Solving Sudoku Programmatically

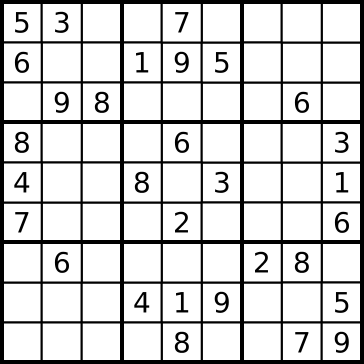
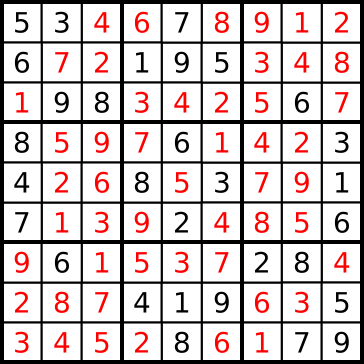
 Solving a Sudoku puzzle programmatically is a task that is both open ended and complex. Sudoku is a Japanese puzzle that usually consists of a 9 x 9 grid of squares divided into nine rows of nine squares, nine columns of nine squares, and nine 3 x 3 boxes, each with nine squares. Henceforth, these rows, columns, and boxes will all be referred to as groups, as the difference between them is not very important. Some, but not all, squares of an unsolved Sudoku puzzle are filled with the numbers 1 through 9. In order to solve a Sudoku problem, all 81 squares of the puzzle must be filled with a number 1 through 9 so that each group contains no repeated number.

Figure 2. Solved Sudoku Puzzle\*

Figure 1. Unsolved Sudoku Puzzle\*

In our Python code, a Sudoku puzzle will be represented by a string 81 characters long, where ‘.’s represent blank squares. For example, the unsolved Sudoku puzzle above would be represented by this string:’53..7....6..195....98....6.8...6...34..8.3.. 17...2...6.6....28....419..5....8..79’. Perhaps the most basic method of solving Sudoku is the recursive brute force method, which will also serve as the basis of our solver program. The recursive brute force method, bruteForce, accepts a Sudoku puzzle string as an argument and first iterates through each position of the Sudoku puzzle string. If a position is blank (represented by a ‘.’), then bruteForce guesses a value between 1 and 9 to place in that position and returns the modified string. If the puzzle is still valid, meaning that there are no duplicate values in any group, then the recursive bruteForce method keeps inserting values into the puzzle until the puzzle becomes either invalid or solved. If the puzzle becomes invalid, then bruteForce returns a blank string and tries other values until the puzzle is solved. If the puzzle is solved, then bruteForce returns the solved puzzle string.

As one can imagine, this recursive brute force method is ponderously slow. It can be sped up and made more efficient by adding several modifications. We will first create allGroups, a list of lists that contains the positions of each group, allSyms, a set with all integers between 1 and 9 inclusive, and cellNeighbors, a dictionary where the keys are positions in the puzzle string and the values are lists that contain all neighboring positions. Please note that allGroups and cellNeighbors contain values that are positions, not values; this is very important. While these three variables will not directly speed up our solver, they will help greatly. The first real speedup is achieved by making bruteForce only choose from values that would be valid if placed in that position instead of selecting any value from allSyms. This is done by generating a dictionary named possible, in which the keys are the positions of blank squares in the puzzle string and the values are the numbers that can be placed at the blank positions without invalidating the puzzle, every time bruteForce is called. possible can be easily generated by iterating through the puzzle string, using the index, i, of the string as possible’s key and appending all values that are not at the neighboring positions in the puzzle string in cellNeighbors[i].

def findPossible(puzzle):

allPossible = {i : allSyms - {puzzle[ps] for ps in cellNeighbors[i] if puzzle[ps] != '.'} for i in range(0, 81)}

return {i : allPossible[i] for i in allPossible if puzzle[i] not in allPossible[i]}

Selecting values using possible instead of any value in allSyms significantly reduces the number of recursive calls to bruteForce and speeds up the program. Furthermore, bruteForce can judiciously choose the empty position that has the highest probability of being guessed correctly. This is found by finding the position with the least number of possible values, as this position will take the least amount of guesses to find the correct value.

