Sudoku Solver

-

(via Backtracking)

Giovanni Matthew DiSalvo

CSCI 411

Professor Tillquist

March 21, 2023

Summary

A sudoku solver using the backtracking method is a straightforward approach which takes on solving a sudoku board by first looking at its options for its current cell (numbers 1-9), then considers constraints (columns, sections, rows), if it runs into through all numbers for a cell and still deems invalid, we backtrack to the previous answer we had filled in until we get something that would allow us to change the answer in our original problem cell (this is the backtracking step). This solver using the backtracking method has a runtime of O(n^m) where n is the number of possibilities for each cell (commonly nine) and m is the amount of unfilled spaces in a board.

What is Sudoku?

Sudoku is a puzzle game where you are given a board, in classic this is a 9 by 9 board, in which some of the board is already filled for you prior to you beginning (the amount of filled cells is varied upon the difficulty of each puzzle). You then can insert numbers ranging from 1-9 into any of the empty cells on the board. The board is split up into 9 subgrids each being of size 3 by 3. This is important because when placing your numbers, your number must be unique within the subgrid, horizontally, and vertically.

EXAMPLE BOARD

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

Here is an example of what a completed round of sudoku looks like after all the numbers have been placed in the correct positions. Note how each sub grid contains numbers 1-9, and they are all unique within their respective horizontal and vertical lines.

History

The game of Sudoku is a relatively new game in which the modernized version of the game, first being introduced in 1979, was designed anonymously by Howard Garns. Garns was a 74 year old retired architect and puzzle constructor from Connersville, Indiana. The puzzle was first introduced to Japan in which it received its name “Sudoku” (5). As for the backtracking Sudoku solving algorithm, it was the first of many algorithms to solve Sudoku which first came to light in 2004, developed by Wayne Gould in 2004. Wayne Gould is a retired judge from New Zealand who had acquired great interest in the game of Sudoku whilst on a visit to Japan (6). Gould had spent six years developing this Sudoku solving algorithm in which he had first started

with a basic brute-force approach and then continually added countermeasures that would accomplish more difficult Sudoku puzzles. Through trials and tribulations, he eventually came up with this final technique and tweaked it accordingly to boost its performance.

Why this method?

There are many other methods that solve Sudoku puzzles including those that use stochastic search, optimization methods, constraint programming, exact cover, relations and residuals. Although backtracking not only guarantees a solution every time, but the solving time is also unrelated to the difficulty of each individual puzzle, meaning the runtime of performing the hardest puzzle would run similarly to the same time of the easiest of puzzles. This is in part the nature of it being an NP-complete problem. The algorithm is also a strategy that can be used to solve puzzles by hand and is a method anyone can follow to solve Sudoku puzzles on their own. The downsides to the backtracking algorithm to solve Sudoku puzzles would be that the run time is often slow compared to those that use deductive methods. Although, the runtime of this algorithm can solve most puzzles in under a second on modern computers, which is much better than the brute force algorithm which took one programmer in 2008, six hours to get a final solution (to be fair, on an old system) for a Sudoku puzzle (2).

Intuition

The intuition of this algorithm is pretty straightforward and is something that goes through a series of actions and checks in a specific order. I will now go through each step and the process that we take to solve a Sudoku puzzle through backtracking:

(This example will be going off Classic Sudoku, but still applies to other variations, only changes would be numbers that can be placed within each cell and number of cells)

Step 1: We start from the cell in the first row and first column, first going from left to right through each row, looking for the first empty cell.

Step 2: Once we find an empty cell, we try to place a number 1-9 in the cell.

Step 3: If the number is valid then we continue to repeat steps 1-3, if it is not valid we go onto step 4.

(Major Step)
Step 4: If the number violates the rules of Sudoku of which we covered earlier, such as the number must not already be in the subgrid, the number must also be unique within their respective horizontal and vertical lines, then we must backtrack to a previous cell of which we have entered before and try other valid numbers. If there are no other valid numbers for said backtracked cell, it will continue this process of backtracking to previous cells until we find a cell we can swap to another valid number. After this, it will then proceed back forwards.

