# Sudoku solver and generator

CMPE 130: Advanced Algorithm Design


Ryota Suzuki 011115039

Chih-yu Huang 012475827

Wenyi Cai 011232000

**Problem statement and motivation**

Our group decided to make a sudoku solver since its simplicity of the rules and the complexity of the application interested us. The rule of the game is very simple. The player is given a 9x9 board with some initial values. Then they have to fill the empty cells with numbers one through nine. The catch is that each row, column, and 3x3 subgrid can only contain one of each number. The problem we wanted to solve was given a sudoku board, what the most efficient method of solving is. We also wanted to make a sudoku board generator while we were at it.

**Solution Description**

Our solution to this problem was to create a program that goes through each empty cell and enter a number that follows the horizontal, vertical, and subgrid rule. All the user has to do is to enter their sudoku board into the console and our program will find a solution to it. To do this, we made a function called "backtracking(int n)", which takes the position as a parameter. For our sudoku board, we used a 2D array to store our numbers. We start at 0 and will keep incrementing it by one as we recurse through the array. We only need one parameter since we can get the y-axis when we divide the parameter by nine and the x-axis when we module the parameter by nine. If the cell is not empty, it will simply recurse to the next position. If it is empty, it will go through numbers one to nine in order and see if it satisfies the three rules. If the number is valid, it will fill the current cell with that number and recurse to the next position. If the cell tries all nine numbers, it means that there is a mistake somewhere else on the board, so it will go back to the previously filled cell and change its value to a different valid number. This process of going back is called backtracking. When position hits 81, then the program has successfully filled all cells (found the solution) and the program will end.

We also made a sudoku generator, which follows a similar algorithm as the solver. To make the board random as possible, we first made three 3x3 subgrid diagonals from each other. This is because we can randomly place numbers in them without having to worry about the horizontal and vertical rules. Then, we used the same fillRest(int n) function from the solver and filled in the rest. However, since our backtrack(int n) function tested each cell with numbers one

to nine in order, there was a slight pattern where the subgrid had smaller numbers at the beginning of it. To make this random as possible, we made a random array that contained one of each number from one to nine. Then tested each cell in random order, giving us a random sudoku board. After making a full 9x9 board, we deleted numbers from it to make the sudoku board solvable. Our goal was to implement a function that created a solvable board with a unique solution. However, this was complicated for a couple of reasons. The method to accomplish this is to delete one number at a time, then solve it. If the board has only one answer, then we take another number and repeat until we hit the desired number of empty cells. The first problem is that this method is very time complex, as it needs to solve the board each time a number is removed. In addition, it is also space complex since we need three copies of the board: the filled board, solving board for the program, and a solvable board to keep track of which cells are empty. However, the major issue is that after removing a certain amount of numbers, the next cell we delete will cause the board to have multiple solutions. Here, we tried using the backtracking method, which after five attempts of finding a unique solvable board will undo the deletion of the last cell and find another cell to delete. However, we were not successful to apply this function to our program.

**Complexity analyze**

For sudoku solver, we are using backtracking to find the answer. If we are not using backtracking, just use brute force, time complexity will be O(N^(N^2)). However, we use backtracking, which potentially allows us not to go over every possible combination. But in the worst-case scenario, we will still iterate every possible combination. So the time complexity remains O(N^(N^2)), where N is the numbers of columns or rows of the square sudoku puzzle.

For playable Sudoku, when user input, we only compare the user input with the specific element in answer board, the time complexity would be O(1).

**Tools used for programming**

The library we used was stdlib.h for the random number generator and time.h to randomize the random number generator. We also used iostream.h to get the user input and print out the board.  We only use Eclipse and VS code for coding.

**Conclusion**

In conclusion, we successfully built a playable Sudoku. In our program, we will create an empty 9x9 board first, then we will fill in the board with backtracking and randomize number, which will make sure we will get different Sudoku puzzles every time when the user is playing. After the board is full, we will start deleting some numbers in the puzzle. We tested our program and in each run, the number of deleted numbers is between 30 to 50 which will make the board become playable for humans.

**Contribution**

Chih-yu Huang: Responsible for finish the check Sudoku rule and backtracking function of finding the solution

Wenyi Cai: Responsible for finish the playable UI

Ryota Suzuki: Responsible for implementing the skeleton of the project, generate randomize Sudoku board.

**Link to code:**

**https://drive.google.com/open?id=1f0xK2DHuiZR_RoX5PsQzUBu5RsqpvLZs**