Solving Sudoku Programmatically

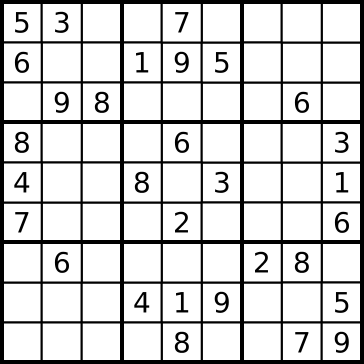
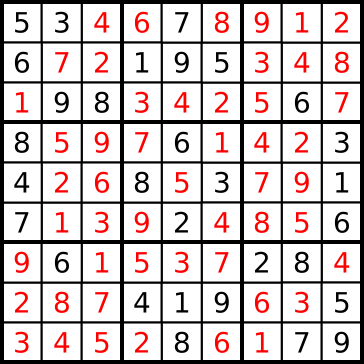
 Solving a Sudoku puzzle programmatically is a task that is both open ended and complex. Sudoku is a Japanese puzzle that usually consists of a 9 x 9 grid of squares divided into nine rows of nine squares, nine columns of nine squares, and nine 3 x 3 boxes, each with nine squares. Henceforth, these rows, 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 all groups, allSyms, a set with all integers between 1 and 9 inclusive, and cellNeighbors, a dictionary where the keys are positions in the 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. Whiles these three variables will not directly speed up our solver, they will soon 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 of the string as possible’s key and appending all values that are not in neighboring positions (cellNeighbors can be easily used to do this). Selecting values using possible instead of any value between 1 and 9 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 implications of this principle are 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.