Step 5: The algorithm is complete when valid numbers fill all the empty cells. We have completed the Sudoku puzzle.

Pseudocode

- Parameter board is a two dimensional array representing the 9 by 9 board with the puzzle numbers already entered, and empty spots being signified by the number 0.

MAIN RECURSIVE FUNCTION

*here we perform the high level process of the algorithm as shown previously

```
Function sudokuSolver(board)
(*check to see if our board is complete before proceeding)
        If (isBoardFilled(board) == true)
                Return board

        Xcoord, Ycoord = nextEmptyCell(board)
        (* we are keeping track of our next empty cell using x, y, coordinates to help us track
        down the next target empty cell our algorithm is going to try and validate)
        For int j, 1 to 9
        (*loop for all possible numbers that can be placed within a cell)
                If (isValid(board, Xcoord, Ycoord, j) == true)
                        board[Xcoord][Ycoord] = j
                        If (sudokuSolver(board))
                                Return board
                        (*Important to note here, reason why we are making it zero again here is
because if we ever return to this state, we are doing so in need of backtracking, and zero
represents empty, so we will have to return to this set of coords and try it with other valid
numbers)
                        grid[Xcoord][Ycoord] = 0

        Return false
```

HELPER FUNCTIONS

- Our isValid function is going to check to see if the number we are trying to fill in, follows the rules of Sudoku in the given x-y coordinate on the board. This includes checking if any numbers within the same column match, same row match, and the subgrid. Then returns true/false, this is a major piece to our algorithm, it allows us to make sure that every single one of our inputs is true every step of the way and will make sure to guarantee a solution after every run.

```
Function isValid(board, Xcoord, Ycoord, number)
    For i 0 to 8
        If (board[Xcoord][i] == number)
            Return false
    For j 0 to 8
        If (board[j][Ycoord] == number
            Return false
    (here we are going to find which subgrid we are currently in)
    Row = Xcoord - (Xcoord % 3)
    Column = Ycoord - (Ycoord % 3)

    For z in Row to Row +2
        For d in Column to Column+2
            If (board[z][d] == number)
                Return false
    Return true
```

- The nextEmptyCell function is given the current state of the board and will return us the x-y coordinates of the next empty cell. isBoardFilled is practically the same function, but returns true/false depending if the board still has empty spaces or not.

```
Function nextEmptyCell(board)
    For x in 0 to 8
        For y in 0 to 8
            If (board[x][y] == 0)
                Return x, y
    Return -1
```

```
Function isBoardFilled(board)
    For x in 0 to 8
        For y in 0 to 8
            If (board[x][y] == 0)
                Return false
    Return true
```

Runtime Analysis

The runtime of the algorithm is very much dependent on the amount of possibilities that can be within each cell, this varies from Sudoku puzzles, but it is commonly 9 as most play Sudoku classic. Knowing this, here is the most important segment from our main recursive function contains the following:

```
For int j, 1 to 9
        If (isValid(board, Xcoord, Ycoord, j) == true)
                board[Xcoord][Ycoord] = j
                If (sudokuSolver(board))
```

We are going through 9 possibilities for each cell and we know that this function is recursive which introduces the point that our runtime is going to be exponential. Writing this into a recurrence equation can then be written as T(m) = 9*T(m - 1) + O(1). Solving this results in O(9^m) From this we can make the case that our runtime is O(9^m) for both our worst and average case. The m in this scenario represents the amount of empty cells to be filled. This varies upon the difficulty of the puzzles where the more difficult puzzles will have less empty cells to be filled, and the easier puzzles will have more empty cells. The 9 in O(9^m) represents the numbers that can be filled *within* each empty cell and on a classic sudoku board this is 9, but it can vary between different kinds of sudoku as if you were playing a 16 x 16 board, it would be O(16^m). Although the best case would be O(1) where we don't have to do any backtracking at all and just simply place numbers in the correct positions. This scenario is incredibly rare and almost never happens. Also, our pre-processing runtime we know would be O(n^2), with n being the amount of columns/rows we are going to be creating, because for the algorithm to run we are going to need to create a 2D array. For a classic sudoku we know we are going to be creating a 9x9 array so we know the pre-processing runtime would be O(9^2) because every cell must be filled, even the empty cells must be filled with 0.

