Improved Sudoku Solver: Using Random Ordering

Abhi Uday Pandey

What is the sudoku Puzzle

The objective is to fill a 9×9 grid with digits so that
1) Each column,
2) Each row,
3) Each 3×3 grid
contain digits 1-9 exactly once

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

Abstract

Traditional Sudoku is solved by a computer using the **backtracking algorithm**. In this algorithm every digit from 1-9 is tried at the blank spaces until a solution is reached. Since the digits are tried sequentially there is high chance that it will be very slow in cases where the blank spaces contain the higher order digits such as 5,6,7,8,9. This is because it will take at least 5 tries to get it correct resulting in higher number of function calls.
Also, we can design a sudoku deliberately in way to slow down this backtracking algorithm because of running the algorithm in a sequential manner
Here I have proposed a solution to counter the above problem and to avoid the bias created by the sequential order and which no sudoku can predictively slow down.

Introduction

Rather than trying the digits in sequential order I have tried the digits in a random order. This helps in cases where the actual digit that must occupy its place is tried first and hence the solution is generated faster
To do the above we have break the problem into 2 steps

1) Generate a random order where the digits don't repeat and cause infinite loops
2) Try this random order without increasing the function calls

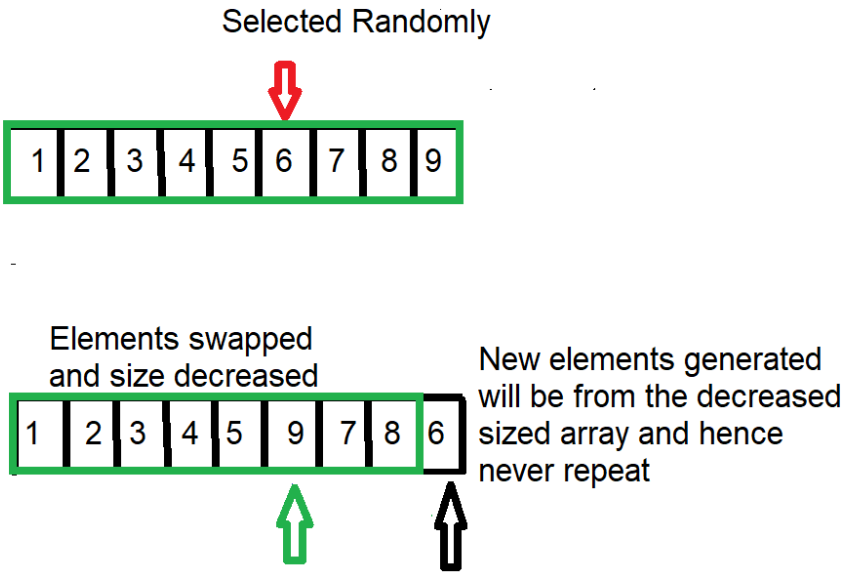
Methodology

To generate the random order we must take care that we don't take too much time otherwise our optimization won't be fruitful. Also, our algorithm must not produce repeated elements.

For doing this the following algorithm is used

1. Get a random number between 0 and size of array as index
2. Use this number as index and get the value there
3. Lastly swap the last number and the number you obtained and reduce the size by one

This Algorithm can be better understood with the following example
Imagine a sack full of 1 to 9 digits now take 1 digit with eyes closed insert into a stack. Now since the sack contains 1 less digit that will never be repeated and the next time, we take out digit it will be different from the one obtained earlier.



Comparison and analysis

In a sudoku there should minimum 17 filled values to maintain the uniqueness of the solution
Assumptions

1) Consider 5 empty cells in each row
2) Max Number of empty Cells as 45
3) Number of empty Cells at average 22
4) Hardest sudoku's will have maximum number of empty cells

Here all times are shown for traditional sudoku puzzle hence having a constant dimension it will have constant time

Traditional Backtracking	Our Approach
Best Case-O((1+2+3+4+5)*9)+function calls	Best Case-O(45)
Average Case-O(9^22)	Average Case-O(9^22)
Worst Case-O(9^45)	Worst Case-O(9^45)

Although all the mentioned times here are constant these values are very large and hence Our new Approach improves the performance
Of the traditional approach and this approach cannot be slowed down purposely

The explanation for the calculation in the table can be find in the below section

Reasons for better performance

Best case

Traditional case- The best case for traditional approach occurs when 1,2,3,4,5 is missing from all rows in this order hence total tries for 1->1, for 2->2 and so hence each row is calculated in 15 tries total and total becomes 15*9=145+ the amount of time taken in making function calls

Our approach- Here there could be a case where all the randomized orders have correct places in the first try and hence it fills without any further function calls

Average and worst case are same for both since there cannot be specific calculations for those. However, at average this will also be faster since due to the property of random numbers (That they don't favor any particular case

Data collected on implementation (language used C++)

Below table showcase the improvement of the newer approach. Although this improvement doesn't seem much this will much higher with higher dimensional versions of sudokus and other similar puzzles for example 25*25 sudokus

Number of sudokus tested	Original version		Improved version	
	Time (micro seconds)	Number of Function calls	Time (micro seconds)	Number of Function calls
10	-	8787	-	6947
50	-	21942	-	20592
80	15621	41396	14906	37348
100	15656	48137	15252	44727
150	15668	70365	15612	69980
250	31239	102498	30670	114798
400	46810	168762	46715	174512
450	46863	188044	46862	199948
500	46912	208805	46909	191823

I couldn't find a function precise enough to calculate time for 10 and 50 sudokus, however looking at the number of function calls we can clearly derive that they have reduced significantly and hence less number of tries were required for reaching the solution

Also note, the number of reduced function calls by taking 450 and 500 sudokus, even though the number of sudokus tested has increased. This shows the power of random numbers because at 500 they A particular combination was obtained that reduced the function calls even though the sudokus improved.

You can find the dataset and the code in this repository itself data

This is important in brute force solutions because so that the bias is removed.

Future Scope

This can be further improved by using **bitmasks** and improving the helper functions. Also, the random generator can be made a little bit faster by utilizing various libraries. This method of using random numbers can be applied to various situations where the sequential bias exists

Conclusion

Using this Approach, a faster sudoku solver was obtained. It was faster due to the various reasons mentioned before. Also, the data obtained supports the logic.
The difference in computation time will become for significant as the magnitude of the dimensions of sudoku puzzle increases.

The number of function calls also decrease which was pretty much expected