Sudoku Solving Algorithm

Luigi Challis

December 10, 2015

1 Introduction

Sudoku is a classic logic-based number placement puzzle with rules very easy to grasp. The appearance of the puzzle can have a seemingly complex appeal but for the ones who have read the rules, they know that there is only simple logical thinking behind the numbers.
The origin of the word, Sudoku, come from the Japanese, *Su-ji wa dokushin ni kagiru*, which means, "The numbers must be single". This can be seen as perhaps one of the fundamentals of understanding the rules.
A standard Sudoku puzzle is a 9x9 grid, consisting of 9 rows, 9 columns and 9 sub grids [see Figure 1]. These three categories are to be filled with the numbers (n = possible values) 1 to 9 where there cannot be any repeats of a number in a row, column or sub grid [see Figure 2].

$$n = 1, 2, 3, 4, 5, 6, 7, 8, 9$$

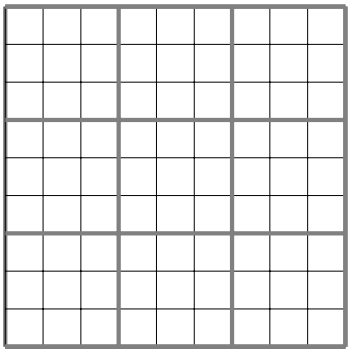

Figure 1: Completely empty 9x9 Sudoku *(left)*

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

Figure 2: Solved 9x9 Sudoku *(right)*

2 How to Solve a Sudoku via Coding

To approach writing the code for this project I decided to use the simplest of Sudoku solving techniques, the "Naked Single" [7], and the Brute Force method of coding [5][4] as any more advanced techniques would require more advanced coding which I am not yet capable of writing.

Quite simply, using the information provided about Sudoku in Section 1, we determine the values of an empty cell by observing the values of filled cells in the corresponding row, column and sub grid. Given that there is only one possible value that can exist in the empty cell then that must be the true value. When I

write this out in code I will use three different functions that check the rows, columns and sub grids of a cell. [3] The following two images, taken from the top three sub grids of an unsolved matrix, demonstrate the "Naked Single" Technique. The small faint numbers drawn in some of the empty cells are the possible values that the cell can have.

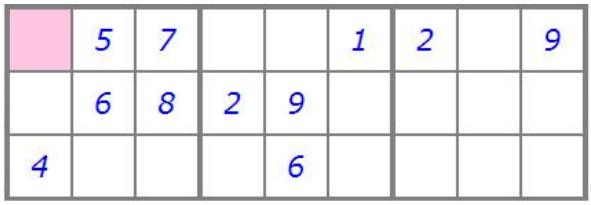

Figure 3: Without Candidates *(left)*

Figure 4: With Candidates *(right)*

[6]

However, this of course is not always necessarily the case. There would be most likely more than one candidate of the same value that appears in the corresponding row, column and/or sub grid to the empty cell - A candidate being a possible value. This gives us a problem.
I would have to create another function (4th) that collects the different possibilities of each empty cell. I would also need a 5th function to set a final 'true' value, if the cell has only one possibility. It would repeat this a number of times through the grid, filling up the empty cells which have only one candidate until the cells with more than one candidate are slowly reduced to a single possibility as all the other possibilities have been used up. This is how I will solve the Sudoku. I have removed some parts of the code irrelevant to my explanation, for instance, the different levels of difficulty and the solution. However, the entire code can easily be viewed at:
`https://cloud.sagemath.com/projects/5b4bbcc6-dda7-4ba6-b8d6-5f8ee8f0fdda/files/Sudoku%20Soving%20Algorithm.sagews`.

```
def getpossibilities():
# Function to obtain the different possibilities for each cell and hold them within a list.
    possibilities_total = []  # List to contain all possibilities of empty cells.
    for y in range(0, 9):
        for x in range(0,9):
            if(grid[y,x]==0):  #  If any cell is empty, then do the following...
                count = 0  # Set count to zero.
                val = []  # The possible values are to be contained within this list.
                for i in range(1,10):
                    if((checksubgrid(i,[y,x]) == True) and (checkcols(i,[y,x]) == True)
                    and (checkrows(i,[y,x]) == True)):
                        count += 1
                        # If there is no other of the same value in any other cell in the
                        corresponding row, column or subgrid then add 1 to count. i.e.
                        True for all.
                        val.append(i)  # Add the value to to val list if true for all.
                possibilities_total.append([y,x,val,count])
                # Add the list of a cell to possibilities_total list.

    return possibilities_total  # Print possibilities_total list
```

```
getpossibilities()
# This prints the list containing the empty cells with their coordinates, possible values
and number of values

for g in range(100): # Repeat 100 times. Number of around 100 achieved by trial and error.
    try:
        cells = getpossibilities()
        i = 0
        # Set i to be zero, where i is the list number. e.g. i = 5 would be list number 6.
        while(True):
            if(cells[i][3] == 1):  # If there is only one possibility of list i, then...
                grid[cells[i][0], cells[i][1]] = cells[i][2][0]
                # The cell in list i is equal to the first value of the possible values
                which would be the only value in that sublist.
            i += 1  # Add 1 to i to proceed with the second list in possibilities_total.
    except IndexError:  # When reached IndexError, repeat to cells = getpossibilities().
        cells = getpossibilities()

show(grid)  # Print the final solved Sudoku.
```