Experimentation

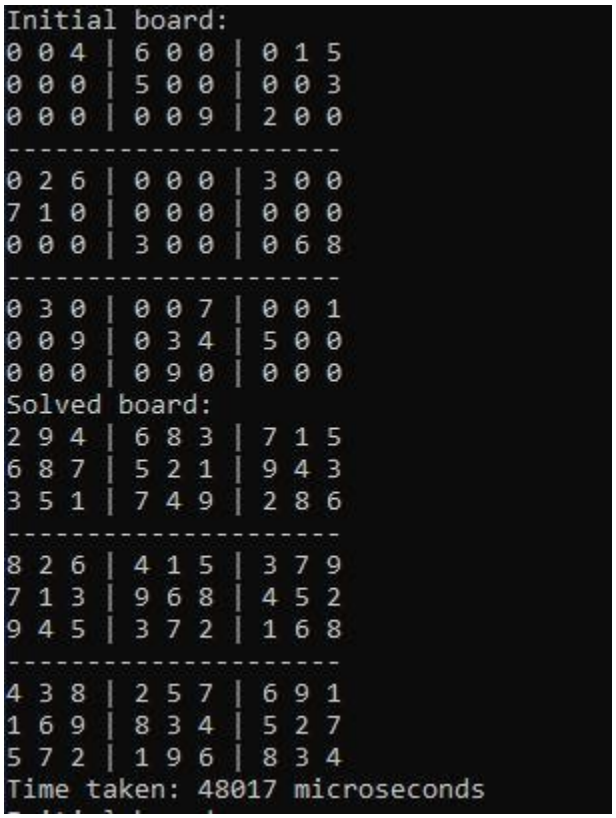
Here I implemented the previous pseudocode in C++ and used the chrono library to record the time it takes to complete the algorithm given a board in microseconds, note that this does not include the time to instantiate the arrays, as we are going to focus solely on the algorithm and it's computation time to complete the task of solving it. This will also make it more fair to compare the actual computation time when we compare it to a 16x16 Sudoku board later. For the first part of the experiment I wanted to look at given **easy level** Sudoku boards, how does our runtime look?

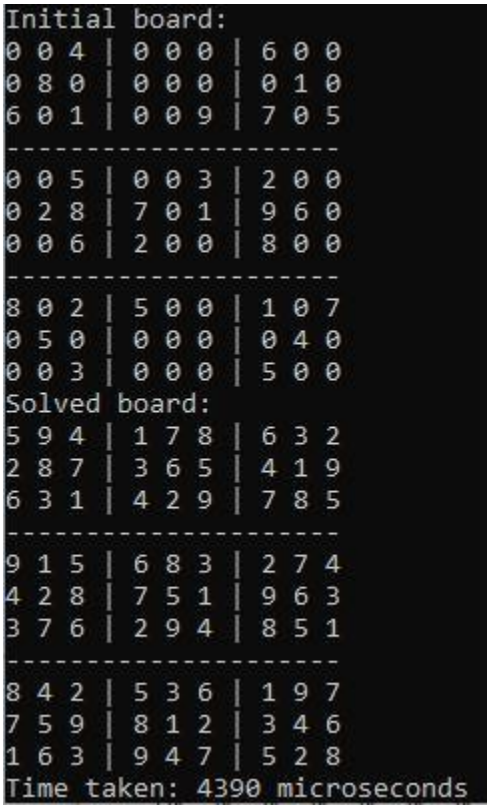
```
Initial board:
0 0 6 | 5 0 0 | 2 0 0
5 3 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 7 0
---------------------
0 2 0 | 6 0 0 | 0 0 0
0 0 0 | 0 7 0 | 0 0 0
0 0 0 | 0 0 8 | 0 0 0
---------------------
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 1 0 4
0 0 0 | 0 0 0 | 0 0 0
Solved board:
1 4 6 | 5 3 7 | 2 8 9
5 3 7 | 2 8 9 | 4 1 6
2 8 9 | 1 4 6 | 3 7 5
---------------------
3 2 1 | 6 5 4 | 7 9 8
4 6 8 | 9 7 1 | 5 2 3
7 9 5 | 3 2 8 | 6 4 1
---------------------
6 1 2 | 4 9 5 | 8 3 7
9 7 3 | 8 6 2 | 1 5 4
8 5 4 | 7 1 3 | 9 6 2
Time taken: 103 microseconds
```

```
Initial board:
0 0 0 | 2 6 0 | 7 0 1
6 8 0 | 0 7 0 | 0 9 0
1 9 0 | 0 0 4 | 5 0 0
---------------------
8 2 0 | 1 0 0 | 0 4 0
0 0 4 | 6 0 2 | 9 0 0
0 5 0 | 0 0 3 | 0 2 8
---------------------
0 0 9 | 3 0 0 | 0 7 4
0 4 0 | 0 5 0 | 0 3 6
7 0 3 | 0 1 8 | 0 0 0
Solved board:
4 3 5 | 2 6 9 | 7 8 1
6 8 2 | 5 7 1 | 4 9 3
1 9 7 | 8 3 4 | 5 6 2
---------------------
8 2 6 | 1 9 5 | 3 4 7
3 7 4 | 6 8 2 | 9 1 5
9 5 1 | 7 4 3 | 6 2 8
---------------------
5 1 9 | 3 2 6 | 8 7 4
2 4 8 | 9 5 7 | 1 3 6
7 6 3 | 4 1 8 | 2 5 9
Time taken: 15 microseconds
```

