Abstract

Developers have tried to find algorithms in order to generate a variety of puzzles for human players so that they could be even solved by computer programming.

In this Report, We will be discussing the Problem of Generation and Solving of Sudoku Problem, Analyzing the Various Approaches to Problem and Also Implementing the Solution in User Interactive Manner [Enhanced UI]

The purpose is to implement a more efficient algorithm and then compare it with another Sudoku solving Algorithm.

The Results have proven that Crook's Algorithm is faster than the Normal Backtracking Solution, But in Some Cases Fails to Reach the Right Results.

1.INTRODUCTION

1.1 What is Sudoku?

Sudoku is a logic-based, combinatorial *number placement* puzzle game where the objective is to fill a square grid of size 'n' with numbers between 1 to 'n'.

The numbers must be placed so that :

- each column,
- each row, and
- each of the sub-grids (if any) contains all of the numbers from 1 to 'n'.

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

1.2 Problem Statement

Problem Description: You are given a Sudoku puzzle and you need to fill the empty cells without violating any rules.

A sudoku solution must satisfy all of the following rules:

1. Each of the digits [1-9] must occur exactly once in each row.
2. Each of the digits [1-9] must occur exactly once in each column.
3. Each of the digits [1-9] must occur exactly once in each of the 3x3 sub-boxes of the grid.

1.3 Are All Sudoku Solvable?

The answer is: **YES**

Unless it doesn't Violate the 3 Standard Rules of Sudoku [Row, Column, Grid]

Any Sudoku can have:

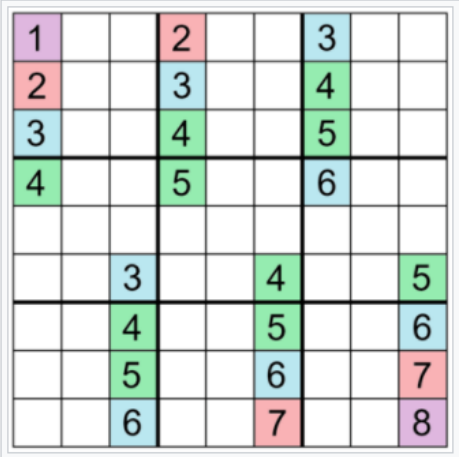
- ★ Unique Solution
- ★ Multiple Solution [Where we need to Guess]

1.4 Maths Behind Sudoku [Hints & Difficulty]

The main results are that for the classical Sudoku the number of filled grids is 6,670,903,752,021,072,936,960 (6.67×1021), which reduces to 5,472,730,538 essentially different groups under the validity preserving transformations.

There are **26 types of symmetry**, but they can only be found in about **0.005%** of all filled grids.

A puzzle with a **unique** solution must have **at least 17** clues, and there is a solvable puzzle with **at most 21 clues** for every solved grid. The largest minimal puzzle found so far has **40 clues**.



A 24-clue automorphic Sudoku with translational symmetry

The difficulty level of Sudoku puzzles depends on how the given numbers are placed in the Sudoku board and also how many numbers (clues) are given. Generally, the most significant aspect of difficulty ratings of Sudoku puzzles is which techniques

are required to solve the puzzles. In other words, it is important where the given numbers are placed logically.

The Puzzles, which need more techniques to solve, can be named as the difficult one. On the other side there are puzzles that can be solved by using simple methods and this kind of puzzles can be defined as easy or medium level. As mentioned above, there are four difficulty levels that are used in testing (easy, medium, hard and evil). We have found during testing that the classification of difficulty levels is not easy as it is stated. This is due to the fact that there have been puzzles which marked a hard level but they had been solved using simple techniques and vice versa.