[1][2]

3 Conclusion

In this project I have created an original piece of code that can solve a range Sudoku puzzles, successfully. There is a limit of diculty to which this code can be applied. My code has the ability to solve Easy and Medium levels of diculty. However not "Trivial", as discussed in Section 2, I used the simplest of methods to create this algorithm. To solve a Trivial level the code would be shorter and operate faster, however, this is code that I am not capable of creating yet. Here are the results that I obtained with my code.

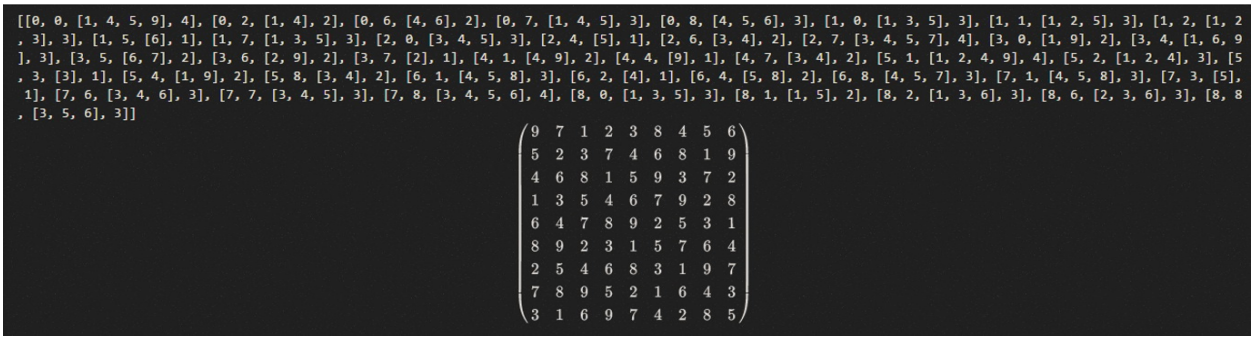

Figure 5: Results

The bottom section is the completed Sudoku grid and the section above is a list of all the empty cells registered at the start of the code, specifying their coordinates in the Sudoku (y,x), the dierent values that cell can have and the number of possible candidates. Below is the general formula which expresses the long list shown in Figure 5, where c is an empty cell and v is a value from 1 to 9.

$$All\ empty\ cells = [c_1 = [y, x, [v_1, v_2, v_3, ..., v_n], n], c_2, c_3, ..., c_n]$$

In Section 2, I discussed the two functions that I created, one that results in finding all the candidates possible for all empty cells, position [2], and the other function that repeats setting the first value in the candidate list, $[v_1, v_2, v_3, ..., v_n]$, as the true value if, and only if, there is only one possible value i.e. $[v]$. The above formula is what is used to do this along with the code to help achieve to the finished Sudoku.

References

[1] J. F. Crook. "a pencil-and-paper algorithm for solving sudoku puzzles". http://www.ams.org/notices/200904/tx090400460p.pdf.

[2] Vince Knight. "lab sheets". urlhttp://vknight.org/Computing$_f or_m athematics$/.

[3] Cornell Maths. "mathematics and sudokus: Solving algorithms (i)". http://www.math.cornell.edu/~mec/Summer2009/meerkamp/Site/Solving_any_Sudoku_I.html.

[4] n.p. "brute force". http://www.webopedia.com/TERM/B/brute_force.html.

[5] The Public. "more math into latex: A guide for documentation and presentation". http://www.minimalbeispiel.de/mini-en.html, 2008. [Last Assessed: 1 December 2015].

[6] Martin Scharrer. "putting two images next to each other". http://tex.stackexchange.com/questions/23835/putting-two-images-next-to-each-other-that-are-0-5-textwidth-wide.

[7] Sodoku Solutions. "simple solving tehcniques: Naked single". http://www.sudoku-solutions.com/index.php?page=solvingNakedSubsets#nakedSingle.