Given these two **easy level** Sudoku boards, we can see that our runtime is incredibly fast! Keep in mind that results can vary and the processing was done by an i7 13700k processor. Also, microseconds are 1x10^-6 seconds, so *1* microsecond is *0.000001* seconds. So our first puzzle

came in at a lightning fast 0.000104 seconds, and our second puzzle came in at 0.000015 seconds! Now those numbers are good, but they were easy puzzles, some of the best Sudoku players in the word could have done those puzzles, maybe not in microseconds, but in seconds. What if we tried the algorithm given **difficult level** Sudoku puzzles?

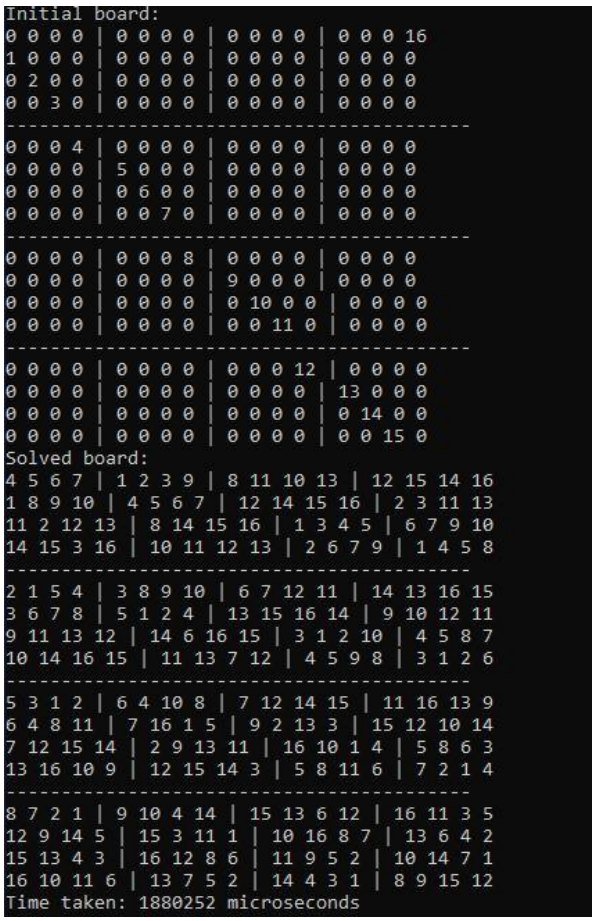




Now we can see from our results given the **difficult level** Sudoku puzzles that we see a substantial slow down when we ante up the difficulty in comparison to our runtime numbers on the easy level Sudoku puzzles. Given our current numbers, we were 466x slower on our slowest difficult level Sudoku puzzle to our slowest easy level Sudoku puzzle. For our fastest difficult level Sudoku puzzle we were 292x slower than our fastest easy level Sudoku puzzle. Although it does seem like we slowed down substantially, we still only took 0.048017 seconds to complete the slowest difficult puzzle we had, which is still incredibly fast, and faster than any human would take to even process the numbers given the puzzle.