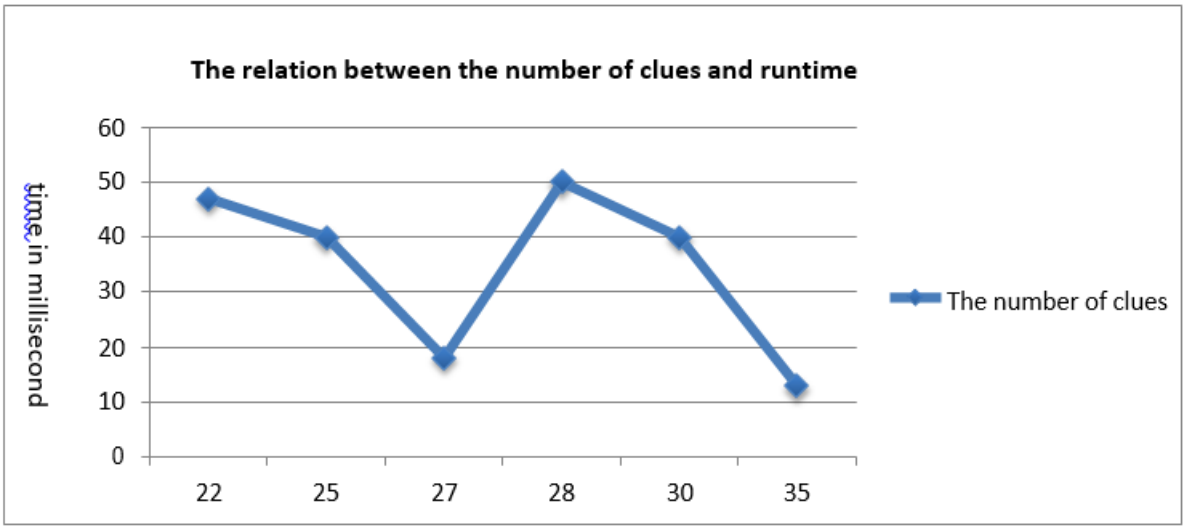


***Diagram 3**. This diagram shows the relationship between the number of clues (in the horizontal axis) and the run-time of the puzzles (in the vertical axis).*

There are people who believe that the difficulty ratings have to do with the number of revealed numbers on the puzzle board. Generally, a Sudoku puzzle needs at least 17 clues to be solvable. It means that solving a Sudoku puzzle with 17-clues is more difficult than a puzzle with 30-clues. The more numbers are given, the easier and quicker the solution is. This statement may not always be true, since the testing has shown that the puzzles with fewer clues could be solved in a shorter time than the puzzles with more clues. Diagram 3 below describes the relationship between the number of clues in the puzzles and their run-time. It might be expected that if the number of clues becomes more, the run-time of solving the puzzle would be shorter. For instance, when solving the puzzle with 28 clues the solving time increases rapidly. The reason is that the puzzle needs more techniques to solve it or the algorithm needed to iterate as long as the solution is found.

2.ALGORITHMS TO SOLVE PROBLEM

2.1 Pen and Paper Algorithm

(1) Unique missing candidate

- Check if a Row/Column/Grid has only 1 Number Left

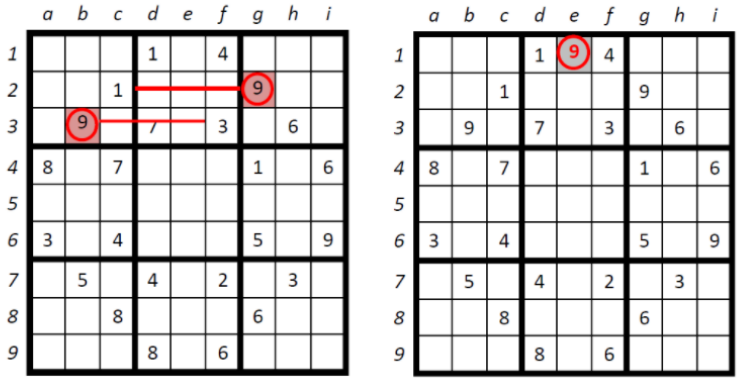
1	2	4	?	7	5	6	8	9
5		6	8	4		7	1	
7		9	1		6	3		5
2	1		4	5		8	9	6
4		7	6		8	1		3
6	9	8	2	1			5	7
	4	1	5		2		7	8
8		2	9			5		4
9	6	5		8	4	2		1

(2) Naked Single

- Find a square that takes a single value, considering squares in the same row, column and box.

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
1				1		4			
2			1				9		
3		9		7		3		6	
4	8	?	7				1		6
5							3		
6	3		4				5		9
7		5		4		2		3	
8			8				6		
9				8		6			

(3) Hidden Single



2.2 Brute Force Algorithm/ Backtracking / Depth for Search

Backtracking is an algorithm for finding all (or some) of the solutions to a problem that incrementally builds candidates to the solution(s). As soon as it determines that a candidate cannot possibly lead to a valid solution, it abandons the candidate. Backtracking is all about choices and consequences.

