/frame.py:9867(apply)
      243     0.000     0.000     0.001     0.000 /pandas/core/series.py:1270(_set_with_engine)
     ....     .....     .....     .....     ..... mostly built-in functions of packages
```

Listing 3.4: Profiling output after list comprehension improvements

It can be seen the cumulative times are much lower, and the time spent on some lines was reduced by more than a factor of 10. This was all running for a near-empty sudoku that would make large use of the backtracking algorithm to solve:

```
000|000|000
001|000|200
000|000|000
---+---+---
000|000|000
000|000|000
000|000|000
---+---+---
000|000|000
000|000|000
000|000|000
```

Listing 3.5: sudoku_near_empty.txt

Timing the improvement, the run time was 0.049503 seconds, and 0.020695 seconds after the improvement. This was only timing the line where the `solve_sudoku()` function is called. One could see, however, that a large portion of the time is taken by pandas DataFrame functions. Thus one could ask if the candidate checking method was actually slowing down the solver. Testing this using an easy sudoku which can be solved entirely by the candidate checking method:

```
002|560|470
058|403|000
004|020|008
---+---+---
781|000|040
409|100|726
006|047|800
---+---+---
007|006|013
005|030|007
060|709|200
```

Listing 3.6: sudoku_easy.txt

The following profiling output was obtained:

```
           ncalls   tottime   percall   cumtime   percall  filename:lineno(function)
```