16x16 Sudoku Runtimes

Now that we see that our 9x9 classic Sudoku puzzles were done in great times, how do Sudoku puzzles on a 16x16 board compare? We are retaining the same experiment parameters and are going to only alter the main pseudocode to support 16 columns/rows/terms and 4 subgrids. Here we are first going to do an **extremely easy level** 16x16 Sudoku puzzle.



*please excuse the solved board format, as the inputted numbers are now larger, it breaks the format

As we can see with a very simple Sudoku puzzle, it is now taking us 1,880,252 microseconds which is 1.9 seconds. This is a huge performance hit as we begin to start looking at 16x16 puzzles, as this puzzle was extremely easy in comparison to our easy 9x9 and took 18,250 times slower than our slowest easy 9x9 puzzle.

Difficult level 16x16

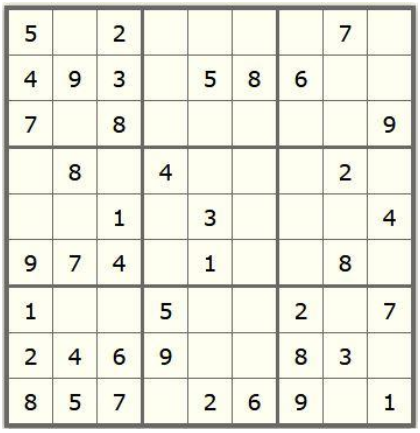
Now that we see the time it took to run an extremely easy 16x16 puzzle, what does a **difficult** 16x16 Sudoku puzzle's runtime look like?

```
Initial board:
0 15 0 1 | 0 2 10 14 | 12 0 0 0 | 0 0 0 0
0 6 3 16 | 12 0 8 4 | 14 15 1 0 | 2 0 0 0
14 0 9 7 | 11 3 15 0 | 0 0 0 0 | 0 0 0 0
4 13 2 12 | 0 0 0 0 | 6 0 0 0 | 0 15 0 0
-----------------------------------------
0 0 0 0 | 14 1 11 7 | 3 5 10 0 | 0 8 0 12
3 16 0 0 | 2 4 0 0 | 0 14 7 13 | 0 0 5 15
11 0 5 0 | 0 0 0 0 | 0 9 4 0 | 0 6 0 0
0 0 0 0 | 13 0 16 5 | 15 0 0 12 | 0 0 0 0
-----------------------------------------
0 0 0 0 | 9 0 1 12 | 0 8 3 10 | 11 0 15 0
2 12 0 11 | 0 0 14 3 | 5 4 0 0 | 0 0 9 0
6 3 0 4 | 0 0 13 0 | 0 11 9 1 | 0 12 16 2
0 0 10 9 | 0 0 0 0 | 0 0 12 0 | 8 0 6 7
-----------------------------------------
12 8 0 0 | 16 0 0 10 | 0 13 0 0 | 0 5 0 0
5 0 0 0 | 3 0 4 6 | 0 1 15 0 | 0 0 0 0
0 9 1 6 | 0 14 0 11 | 0 0 2 0 | 0 0 10 8
0 14 0 0 | 0 13 9 0 | 4 12 11 8 | 0 0 2 0
Solved board:
8 15 11 1 | 6 2 10 14 | 12 7 13 3 | 16 9 4 5
10 6 3 16 | 12 5 8 4 | 14 15 1 9 | 2 11 7 13
14 5 9 7 | 11 3 15 13 | 8 2 16 4 | 12 10 1 6
4 13 2 12 | 1 9 7 16 | 6 10 5 11 | 3 15 8 14
-----------------------------------------
9 2 6 15 | 14 1 11 7 | 3 5 10 16 | 4 8 13 12
3 16 12 8 | 2 4 6 9 | 11 14 7 13 | 10 1 5 15
11 10 5 13 | 8 12 3 15 | 1 9 4 2 | 7 6 14 16
1 4 7 14 | 13 10 16 5 | 15 6 8 12 | 9 2 3 11
-----------------------------------------
13 7 16 5 | 9 6 1 12 | 2 8 3 10 | 11 14 15 4
2 12 8 11 | 7 16 14 3 | 5 4 6 15 | 1 13 9 10
6 3 14 4 | 10 15 13 8 | 7 11 9 1 | 5 12 16 2
15 1 10 9 | 4 11 5 2 | 13 16 12 14 | 8 3 6 7
-----------------------------------------
12 8 4 3 | 16 7 2 10 | 9 13 14 6 | 15 5 11 1
5 11 13 2 | 3 8 4 6 | 10 1 15 7 | 14 16 12 9
7 9 1 6 | 15 14 12 11 | 16 3 2 5 | 13 4 10 8
16 14 15 10 | 5 13 9 1 | 4 12 11 8 | 6 7 2 3
Time taken: 4260340 microseconds
```