We will now create a Sudoku solver using backtracking by encoding our problem, goal, and constraints in a step-by-step algorithm. A brute force algorithm visits the empty cells in sequential order, filling in digits sequentially, or backtracking when no digit can be filled without violating any rule.
Initially, the program would solve the puzzle by placing the digit “1” in the first empty cell and checking if it is allowed to be there. If there are no violations (checking row, column and box constraints) then the algorithm advances to the next cell and places a “1” in that cell. When checking for violations, if it is discovered that the “1” is not allowed, the value is changed to “2”. If a cell is discovered where none of the 9 digits is allowed, then the algorithm leaves that cell blank and moves back to the previous cell. The value in that cell is then incremented by one. This is repeated until an allowed value in the last empty cell is discovered.

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

If we try to draw the recursion tree then each step will branch into 9 branches and at each level, the problem size is reduced by 1.

```
Find an empty cell with coordinates (row, col).
if no cell is found then
    return true;
end
for digits x from 1 to 9 do
    if we can place x at position (row, col) and the puzzle remains valid then
        recursively continue to fill the rest of the puzzle;
        if recursion succeeds i.e. returns true then
            return true
        end
        else
            free cell and try with the next digit;
        end
    end
end
if none of the digits lead to a solution then
    return false
end
```

The unique missing method and the naked single method are able to solve all puzzles with easy and medium level of difficulties. In order to solve puzzles with even more difficult levels such as hard and evil the backtracking method has been used to complete the algorithm. A human player solves the puzzle by using simple techniques. If the puzzle is not solvable by using the techniques the player then tries to fill the rest of the empty squares by guessing.

The backtracking method, which is similar to the human strategy (guessing), is used as a help method to the pencil-and-paper algorithm. In other words, if the puzzle cannot be filled when using the unique missing method and the naked single method, the backtracking method will take the puzzle and fill the rest of empty squares. Generally, the backtracking method finds an empty square and assigns the lowest valid number in the square once the content of other squares in the same row, column and box are considered. However, if none of the numbers from 1 to 9 are valid in a certain square, the algorithm backtracks to the previous square, which was filled recently.

The above-mentioned methods are an appropriate combination to solve any Sudoku puzzles. The naked single method can find quickly single candidates to the

empty squares that need only one single value. Since the puzzle comes to its end solution the unique missing method can be used to fill the rest of the puzzles. Finally, if either method fills the board the algorithm calls the backtracking method to fill the rest of the board.

The time needed for the algorithm to run is not directly related to the difficulty of the puzzle. It is dependent on the number of times the recursion is repeated (the function calls itself). There have been puzzles constructed such that the performance of the

2.3 Crook's Algorithm

Step1: Markup

The first step is easy, list out all possible numbers in each cell.
If you have used an app to play Sudoku, there should already be a function allowing you to write down some possible numbers in a cell. In the first step of the algorithm, for each cell, you follow the rules and write down all possible numbers.

Step2: Find singleton

The team "singleton" seems so complicated. However, the concept is easy, finding if there is a row, column, or box with only one possible value throughout the row, column, or box. Since there is one and only one number in each row, column, and box, if there is only a cell in the row, column, or box with the respective number, this number must be in this cell. So you fill in this cell with this number and then update markups in an affected row, column, or box.

Step3: Find Preemptive Sets

This step is quite difficult as I spent a lot of time to understand this part. Below is the definition of preemptive sets from Crook's paper.

A preemptive set is composed of numbers from the set [1, 2, . . . , 9] and is a set of size m, 2 ≤ m ≤9, whose numbers are potential occupants of m cells exclusively, where exclusively means that no other numbers in the set [1, 2, . . . , 9] other than the members of the preemptive set are potential occupants of those m cells.

A preemptive set is denoted by {[$n_1, n_2, \ldots, n_m$], [c (i_1, j_1), c(i_2, j_2), . . . , c(i_m, j_m)]}, where [$n_1, n_2, \ldots, n_m$], $1 \leq n_i \leq 9$ for $i = 1, 2, \ldots, m$, denotes the set of numbers in the preemptive set and [c (i_1, j_1), c(i_2, j_2), . . . , c(i_m, j_m)] denotes the set of m cells in which the set [$n_1, n_2, \ldots, n_m$], and subsets thereof, exclusively occur.