While these two simple speedups will significantly decrease the runtime of our solver, they are not enough for our purposes. We will implement two deductions using two well known Sudoku principles. The first principle that we will use is called the naked single, which states that if there is only one possible value that can go in a position, then that value must go in that position and can also be removed from all of the neighboring positions’ set of possibilities. While this seems extremely obvious, the removal of the value from neighboring positions’ possibilities is extremely useful. The second principle that we will implement is the naked subset principle. This principle states that if in a group there are *k* positions that have only the same *k* possible values, then those *k* possible values must be placed in one of those positions and can also be removed from each of the neighboring positions’ set of possibilities. We will implement these two deduction by creating a makeDeductions method that returns a tuple and is called at the very beginning of our bruteForce method. Instead of generating possible during each bruteForce call, we will generate possible at the beginning of makeDeductions, which will modify possible and our puzzle string using our deductions, and then return a tuple containing our puzzle string and possible, which will then be unpacked and used in the rest of our bruteForce method. Inside makeDeductions, we will effect our first principle, the naked single, by iterating through each pos in possible.

for pos in possible:

if puzzle[pos] == ".":

'''If only one value is not excluded from being in a box, then the value must go in that box'''

if len(possible[pos]) == 1:

for char in possible[pos]:

for check in cellNeighbors[pos]:

if check in possible:

if char in possible[check]:

possible[check].remove(char)

puzzle = puzzle[:pos]+ char + puzzle[i+pos:]

If the character in the puzzle at position pos, puzzle[pos], is equal to a ‘.’, meaning that it is a blank space, and there is only 1 possible value at position pos, meaning len(possible[pos]) is 1, then we store that value as char and insert it into the puzzle string at that position. We then iterate through each neighboring position in cellNeighbors[pos] and, if char is in a neighbor’s set of possibilities in possible, remove char from that set. The second deduction will be implemented by iterating through each group in allGroups.

for grp in allGroups:

subsets = {pos: possible[pos] for pos in grp if pos in possible}

array = list(subsets.values())

for pos in subsets:

if len(subsets[pos]) == array.count(subsets[pos]):

for other in subsets:

if subsets[pos] != subsets[other]:

for char in subsets[pos]:

if char in possible[other]:

possible[other].remove(char)

A dictionary named subsets will be made, where the keys are the positions in group that are unfilled in the puzzle and the values are the sets of possibilities for those unfilled positions. A list named possibilities will also be created, which contains only the values of subsets, this can be compactly represented by list(subsets.values()). We will then iterate through each pos in subsets and if the length of the set of possibilities at pos, len(subsets[pos]), is equal to the number of occurrences of the set of possibilities at pos, array.count(subsets[pos]), then we remove all characters in the set of possibilities at pos from the sets of possibilities in possible of all other positions in group that do not share the exact same possibilities as pos. Once this is implemented, another quick speed up can be added to the loop in which the first deduction is contained. If there is an empty position in the puzzle string which has 0 possibilities, then the puzzle string is invalid and makeDeductions should return an empty string. bruteForce should then be modified to also return a blank string if makeDeductions returns one.

def bruteForce(puzzle):

global guesses, allGroups, cellNeighbors

guesses += 1

tpl = makeDeductions(puzzle)

if tpl == "":

return ""

puzzle, possible = tpl

pos = puzzle.find('.')

if pos < 0:

return puzzle

min = 10

for i in possible:

if puzzle[i] == ".":

s = len(possible[i])

if s < min:

min = s

pos = i

for char in possible[pos]:

bf = bruteForce(puzzle[:pos] + char + puzzle[pos+1:])

if bf != "":

return bf

return ""

Once all of these speed ups are put into action, our Sudoku solver should be reasonably fast. Our solver has evolved from being a dumb brute force algorithm into a fast and fairly elegant, deduction-powered Sudoku solving program.