As we can see, our runtime once again jumped up to 4.26 seconds as we passed a difficult Sudoku puzzle, and as we incremented our difficulty and our board size our runtime has exponentially increased. This difficult puzzle was still 88x slower than our slowest 9x9 puzzle, but 4.26 seconds is still incredibly quick for a 16x16 puzzle. As we have seen from our experiments, we could assume that 25x25 puzzles would have similar gaps in runtimes from our 9x9 to 16x16 results as it would from 16x16 to 25x25. It is also safe to assume that even though we are using one of the lesser efficient algorithms in comparison to the new algorithms such as those Sudoku solving algorithms that use stochastic search, optimization methods, constraint programming, exact cover, relations and residuals, our runtimes with backtracking is still within relatively good speed for the subject use we are currently using it for, with 9x9 and 16x16 boards with even the highest of difficulties being solved in seconds.

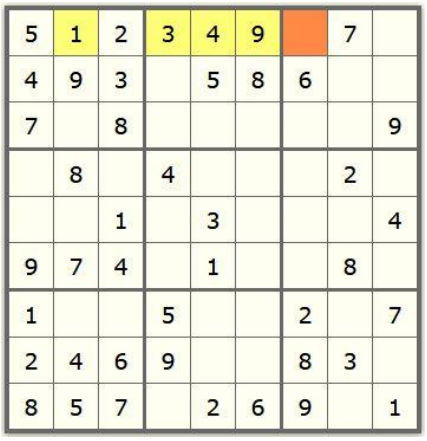
Visualization of Intuition

First we start off with our initial puzzle.



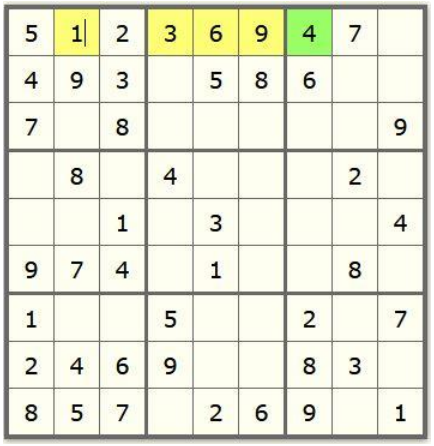
5		2					7	
4	9	3		5	8	6		
7		8						9
	8		4				2	
		1		3				4
9	7	4		1			8	
1			5			2		7
2	4	6	9			8	3	
8	5	7		2	6	9		1

Then we fill our puzzle with numbers 1-9 into our puzzle until we hit a cell where no numbers 1-9 are valid, here we find that if we attempt to put any number 1-9 we are blocked for the most part because of the horizontal and vertical uniqueness, so we find our first problem cell.