The range of a preemptive set is a row, column, or box in which all of the cells of the preemptive set are located. When m = 2 or 3, the range can be one of the sets [row,box] or [column, box] .

To simplify it, if m numbers are a set or a superset of a markup of m cells within a row, column, box, [row, box] or [column, box], then this combination of numbers and cells are a preemptive set.

Step4: Eliminate possible numbers outside preemptive sets

This step is the most important part of the algorithm and this step helps reduce possible numbers of cells.

Let X be a preemptive set in a Sudoku puzzle markup. Then every number in X that appears in the markup of cells not in X over the range of X cannot be a part of the puzzle solution.

In this step, for all numbers in a preemptive set, they cannot be possible numbers for a cell outside the preemptive set. The reason here is simple, as each number can be once only in each row, column or box. If this number is in the preemptive set, this number must be placed in those cells in the set. As a result, all cells in the same row, column and box cannot have this number as a possible number in the markup.

3.ANALYSIS OF ALGORITHM

3.1 Backtracking Approach – Complexity Analysis

Time Complexity: O(n ^ m) where n is the number of possibilities for each square (i.e., 9 in classic Sudoku) and m is the number of spaces that are blank.

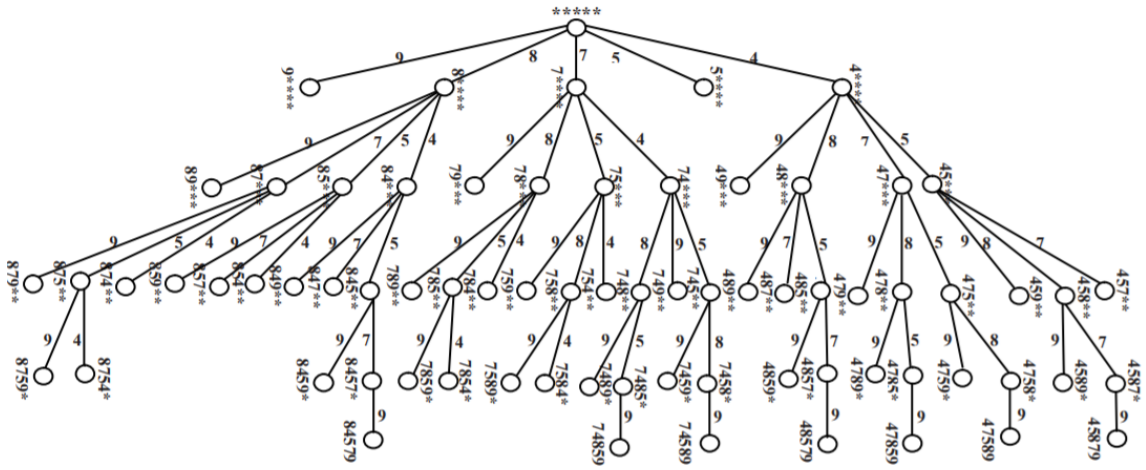
The problem can be designed for a grid size of N*N where N is a perfect square. For such an N, let M = N*N, the recurrence equation can be written as

$$\mathbf{T(M) = 9*T(M-1) + O(1)}$$

where T(N) is the running time of the solution for a problem size of N. Solving this recurrence will yield, O(9^M).

Explanation: This can be seen by working backwards from only a single empty spot. If there is only one empty spot, then you have 'n' possibilities and you must work through all of them in the worst case. If there are two empty spots, then you must work through n possibilities for the first spot and n possibilities for the second spot for each of the possibilities for the first spot. If there are three spots, then you must work through n possibilities for the first spot. Each of those possibilities will yield a puzzle with two empty spots that now have n^2 possibilities.

You may also say that this algorithm performs a depth-first search through the possible solutions. Each level of the graph represents the choices for a single square. The depth of the graph is the number of squares that need to be filled. With a branching factor of n and a depth of m, finding a solution in the graph has a worst-case performance of O(n ^ m).



Space complexity: it's the recursion stack that is used as an auxiliary space which is N*N step deep. Remember we need to fill in 81 cells in a 9*9 sudoku and at each level, only one cell is filled. So, space complexity would be O(M).