```
        1    0.001    0.001    0.161    0.161 /src/solve_sudoku.py:26(solve_sudoku)
       18    0.005    0.000    0.138    0.008 /src/solver_tools.py:56(markup)
     1458    0.004    0.000    0.098    0.000 /pandas/core/series.py:1180(__setitem__)
     1629    0.005    0.000    0.078    0.000 /pandas/core/series.py:1396(_maybe_update_cacher)
     1458    0.004    0.000    0.063    0.000 /pandas/core/frame.py:4430(_maybe_cache_changed)
     1620    0.003    0.000    0.031    0.000 /pandas/core/frame.py:3779(_ixs)
     1458    0.013    0.000    0.026    0.000 /pandas/core/internals/managers.py:1045(iset)
     2728    0.008    0.000    0.023    0.000 /pandas/core/frame.py:3856(__getitem__)
     1620    0.002    0.000    0.017    0.000 /pandas/core/frame.py:4387(_box_col_values)
       19    0.000    0.000    0.016    0.001 /pandas/core/frame.py:668(__init__)
       19    0.001    0.000    0.015    0.001 /pandas/core/internals/construction.py:423(
    dict_to_mgr)
26809/15959    0.005    0.000    0.009    0.000 {built-in method builtins.len}
     1620    0.001    0.000    0.008    0.000 /pandas/core/frame.py:656(
    _constructor_sliced_from_mgr)
     1458    0.001    0.000    0.008    0.000 /pandas/core/series.py:1270(_set_with_engine)
     1620    0.005    0.000    0.007    0.000 /pandas/core/internals/managers.py:991(iget)
     1458    0.002    0.000    0.007    0.000 /pandas/core/series.py:1385(
    _check_is_chained_assignment_possible)
     1270    0.002    0.000    0.007    0.000 /pandas/core/series.py:1016(__getitem__)
     2728    0.002    0.000    0.006    0.000 /pandas/core/frame.py:4405(_get_item_cache)
     1458    0.001    0.000    0.006    0.000 /pandas/core/internals/managers.py:1977(
    setitem_inplace)
     1639    0.004    0.000    0.005    0.000 /pandas/core/generic.py:6147(__finalize__)
     5456    0.004    0.000    0.005    0.000 /pandas/core/indexing.py:2678(
    check_dict_or_set_indexers)
47346/47327    0.004    0.000    0.005    0.000 {built-in method builtins.isinstance}
     4186    0.003    0.000    0.005    0.000 /pandas/core/indexes/range.py:394(__contains__)
        3    0.002    0.001    0.005    0.002 /src/solver_tools.py:20(check_sudoku)
     ....    .....    .....    .....    ..... mostly built-in functions of packages
```

Listing 3.7: Profiling output for easy sudoku

Running the solver for this sudoku, it found a solution in 0.05651 seconds but did so much quicker in 0.00988 seconds when only getting the markup file once without updating it. Unfortunately, this meant that conducting a candidate check to get rid of obvious values was actually not reducing the solving time overall. Using the `perf_timer.py` file described in section 3.1.3., a comparison was made between average runtime for many sudokus, with and without the candidate checking method. The results showed almost a factor of 3 decrease in runtime when using backtracking only and marking up the sudoku just once (Backtracking only: 0.0054 compared to 0.0167 seconds).

3.2.1 Future improvements

In the interest of approaching sudokus with mulitple solutions, one backtracking order could be set up to shuffle the backtracking cells using `numpy.random.suffle()` on the backtrack cells list, this way obtaining many different solutions. Though it would run the risk of being slow, as we know the order of the cells has a large impact on the runtime.

Chapter 4

Validation, Unit Tests and CI set up

4.1 Validation and Unit Tests

Most sudokus in the offered `sudokus/` directory were either obtained from the sudoku app from sudoku.com [11] by EasyBrain. Thus, the solutions given by the solver were checked using these. More importantly, especially the case of multiple solutions, a `check_sudoku()` function was written to check that the sudoku is valid. This means checking there are no duplicates in the rows, columns and boxes. It is used at different points in the solver, checking the sudoku is valid after loading, after candidate checking, and after backtracking. It is also useful for running the `perf_timer.py` file, as the solved sudokus are not returned so it is worth checking that the solver did indeed find a valid solution. It takes an argument `final_check` which, if set to False, will mean the check will ignore 0 values, as they are expected in the sudoku when it is not yet solved. Multiple error traps were set in the functions and main code. They include checks to ensure the format of the input sudoku is correct, and multiple checks to ensure the sudoku is valid at different points in the solver, and that it is not unsolvable. If any of these checks fail, the solver will stop and print an appropriate error message, with a traceback to the line where the error occured. Unit tests were also set up. They are mostly straightforward, and they check the functions return the expected output for different inputs. Either they check that for a given correct input, the output is the correct one. Or, for a given "bad" input, they check that the error traps function as expected and that an error is indeed raised. This was done using the `unittest` package.

4.2 CI set up

Continuous Integration of this project was set up using a `pre-commit-config.yaml` file. This enables to have a few automated processes that run checks on the code before every commit:

1. `check-yaml` Checks that the `.yaml` files were not changed. This includes the `pre-commit-config.yaml` file itself.

2. `end-of-file-fixer`, `trailing-whitespace`, `mixed-line-ending` Checks the end of the files are correct. This is to ensure that the files end with a new line, and that the line endings are consistent.

3. `debug-statements` Checks that there are no debugging statements left in the code. This is to ensure that the code is clean and ready to be run.

4. `black` Checks the format of the code, and reformats it if necessary. It is there to minimize the number of errors the next hook could find.

5. `flake8` Checks the code for PEP8 compliance. It is there to ensure the code is clean and readable, following a set of standards. E.g. line length cannot exceed 79 characters (due to most code editors having set window widths [6]).

6. `pytest` Runs all the files that fall under a certain naming convention (Here: `t̂est//test_.py`, i.e. all files in the `test/` directory that start with `test_` and end with `.py`, to avoid an error with pytest catchin the `__pycache__` files in the `test/` directory).

The code was also written with Doxygen compatible documentation, and accordingly, there was a `Doxyfile` created to generate the documentation. There should be a `html/` folder in the `docs/` directory, but Doxygen can be run to get it if it is missing.

Chapter 5

Packaging and Usability

5.1 Packaging

The structure of the project is as follows:

```

sudoku_solver/
|-- docs/
|    |-- Doxyfile
|     -- html/
|-- report/
|    |-- refs.bib
|    |-- report.pdf
|     -- report.tex
|-- src/
|    |-- __init__.py
|    |-- preprocessing.py
|    |-- solver_tools.py
|     -- solve_sudoku.py
|-- profiling/
|    |-- perf_timer.py
|    |-- profiling.py
|     -- profile.txt
|-- test/
|    |-- __init__.py
|    |-- test_preproc.py
|    |-- test_markup.py
|    |-- test_backtrack.py
|     -- test_check_sudoku.py
|-- sudokus/
|    |-- anti_backtracking.txt
|    |-- bad_format_sudoku.txt
|    |-- dad_sudoku.txt
|    |-- sudoku_easy.txt
|    |-- sudoku_near_empty.txt
|    |-- sudoku_test.txt
|    |-- sudoku1.txt
|     -- sudoku2.txt
|-- .gitignore
|-- .pre-commit-config.yaml
|-- environment.yml
|-- Dockerfile
|-- LICENSE
 -- README.md
```

Listing 5.1: Directory Structure

5.2 Usability

The repository comes with a Dockerfile which the user is encouraged to use. One can build an image from it, using:

```
docker build -t c1_coursework .
```

Listing 5.2: How to build the Docker image

Once the image built, the container can be run using:

```
docker run --rm -ti C1_Coursework
```

Listing 5.3: How to run the Docker container

This will run a small bash terminal-like interface (-ti). The main solver file `solve_sudoku.py` can then be run from the command line using the following command:

```
python -m src.solve_sudoku path/to/sudoku.txt 'backtracking_type'
 backtracking_only
```

Listing 5.4: How to run the solver

The `-m` flag is used to run the file as a module, which is to deal with the relative imports in the code. The `path/to/sudoku.txt` is the path to the sudoku file, in the format specified in Listing 2.1. The `backtracking_type` argument can either be `'forward'`, `'backward'`, or `'ordered'`. The `backtracking_only` argument is a `Bool` value, and is used to set the solver to only use backtracking (`True`) or also use the candidate checking method (`False`). The solver will print the solved sudoku to the command line, and also write a `.txt` file with the same name as the input sudoku file, in a `sudoku_solution/` directory, which will get created if it doesn't exist yet.

For the `perf_timer.py` and `profiling.py` files, the user can run them from the command line using the following commands:

```
python -m profiling.perf_timer backtracking_only
python -m profiling.profiling path/to/sudoku.txt '
 backtracking_type' backtracking_only
```

Listing 5.5: How to run the profiling and timing files

For the `perf_timer.py` file, the sudokus.csv can be obtained from this kaggle website [5]. One can decide to only take a subset of the available sudokus. For the timing results in Chapter 3, around 2000 sudokus were used.

Chapter 6

Summary

Thus, our sudoku solver now works as intended. It can solve any sudoku, though give only 3 solutions to sudokus with a large number of solutions.

Bibliography

[1] Kolade Chris FreeCodeCamp. Python set vs list: When to use each. https://www.freecodecamp.org/news/python-set-vs-list/, 2023. Accessed: [12/2023].

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

[3] 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].

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

[5] Bryan Park. Sudoku dataset. https://www.kaggle.com/datasets/bryanpark/sudoku/data, Unknown. Accessed: [Current Date].

[6] Python. Pep 8 – style guide for python code. https://peps.python.org/pep-0008/, 2023. Accessed: [12/2023].

[7] mariogarcc 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].

[8] nubela Stack Overflow. Optimizing the backtracking algorithm solving sudoku. https://stackoverflow.com/questions/1518346/optimizing-the-backtracking-algorithm-solving-sudoku, 2009. Accessed: [12/2023].

[9] quantumsoup 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].

[10] Sebastian S Stack Overflow. Basics of recursion in python. https://stackoverflow.com/questions/30214531/basics-of-recursion-in-python, 2015. Accessed: [12/2023].

[11] Sudoku.com. Sudoku. https://sudoku.com, 2023. Accessed: [12/2023].

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