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

Then we must backtrack through our previously inputted numbers and find which inputted cells we can switch to a different number, in order to validate our current problem cell. For this scenario it happened that



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

we needed to switch our previously inputted 4, to a 6, so we can use that 4 in a more optimal position, and validate our puzzle.

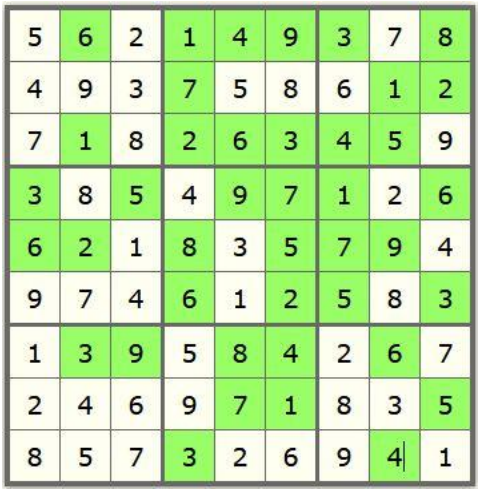
Continuing, we find our next problem cell, to be on the end of the second row, being the problem cell at the end of the row typically means that we are running out of horizontal numbers to remain unique without breaking the verticality of being unique which is what is going on here.

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

We then backtrack two previously inputted cells away from the left to find that switching our previous 2 to a 7 allows us to validate our current problem cell, and we can then continue.

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

We repeat this process until our puzzle is completely filled with valid cells. We completed our puzzle!



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

Presentation Feedback Given

Reviewer 1 Boxi Chen: Good explaining what and why we use this algorithm, nice that it has a history background to support it,good visualization, but how to show or explain backtrack work will help while you are showing pictures. Less words, short points will help me read slides
Is there any other way to solve the problem?

- If I present this via slides again this is something I definitely need to improve on.

Reviewer 2 Adrian Valencia: Great introduction explaining the game of Sudoku and the history behind the game and the development of the algorithm. Good explanation behind the intuition of the algorithm. The visual representation of how the pseudocode traverses the Sudoku puzzle enhances the clarity immensely! The run time is a bit tricky to assess for this algorithm, but your breakdown of your understanding of it was clear and concise and consistent with your pseudocode! Well done.

- Confirmed my analysis + made some additional explanation and dug deeper

Reviewer 3 Jacob Dodson: The introduction of the algorithm helps me understand the pseudo code. The one thing that would help clear up my confusion is having the images next to each other when explaining how the pseudocode works. Trying to fully visualize the previous step and the current one was slightly confusing.

- Added some text to clarify what is happening during all of my visualizations and why we are doing it

Reviewer 4 Lang Bledsoe: Great introduction. When explaining the rules, I would highlight what a row, column, and square are so that the viewer/reader can better understand. Your visuals are great, but I would use more to show how the program works as it runs, as Sudoku is something that can easily be visualized. Also, what does ‘m’ represent in your recurrence relation?

- Added what m represents and further explained it

Works Cited

"Implement A Sudoku Solver - Sudoku Solving Backtracking Algorithm ('Sudoku Solver' on LeetCode)." *YouTube*, YouTube, 8 Jan. 2019, https://www.youtube.com/watch?v=JzONv5kaPJM&ab_channel=BackToBackSWE. Accessed 23 Mar. 2023.

"Sudoku Solving Algorithms." *Wikipedia*, Wikimedia Foundation, 16 Nov. 2022, https://en.wikipedia.org/wiki/Sudoku_solving_algorithms.

Kaufman, Anna. "What Is Sudoku? Here's How to Play the Math Puzzle and Some Tips to Solve One." *USA Today*, Gannett Satellite Information Network, 24 Feb. 2023, https://www.usatoday.com/story/life/2022/08/12/what-is-sudoku-solve-puzzle/10299742002/.

"Sudoku." *Wikipedia*, Wikimedia Foundation, 15 Mar. 2023, https://en.wikipedia.org/wiki/Sudoku.

Gould, Wayne. "How to Solve Sudoku Puzzles Using a Computer Algorithm." Journal of Recreational Mathematics, vol. 34, no. 1, 2005, pp. 1-14.