Complexity Analysis:

- **Time complexity:** O(9^(n*n)).
  For every unassigned index, there are 9 possible options so the time complexity is O(9^(n*n)). The time complexity remains the same but there will be some early pruning so the time taken will be much less than the naive algorithm but the upper bound time complexity remains the same.
- **Space Complexity:** O(n*n).
  To store the output array a matrix is needed.

3.2 Comparative Performance Analysis

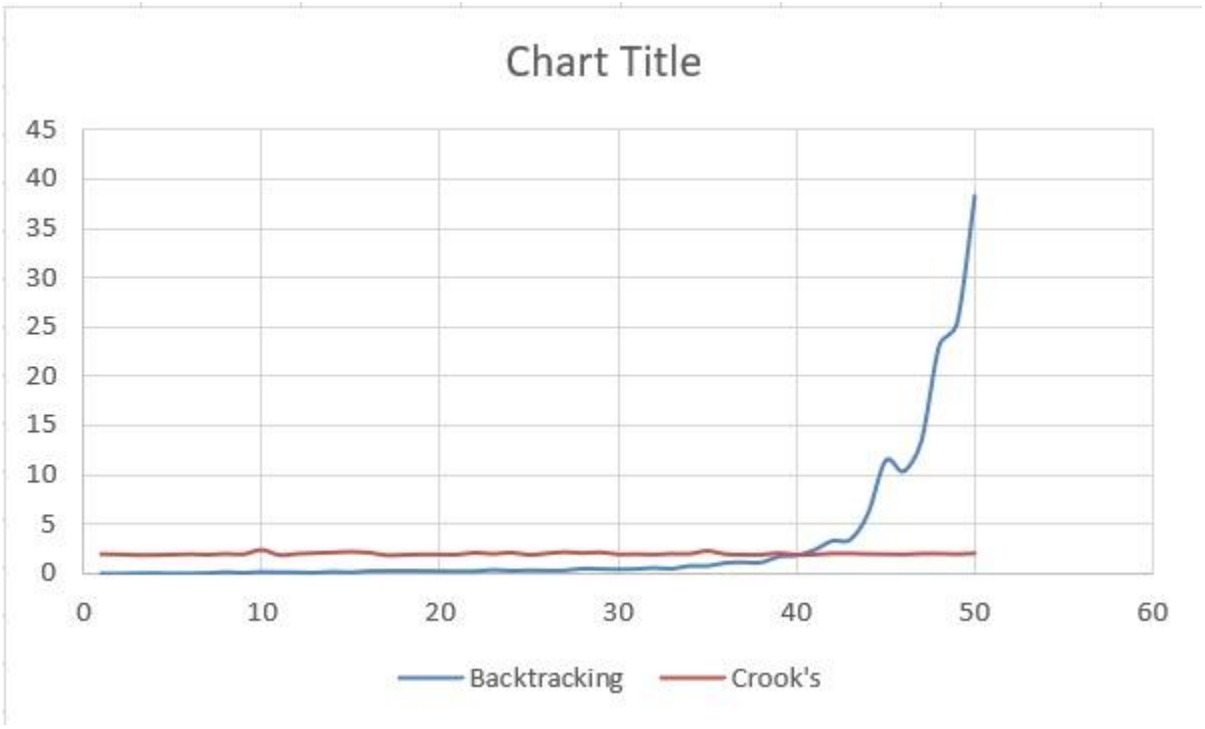


Diagram 1. *A comparison between the backtracking and Crook's algorithm*

This diagram depicts the differences between the pencil-and-paper algorithm and the brute force algorithm based on how long it takes to solve the puzzles by a computer. The Crooks algorithm solves the puzzle quicker than the brute force algorithm. The given data is based on the averaging of the computing time for several puzzles that have been tested with the same difficulty levels such as easy, medium, hard, and evil respectively. For instance, the time obtained for the easy level is the result of averaging several computing times with the easy level. In the given diagram the vertical axis represents the running time of the puzzles in milliseconds and the horizontal axis the difficulty levels.

Generally, the backtracking method, which is similar to the brute force algorithm, can solve the puzzles quicker than the pencil-and-paper algorithm. The question now is why should we use backtracking and human methods together? There are three reasons to do so. Firstly, the purpose of this work is to implement an algorithm applying human strategies. Secondly, human players also use the backtracking method when they get stuck. It means that players check different alternatives and place the numbers in the empty squares by guessing when there are no options left. Finally, employing human strategies makes the algorithm more efficient based on the number of comparisons. In other words, the naked single method fills the empty squares by performing fewer comparisons in a short time as

it uses a better technique. However, the backtracking method runs more number of analytical circulation while solving the puzzles resulting in the consumption of memory space.

4.FEATURES OF OUR PROJECT

4.1 Valid Sudoku Generation (API)

Our solution is based on 5 steps:

1. First generate a full grid of numbers. This step is more complex as it seems as we cannot just randomly generate numbers to fill in the grid. We have to make sure that these numbers follow the Sudoku rules. We will do this in 2 parts:

We will take an empty grid and then fill all the diagonal sub-grids. We can observe that in the above matrix, the diagonal matrices are independent of other empty grid initially. So we only need to check that the inserted numbers are not repeated in the respective subgrid.

Then the next step will be to fill the remaining 6 sub-grids which are not part of the diagonal. We will do this using backtracking, i.e, try all numbers until a safe number to fill empty space is found, check for the next empty space and if no possible number is found we backtrack.

Once the grid is filled, remove K elements randomly in such a way that after removing the grid will have a unique solution. It can be done using a backtracking sudoku solver.

3. Each time a value is removed we will apply the sudoku solver algorithm to see if the grid can still be solved and to count the number of solutions it leads to. If the resulting grid only has one solution we can carry on the process from step 3. If not we will have to put the value we took away back in the grid. We have to continue doing this until k values are removed from the grid

Based on the number of removals from the completed grid, the resulting grid will be more difficult to solve.

4.2 Backend API

The Flask API used in this project takes request from the User to:

- generate sudoku with user selected number of missing squares
- solve sudoku selected by the user

It does the job of generating sudoku by using the generator algorithm, then send the sudoku board as a JSON object to the User. Similarly, when the user selects to solve sudoku using any of the two available methods (Backtracking or Crook's Algorithm), the incomplete sudoku board is received by the Flask API, solves it using the respective algorithm and sends the solved board as a JSON object. This Flask API also calculates the time taken to solve the received sudoku by the given method and sends it within the JSON object so that the user can view and compare the durations.

4.3 Front End Webpage

This gives a UI Interface to the user with buttons for generating and solving sudoku and displays time taken to solve the sudoku by the algorithm.

Source Code Link:

https://github.com/BhagyaRana/SUDOKU-SOLVER

Individual Contribution in Github Repository :

https://github.com/BhagyaRana/SUDOKU-SOLVER/graphs/contributors

5.RESULTS

	Backtracking method	Pencil and paper method
Strengths	Guarantees a solution and can discover multiple solutions if they exist(carries out an exhaustive search)	If puzzle conforms to one or more of its rules a solution is obtained speedily
Weakensses	It is time and memory consuming	If none of the rules can be applied to the puzzle it fails to obtain a solution

6.CONCLUSION

We have Successfully Learned the Mathematical Logic Working Behind the Sudoku and How to Computationally Solve this Problem in Step-By-Step Manner.
We have also Applied the Development Knowledge to Enhance the User Experience [Instead of Running on Terminal]
We have also Successfully Understood the BackTracking Concept and Applied it in Our Project Sudoku Solver!

7.FUTURE SCOPE

(1) This Project can be Integrated with Computer Vision [AR,OpenCV & Machine Learning] to Extract Sudoku from Newspaper and Solve it in Real Time & Display on Paper.
(2) Instant Help by Giving Hints and Users Progress while Playing
(3) Score System Based on Time and Accuracy [LeaderBoard and Game Development]

8.REFERENCES

(1) https://en.wikipedia.org/wiki/Mathematics_of_Sudoku
(2) https://en.wikipedia.org/wiki/Sudoku_solving_algorithms
(3)
(4) https://afteracademy.com/blog/sudoku-solver
(5) http://www.ams.org/notices/200904/rtx090400460p.pdf
(6) https://towardsdatascience.com/solve-sudoku-more-elegantly-with-crooks-algorithm-in-python-5f819d371813
(7) https://github.com/bertoort/sugoku
(8) https://github.com/autopear/Sudoku-Solver
(9) https://github.com/tt-mp/sudoku
(10) https://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand13/Group1Vahid/report/Aref-Fiorella-KexJobb-sist.pdf